Rapport de Stage

Comment les grands modèles de langage parviennent-ils à jouer aux jeux de plateau ?

Luc Pommeret

Stage du 1er mai 2024 au 1er octobre 2024,
à l'Institut de Recherche en Informatique Fondamentale (IRIF)
sous la direction de Michel de Rougemont

MASTER LOGOS

INSTITUT
DE RECHERCHE
EN INFORMATIQUE
FONDAMENTALE

Université
Paris Cité

Remerciements

Je tiens à remercier Michel de Rougemont et Achraf Lassoued de m'avoir poussé toujours plus loin dans mes recherches, et de m'avoir suivi lorsque je m'engageais dans des *terra incognita*. Nous nous retrouvions chaque semaine, et je devais faire une présentation de mes recherches (avec diaporama) à chaque fois. La discipline d'Achraf liée aux commentaires éclairants de Michel m'ont bien aidé à avancer !

Je souhaite également remercier l'association APIA de m'avoir accueilli trois jours à La Rochelle à la PFIA 2024 pour présenter mes recherches sur le bruit dans ChessGPT.

Je souhaite remercier Adam Karvonen [10] pour m'avoir donné des idées, et pour les discussions utiles que nous avons eues.

Un grand merci à Claude 3.5 d'Anthropic [19] et à ChatGPT d'OpenAI [16] d'avoir débuggé et généré un grand nombre de lignes de code, ce qui a grandement accéléré mes recherches[1] !

Je remercie bien sûr l'IRIF et Claire Mathieu qui ont financé mon stage ainsi que mon voyage à La Rochelle, qui fut très agréable.

Et puis Youna de m'avoir accompagné à La Rochelle et de m'avoir encouragé lors de ma présentation.

1. Je pense qu'il faut les remercier. Mais ils posent question : jusqu'où vont les remerciements ? Faut-il remercier Python, ou encore ma machine d'avoir bien fonctionné ?

Table des matières

Notebooks et ressources

Les notebooks et ressources associés à ce travail sont disponibles aux liens suivants :

— **PoèteGPT** : Notebook n°1
— **Grokking** : Notebook n°2
— **PRH** :
  — Notebook n°3
  — Article de blog
— **SAE et Idéaux-type** :
  — Notebook sur la neurochirurgie des transformers
  — Notebook sur les idéaux-types

Chapitre 1

Introduction

Πάντες ἄνθρωποι τοῦ εἰδέναι ὀρέγονται φύσει[a]

---

a. « Tous les hommes désirent naturellement savoir. »
Aristote, *Métaphysique*

Les grands modèles de langage (LLM, pour Large Language Models) ont démontré ces cinq dernières années des capacités étonnantes dans de très nombreux domaines, allant de la génération de texte à la résolution de problèmes complexes, en passant par l'analyse de données et la traduction automatique. La particularité de ces modèles généralistes réside dans l'émergence de capacités en fonction du nombre de paramètres, du volume et de la qualité des données d'entraînement, ainsi que du temps d'apprentissage. Les modèles de langage dits « génératifs », tels que ChatGPT et ses homologues, sont devenus les plus célèbres représentants de cette technologie. Leur fonctionnement repose sur la génération récursive du prochain token à partir du contexte qui précède, créant ainsi des séquences de texte cohérentes, et parfois étonnamment « humaines ».

L'histoire de l'informatique nous montre que trois âges ont été traversés, qui semblent mus par une même idée, mais avec différents moyens. Le premier âge est celui que l'on pourrait qualifier d'âge du calculable. Le but était d'étudier les fonctions qui sont calculables, et de trouver un critère qui distingue celles qu'on ne peut pas calculer. C'est l'époque de Turing, des formalismes permettant d'émuler toute fonction calculable (λ-calcul, machine de Turing…). Le second âge est celui des algorithmes, où à la nécessité de la calculabilité nous ajoutons des propriétés sur celle-ci : la complexité en est une, et pas des moindres, elle décide si une fonction calculable peut être mise en pratique ou non. Le troisième âge est celui que nous vivons, où les algorithmes s'intéressent à des distributions statistiques, et qui, en abandonnant le monde de la certitude, obtiennent

des performances inégalables par les algorithmes du monde logique. Le Transformer est à la pointe de cet âge nouveau, et par sa fonction même : en sortie, il produit une distribution de probabilité, qu'il suffit ensuite d'échantillonner pour obtenir… du langage par exemple.

Dans le cadre de ce stage, j'ai choisi d'utiliser l'univers déjà balisé des jeux de plateau pour étudier plus en détail le fonctionnement des transformers, l'architecture sous-jacente de ces LLM. Ce choix n'est pas anodin : les jeux de plateau offrent un environnement contrôlé avec des règles définies, permettant d'observer plus facilement l'émergence de stratégies et de comportements complexes. En particulier, je me suis intéressé à la mise en place de stratégies (gagnantes ou non) pour comprendre certaines propriétés fondamentales de ces modèles.

Les réseaux de neurones profonds, et *a fortiori* les transformers, sont pour les humains de véritables boîtes noires quasiment impossibles à interpréter directement. Pour de nombreuses raisons[1], il serait souhaitable de mieux comprendre ce qui se passe dans ces modèles complexes. Cette compréhension pourrait non seulement améliorer la transparence et la fiabilité des systèmes d'IA, mais aussi permettre de les optimiser et de les rendre plus efficaces dans diverses applications. Le but de ce stage était donc d'approfondir notre compréhension des capacités émergentes des LLM.

Au fil de mes recherches, j'ai progressivement adopté l'angle de l'interprétabilité (ou *explainability*), qui consiste à utiliser différentes méthodes pour établir un lien entre l'état des couches du réseau de neurones et des propriétés logiques observables. Parmi ces méthodes, on peut citer l'analyse des activations neuronales, la visualisation des représentations internes, ou encore l'étude des mécanismes d'attention. En somme, ce que j'essaye de faire avec les nombreux chercheurs qui s'intéressent à ce domaine encore très inexploré, c'est retrouver la logique dans ce brouhaha statistique qu'est le réseau de neurones.

1.1 Tokénisation

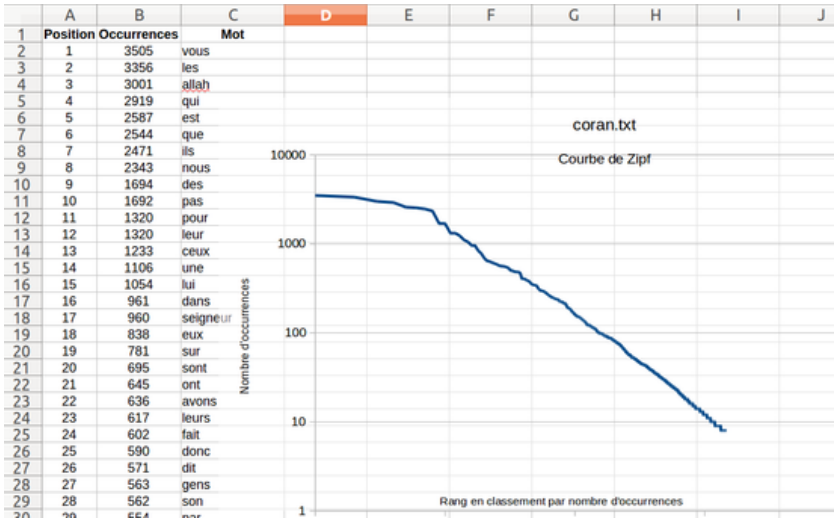
La première étape de la construction d'un LLM—parfois sous-estimée, bien que les choix qui y sont faits soient primordiaux— est la tokénisation. Elle consiste à sectionner les données d'entraînement en unités syntaxiques simples, qui sont les plus petits niveaux que le modèle va "voir".

Il existe différentes manières de tokeniser des données :

— **Tokénisation par caractère** : chaque caractère devient un token. C'est le choix que j'ai fait pour étudier les jeux. Cette approche a l'avantage d'être simple et de produire un vocabulaire de taille limitée, mais peut nécessiter des séquences plus longues pour capturer du sens.

1. À la fois politiques (on peut penser à l'*AI Act*), scientifiques (pour améliorer nos connaissances fondamentales) et économiques (pour optimiser et mieux contrôler ces systèmes).

— **Tokénisation statistique** : on se fonde sur des régularités statistiques des langues humaines, comme la loi de Zipf, qui donne la proportion de la fréquence de chaque mot selon son ordre dans la fréquence. Cette approche est notamment utilisée par les algorithmes comme BPE (Byte-Pair Encoding) ou WordPiece.
— **Tokénisation informée** : on fixe notamment les entités nommées comme "Aéroport Charles De Gaulle", "Banque de France", etc. Cette approche nécessite une expertise du domaine mais peut améliorer significativement les performances sur des tâches spécifiques.



Position	Occurrences	Mot
1	3505	vous
2	3356	les
3	3001	allah
4	2919	qui
5	2587	est
6	2544	que
7	2471	ils
8	2343	nous
9	1694	des
10	1692	pas
11	1320	pour
12	1320	leur
13	1233	ceux
14	1106	une
15	1054	lui
16	961	dans
17	960	seigneur
18	838	eux
19	781	sur
20	695	sont
21	645	ont
22	636	avons
23	617	leurs
24	602	fait
25	590	donc
26	571	dit
27	563	gens
28	562	son

FIGURE 1.1 – La loi de Zipf dans le Coran (Wikipédia)

Formellement, la tokénisation permet d'associer un entier naturel à chaque token (peu importe l'ordre), et on parlera de *vocabulaire* pour désigner l'ensemble des tokens. La taille du vocabulaire est un hyperparamètre crucial :

— Un vocabulaire trop petit risque de fragmenter excessivement les mots en sous-unités
— Un vocabulaire trop grand augmente la complexité du modèle et peut nuire à la généralisation

Il est également important de noter que la tokénisation inclut souvent des tokens spéciaux comme :

— `<PAD>` pour le padding des séquences à longueur fixe
— `<UNK>` pour les tokens hors vocabulaire
— `<BOS>` et `<EOS>` pour marquer le début et la fin des séquences

S'il est certain que le sujet des tokens présente un intérêt primordial, ce n'est pas vraiment (à l'instar du RLHF) le sujet central de ce stage.

1.2 Architecture des Transformers

Au cœur de ces modèles se trouve l'architecture Transformer, une structure complexe qui a révolutionné le traitement du langage naturel[2]. Dans cette étude, nous utilisons une version adaptée de cette architecture pour explorer les capacités des LLM dans les jeux de plateau. Examinons en détail les composantes clés d'un Transformer :

1.2.1 Plongement (Embedding)

La première étape consiste à transformer chaque token d'entrée en un vecteur de grande dimension. Ce processus, appelé plongement ou embedding, permet d'encoder la sémantique de chaque token dans un espace vectoriel. Soit w un token, son plongement e_w est défini par :

$$e_w = E[w] \tag{1.1}$$

où E est la matrice de plongement apprise durant l'entraînement. Cette matrice projette chaque token dans un espace $\mathbb{R}^n$, où n est généralement de l'ordre de plusieurs centaines.

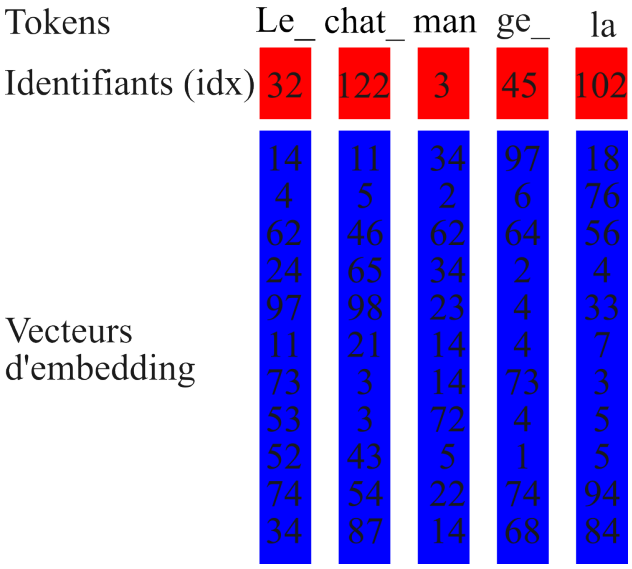


FIGURE 1.2 – Le mécanisme d'embedding

2. voir le notebook n°1 (PoèteGPT) pour un exemple de LLM simple qui fait des vers.

1.2.2 Mécanisme d'attention

À la suite du plongement, nous calculons l'attention, qui est le cœur du Transformer, qui permet de calculer les dépendances entre les différents tokens d'une séquence. Ce mécanisme utilise trois matrices apprises : Q (Query), K (Key), et V (Value). L'attention est calculée comme suit :

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1.2}$$

où d_k est la dimension de la matrice K. Cette formule capture l'importance relative de chaque token par rapport aux autres dans la séquence.

Dans notre implémentation, nous utilisons l'attention à plusieurs têtes (multi-head attention), qui permet au modèle de se concentrer sur différents aspects de la séquence simultanément :

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \tag{1.3}$$

où chaque tête est définie par :

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{1.4}$$

Les matrices W_i^Q, W_i^K, W_i^V, et W^O sont des paramètres appris.

Comme le fait de définir manuellement les poids va à l'encontre du but de l'apprentissage automatique, le modèle doit calculer lui-même les poids d'attention. Par analogie avec le langage des requêtes de base de données, nous faisons en sorte que le modèle construise un triplet de vecteurs : clé, requête et valeur. L'idée générale est que nous avons une "base de données" sous la forme d'une liste de paires clé-valeur. Le décodeur envoie une *requête* et obtient une réponse sous la forme d'une somme pondérée des *valeurs*, où le poids est proportionnel à la ressemblance entre la requête et chaque *clé*.

Le décodeur traite d'abord partiellement l'entrée "<start>" pour obtenir un vecteur intermédiaire h_0^{dec}, le 0-ème vecteur caché du décodeur. Ensuite, ce vecteur intermédiaire est transformé par une application linéaire en un vecteur de *requête* $q = W_q h_0^{dec}$. Parallèlement, les vecteurs cachés produits par l'encodeur sont transformés par une autre application linéaire en vecteurs de *clé* $k_i = W_k h_i^{enc}$. Ces applications linéaires sont utiles pour donner au modèle suffisamment de liberté pour trouver la meilleure façon de représenter les données.

Maintenant, la requête et les clés sont comparées en prenant leurs produits scalaires : $q \cdot k_0, q \cdot k_1, \ldots$. Idéalement, le modèle devrait avoir appris à calculer les clés et les valeurs de telle sorte que $q \cdot k_0$ soit grand, $q \cdot k_1$ soit petit, et que le reste soit très petit. Cela peut être interprété comme signifiant que le poids d'attention devrait être principalement appliqué au 0-ème vecteur caché de l'encodeur, un peu au 1er, et essentiellement pas aux autres.

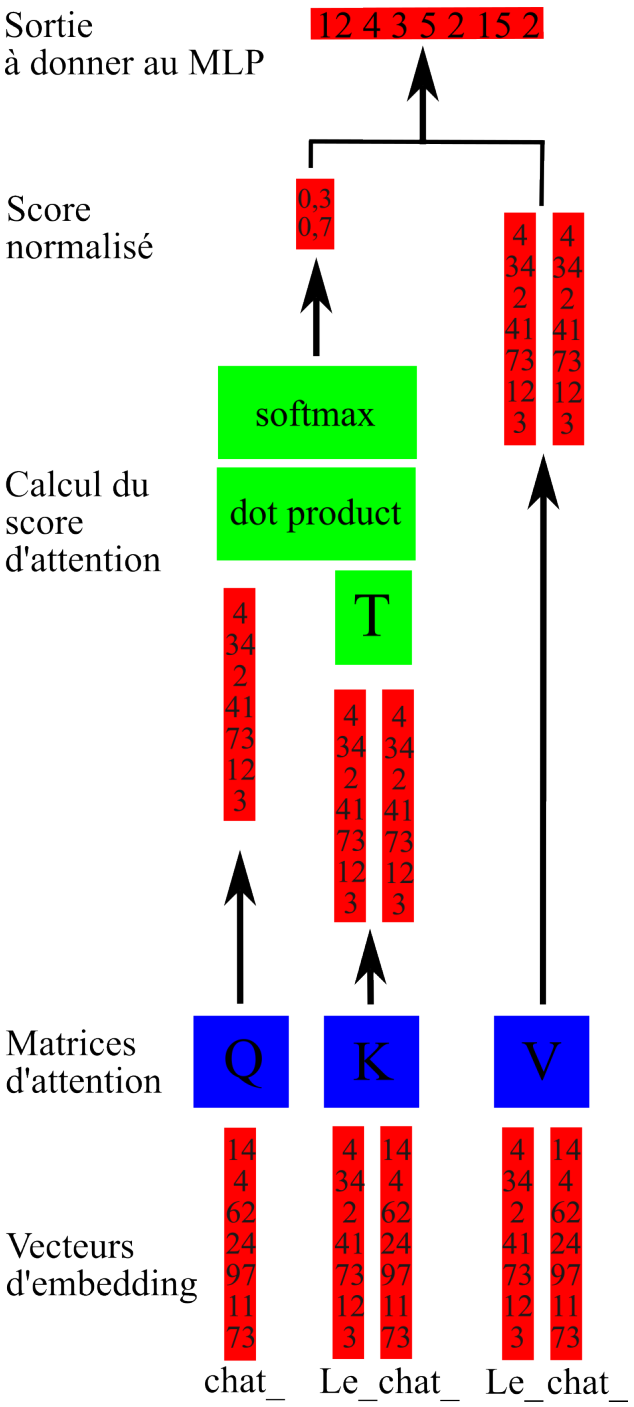


FIGURE 1.3 – Le mécanisme d'attention pour le vecteur de "chat" (permet d'incorporer le contexte dans le vecteur de "chat")

Pour faire une somme correctement pondérée, nous devons transformer cette liste de produits scalaires en une distribution de probabilité sur k_i. Cela peut être réalisé par la fonction softmax, nous donnant ainsi les poids d'attention :

$$a_i = \frac{\exp(q \cdot k_i)}{\sum_j \exp(q \cdot k_j)} \tag{1.5}$$

Ceci est ensuite utilisé pour calculer le *vecteur de contexte* :

$$c = \sum_i a_i v_i \tag{1.6}$$

où $v_i = W_v h_i^{enc}$ sont les vecteurs de *valeur*, transformés linéairement par une autre matrice pour donner au modèle la liberté de trouver la meilleure façon de représenter les valeurs. Sans les matrices W_k, W_v, le modèle serait forcé d'utiliser le même vecteur caché pour la clé et la valeur, ce qui pourrait ne pas être approprié, car ces deux tâches ne sont pas identiques.

C'est le mécanisme d'attention par produit scalaire. La version particulière décrite dans cette section est "l'attention croisée du décodeur", car le vecteur de contexte de sortie est utilisé par le décodeur, et les clés et valeurs d'entrée proviennent de l'encodeur, mais la requête provient du décodeur, d'où le terme "attention croisée".

Plus succinctement, nous pouvons l'écrire comme :

$$c = \text{softmax}(qK^T)V \tag{1.7}$$

où la matrice K est la matrice dont les lignes sont k_i. Notez que le vecteur de requête, q, n'est pas nécessairement le même que le vecteur clé-valeur k_i. En fait, il est théoriquement possible que les vecteurs de requête, de clé et de valeur soient tous différents, bien que cela soit rarement fait en pratique.

1.2.3 Perceptron multi-couche (Feed-Forward Network)

Après le calcul de l'attention, chaque token passe par un réseau de neurones entièrement connecté, appelé perceptron multi-couche ou feed-forward network (FFN). Ce réseau permet de manipuler les représentations abstraites issues de l'attention. Sa fonction peut être représentée comme :

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{1.8}$$

où W_1, W_2, b_1, et b_2 sont les paramètres appris. La fonction $\max(0, x)$ est la fonction d'activation ReLU.

1.2.4 Normalisation de couche (Layer Normalization)

Entre chaque sous-couche (attention et FFN), une normalisation de couche est appliquée pour stabiliser l'apprentissage :

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{1.9}$$

où μ et σ sont la moyenne et l'écart-type calculés sur la couche, γ et β sont des paramètres appris, ϵ est un petit terme pour éviter la division par zéro, et $\odot$ représente le produit de Hadamard (multiplication terme à terme des vecteurs).

1.2.5 Architecture globale

L'architecture complète d'un Transformer consiste en une pile de ces couches (attention multi-têtes, FFN, et normalisation), généralement répétée plusieurs fois. Dans notre étude, nous utilisons une version adaptée de cette architecture, avec un nombre réduit de couches et de têtes d'attention, approprié pour nos tâches sur les jeux de plateau.

1.3 Entraînement du modèle

L'entraînement du modèle est assez classique, il utilise une évolution de la descente de gradient que nous allons détailler.

1.3.1 La fonction d'erreur (loss function)

L'entraînement d'un modèle passe par la définition d'une *distance* (qui satisfait donc les axiomes des distances[3]). De la même manière, dans l'hypermarché du machine learning, nous avons un très grand choix. Citons :

- La distance Euclidienne, qu'on ne présente plus : $d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
- L'erreur MSE (Mean Square Error), que l'on exprime ainsi : $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$ qui est suffisante pour un très grand nombre de problèmes de machine learning
- L'erreur d'entropie croisée, qui nous est très utile en NLP, et que nous allons utiliser dans ce stage, que je détaille ci-dessous.

3. Soit d une distance, les propriétés suivantes sont vérifiées :
- $\forall x, y \quad d(x, y) = d(y, x)$ (symétrie)
- $\forall x \quad d(x, x) = 0$ (séparation)
- $\forall x, y \quad d(x, y) \geq 0$ (positivité)
- $\forall x, y, z \quad d(x, z) \leq d(x, y) + d(y, z)$ (inégalité triangulaire)

L'entropie croisée est particulièrement adaptée aux problèmes de classification. Pour un problème à C classes, elle s'exprime :

$$H(p, q) = -\sum_{i=1}^{C} p_i \log(q_i)$$

où p représente la distribution réelle (généralement un vecteur one-hot) et q la distribution prédite par le modèle.

Un vecteur one-hot est un vecteur binaire où seul l'indice correspondant à la classe correcte vaut 1, les autres valant 0. Par exemple, pour un problème à 4 classes, si la classe correcte est la deuxième, le vecteur one-hot sera : $[0, 1, 0, 0]$. Cette représentation a un avantage majeur : comme un seul terme du vecteur p est non nul (égal à 1), la formule de l'entropie croisée se simplifie considérablement :

$$H(p, q) = -\log(q_c)$$

où c est l'indice de la classe correcte. Cette simplification permet un calcul plus efficace de la fonction de perte lors de l'entraînement.

1.3.2 La descente du gradient

La descente du gradient est une méthode itérative d'optimisation qui permet de minimiser la fonction de perte. Son équation de mise à jour est :

$$\theta_{n+1} = \theta_n - \alpha \nabla L(\theta_n) \tag{1.10}$$

où les θ sont les paramètres du modèle, L la fonction de perte (voir section précédente) et α le taux d'apprentissage.

1.4 RLHF

Pour parler du RLHF, je vais faire un petit détour par une célèbre expérience de pensée du philosophe Nick Bostrom (professeur à Oxford). Imaginez donner comme fonction de loss à votre modèle l'écart avec la maximisation du nombre de trombones. A première vue, on peut se dire que le modèle se comportera en bon capitaliste, fixera les prix de manière adéquate, gerera les usines de trombones, etc.

Mais à y regarder de plus près, cela engendre ce que l'on appelle en philosophie de l'IA un *problème d'alignement*. En effet, qui nous dit que pour résoudre la tâche qui lui incombe, le modèle ne va pas passer par des moyens détournés (comme détruire des ressources terrestres, manipuler les marchés) pour maximiser la production de trombones ? L'experience de pensée de Nick Bostrom imagine le pire (*worst case*).

Pour parer ces possibilités, le monde du deep learning s'est doté d'un outil formidable : l'apprentissage par renforcement assisté par l'humain. Dans le contexte des LLMs, cela s'appelle RLHF (Reinforcment Learning by Human Feedback). Cela consiste à considérer le modèle comme un agent évoluant dans un environnement (ici une conversation), et à lui indiquer s'il fait bien de dire cela ou pas (carotte et bâton).

Plus formellement, le RLHF cherche à maximiser la récompense attendue, qui peut s'écrire :

$$\pi^* = \text{argmax}_{\pi \in \Pi} \mathbb{E}_{(x,y)\sim\mathcal{D}}[R(x, \pi(x))] \tag{1.11}$$

où :
— π^* est la politique optimale que nous cherchons à apprendre
— Π est l'ensemble des politiques possibles
— x représente l'entrée (le contexte de la conversation)
— y est la sortie générée
— $\mathcal{D}$ est la distribution des données
— $R(x, y)$ est la fonction de récompense basée sur le feedback humain

Cette récompense R est généralement apprise par un modèle auxiliaire (reward model) entraîné sur des comparaisons de réponses faites par des humains. Le modèle est ensuite optimisé pour maximiser cette récompense via des algorithmes comme PPO (Proximal Policy Optimization).

Mais pour entraîner mes modèles d'échecs, il n'y a pas besoin d'utiliser le RLHF, donc ce n'est pas —à l'instar de la tokénisation— l'objectif de ce stage.

1.5 Objectifs du stage

Ce stage vise à explorer comment les grands modèles de langage, basés sur l'architecture Transformer décrite ci-dessus, parviennent à jouer aux jeux de plateau. Nous nous concentrons sur deux jeux en particulier : les échecs et le tic-tac-toe (morpion). Nos objectifs principaux sont :

1. Étudier les capacités des LLM dans les jeux de plateau, en particulier leur aptitude à développer des stratégies et à résoudre des puzzles.
2. Analyser l'impact du bruit dans les données d'entraînement sur les performances et les capacités émergentes des modèles.
3. Explorer les représentations internes des modèles pour comprendre comment ils encodent l'information et développent une compréhension des règles et des stratégies des jeux.
4. Examiner le phénomène de "grokking", où le modèle semble soudainement comprendre le jeu après une longue période d'apprentissage apparent sans amélioration significative.

5. Tester l'hypothèse de représentation platonique (PRH) dans le contexte des jeux de plateau, en comparant les représentations issues de différentes modalités d'entrée (texte et image).

À travers ces objectifs, nous espérons non seulement mieux comprendre les mécanismes d'apprentissage des LLM dans le contexte spécifique des jeux de plateau, mais aussi contribuer à la compréhension plus large de comment ces modèles développent des capacités complexes à partir de données d'entraînement.

Dans les chapitres suivants, nous détaillerons notre approche, nos expériences et nos résultats pour chacun de ces objectifs, en commençant par notre étude sur le jeu d'échecs, suivie de notre analyse approfondie du jeu de tic-tac-toe.

Chapitre 2

Première partie du stage (mai-juillet) : le jeu d'échecs, étude de l'impact du bruit dans les données d'entraînement

2.1 Introduction

Cette première partie du stage, qui court de mai à début juillet (présentation lors de la PFIA 2024 à La Rochelle), s'attache à étudier les LLMs entraînés sur des parties d'échecs au format PGN, comme cela a été fait début 2024 par Adam Karvonen [10]

L'exploration des échecs dans le domaine de l'intelligence artificielle offre un terrain particulièrement fertile pour sonder les capacités des Grands Modèles de Langage (LLMs) au-delà des stratégies de jeu conventionnelles. Contrairement aux moteurs d'échecs traditionnels qui se concentrent uniquement sur l'identification du coup optimal, les LLMs peuvent utiliser une multitude de données historiques de parties en notation `PGN` pour élaborer des stratégies nuancées qui reflètent le jeu humain.

Ici, nous utilisons la notation `PGN` non seulement comme un enregistrement des parties passées, mais aussi comme un outil dynamique pour guider les sorties stratégiques des LLMs. En manipulant les en-têtes `PGN` - tels que les classements Elo, les résultats des parties et les détails des joueurs - nous pouvons orienter le comportement du modèle dans la résolution de problèmes d'échecs ou l'émulation de styles stratégiques spécifiques, comme l'illustre l'exemple d'un historique de partie d'échecs au format `PGN` entre Garry Kasparov et Magnus Carlsen (voir figure 2.1). Cette manipulation démontre la flexibilité des LLMs à adopter des stratégies diverses et proches de celles des humains, permettant une analyse approfondie des comportements émergents.

Les propriétés émergentes dans ce contexte résultent des interactions collectives

```
[White "Garry Kasparov"]
[Black "Magnus Carlsen"]
[Result "1/2-1/2"]
[WhiteElo "2900"]
[BlackElo "2800"]
1. e4 e5
2. Nf3 ...
```

FIGURE 2.1 – Partie d'échecs entre Garry Kasparov et Magnus Carlsen

entre les composantes du modèle, qui transcendent les capacités de toute partie individuelle. Nous passerons en revue diverses définitions d'une propriété émergente et étudierons la situation aux échecs. La capacité à ajuster les stratégies en fonction de subtils changements dans le contexte du jeu illustre davantage un niveau de robustesse comparable à celui des joueurs humains, qui adaptent habilement leurs stratégies dans de nouvelles situations. Nous étudions la propriété de résolution de problèmes lorsque nous modifions le paramètre **elo**.

Cet article examine l'impact des ajustements des en-têtes `PGN` sur les processus de prise de décision des grands modèles de langage, influençant leurs styles de jeu en conséquence. Il approfondit le concept de propriétés émergentes, soulignant comment les LLMs développent des capacités qui n'ont pas été explicitement enseignées mais émergent de leur environnement d'entraînement et de leurs entrées. Une attention particulière est accordée à la tâche de résolution de problèmes, définie comme la capacité à identifier le coup correct dans une position donnée avec une forte probabilité.

Les sections suivantes de cet article exploreront ces propriétés émergentes en détail, en utilisant des exemples basés sur les échecs pour démontrer comment de subtiles manipulations dans les données d'entraînement peuvent modifier significativement le comportement des LLMs. Cette exploration non seulement améliore notre compréhension des LLMs dans le domaine des échecs, mais éclaire également leur potentiel plus large dans les applications stratégiques, marquant un changement significatif de la poursuite de solutions optimales vers l'engagement avec des problèmes complexes d'une manière plus proche de celle des humains.

La section 2 explore la nature des propriétés émergentes, la section 3 introduit des techniques de sondage pour les LLMs, et la section 4 décrit les méthodes pour quantifier la capacité à résoudre des problèmes et détecter des propriétés émergentes par rapport au paramètre **elo**.

2.2 Qu'est-ce qu'une propriété émergente ?

Les propriétés émergentes résultent des interactions collectives des composantes d'un système, au-delà des propriétés des parties individuelles. Les *transitions de phase* dans les structures aléatoires avec un paramètre p illustrent ce concept. Considérons les graphes aléatoires suivant le modèle d'Erdös-Renyi [7], où p est la probabilité de sélectionner indépendamment l'une des $\binom{n}{2}$ arêtes symétriques simples. Avec une forte probabilité, si $p < \log n/n$, le graphe n'est pas connecté, tandis que si $p \geq \log n/n$, le graphe est connecté. La valeur $p = \log n/n$ est la transition de phase pour la propriété de connectivité. En physique, le paramètre p est souvent lié à la température, où des transitions de phase similaires existent.

Dans le domaine des grands modèles de langage, les propriétés émergentes peuvent être vues comme la manifestation de compétences qui dépassent le contenu explicite des données d'entraînement. Arora discute de la façon dont ces compétences se développent à partir des potentiels latents intégrés dans l'entraînement [2]. Est-il possible que les données d'entraînement combinent des groupes allant jusqu'à 3 compétences différentes, et que les réponses du LLM témoignent de beaucoup plus de compétences, jusqu'à 6 compétences différentes par exemple pour GPT-4. L'approche d'Arora, basée sur des considérations combinatoires, montre que c'est possible. Les réponses du LLM témoignent de compétences qui n'apparaissaient pas dans les données d'entraînement, et sont donc émergentes.

Cela est davantage exploré dans le contexte du traitement du langage naturel et des jeux stratégiques. Dans le cas du langage naturel, on peut vouloir apprendre la structure syntaxique d'une phrase à partir des vecteurs `Word2Vec` des mots de la phrase [8]. Dans les jeux [13, 4], le langage est la séquence des coups (par exemple b2, f4, ... dans la notation `PGN` pour les échecs ou l'Othello). Les données d'entraînement sont tirées de la base de données Lichess [14] qui stocke l'historique des parties, et le LLM apprend la fonction `next` qui fournit le prochain coup dans une séquence. Le réseau de neurones décrivant la fonction `next` peut-il également prédire la position de chaque pièce du jeu, *c'est-à-dire* le modèle du monde ? Si c'est possible, les auteurs disent que la position du jeu est une propriété émergente, car aucune règle du jeu n'a jamais été donnée. Nicholas Carlini utilise le terme « Modélisation du langage, pas de victoire » pour décrire comment les LLMs peuvent également apprendre à ne pas gagner, mais à jouer comme des humains [4].

Ils obtiennent ces résultats en considérant des sondes, que nous allons maintenant décrire.

2.3 Sondage pour détecter les propriétés émergentes

Les techniques de sondage [3] sont utilisées pour découvrir des structures latentes dans les réseaux de neurones profonds, basées sur certains états internes du réseau. Supposons une entrée x de dimension d dans un réseau qui calcule $f(x)$, et soit x^i le vecteur des valeurs à la couche i du réseau. Le sondage consiste à lire certains des x^i et à apprendre un nouveau réseau comme classificateur pour une nouvelle propriété (une fonction g), à partir des mêmes données d'entraînement. Nous ne calculons pas par rapport à f, mais par rapport à certains états de l'algorithme qui calcule f.

Dans le traitement du langage naturel, [8] considère une phrase comme une entrée x qui est une séquence de n mots donnée à un réseau de neurones qui calcule une fonction f. En interne, il calcule la représentation `Word2Vec` des mots w_i comme des vecteurs v_i de faible dimension. Nous pouvons apprendre une matrice B qui transforme les vecteurs en $v'_i = B.v_i$ de sorte que la nouvelle norme $(B; v)^t.(B.v)$ définit une nouvelle distance dist_B. La fonction g est un classificateur pour la propriété : dist_B est proche de la distance associée à un arbre syntaxique de la phrase originale. Ils sondent le réseau original, et la nouvelle propriété g est émergente.

Dans les jeux, une question naturelle est de demander si nous pouvons approximer la fonction $\mathsf{Position}(i, j) = k$ qui décrit la pièce sur le plateau en position (i, j) à partir d'un sous-ensemble x^i des couches internes du réseau. Dans le jeu d'Othello [13], $k \in \{\perp, \text{noir}, \text{blanc}\}$ tandis qu'aux échecs [4, 10], la valeur k décrit également le type de chaque pièce comme dans la section 2.4.1. Si la fonction $\mathsf{Position}(i, j) = k$ peut être approximée par des sondes sur un sous-ensemble des couches du réseau de neurones définissant la fonction next, nous disons que le *modèle du monde* est une propriété émergente.

2.3.1 Intervention sur les sondes

Si nous approximons la fonction $\mathsf{Position}$, nous pouvons essayer de modifier l'échiquier, en retirant une pièce aux échecs [10] ou en échangeant une pièce noire contre une pièce blanche à l'Othello [13]. Ces modifications sont appelées *interventions sur l'échiquier*. Nous pouvons alors inférer un $(x')^i$ par descente de gradient sur le réseau définissant la sonde et calculer une nouvelle valeur pour la fonction $f(x)$, où les valeurs $(x')^i$ remplacent les valeurs x^i. Nous pouvons ensuite vérifier si la prédiction next est cohérente avec l'intervention. Typiquement, les cartes de chaleur [13] donnent les coups les plus probables et devraient s'adapter à l'intervention. Les interventions peuvent également concerner les niveaux de compétence (le grade **elo** aux échecs) dans [10].

2.4 Compétences de résolution de problèmes

2.4.1 Définitions pour les compétences de résolution de problèmes

Nous nous penchons sur la prouesse de résolution de problèmes des Grands Modèles de Langage tels que Chess-GPT, évaluant leur capacité à utiliser et adapter des stratégies apprises à des défis nouveaux et non structurés. Cette aptitude souligne leurs capacités cognitives émergentes et sert de métrique pour évaluer la profondeur et la flexibilité de leurs comportements appris. Familiarisons-nous avec les termes suivants :

— `PGN` **(Portable Game Notation)** : Une structure de données qui inclut un *en-tête* détaillant les joueurs, leurs classements, le résultat de la partie [1-0], [0-1], ou [1/2-1/2], suivi d'une séquence de coups (voir figure 2.1).

— Le **classement Elo** est un nombre attribué à un joueur ou à un problème indiquant son niveau de difficulté. Pour une définition précise, nous renvoyons à Wikipédia. Cette définition est implémentée sur Lichess.com [14].

L'attrait des problèmes d'échecs réside dans leurs solutions uniques : ces problèmes sont des positions extraites de véritables parties d'échecs, chacune exigeant un coup correct spécifique.

Definition 1. ***M*** *est l'ensemble des coups possibles :*

$$\boldsymbol{M} = \{(p, i, j) : p \in \textit{Pièces d'échecs}, i \in <a; h>, j \in <1; 8>\}$$

où l'ensemble des pièces d'échecs est $\{\emptyset, T, C, F, D, R\}$.

Definition 2. $\texttt{puzzle\_PGN}_e$ *est l'ensemble des paires* (x, y) *telles que* $x \in \texttt{PGN}$ *et* $y \in \boldsymbol{M}$ *est l'unique coup correct à* x*,* e *est le niveau* ***elo*** *du problème tel que donné dans la base de données Lichess [14].*

$$\texttt{puzzles\_PGN}_e = \{(x, y) : x \in \texttt{PGN}, y \in \boldsymbol{M}, e \in \textbf{\textit{elo}}\}$$

où ***elo*** *est l'intervalle* $<1000; 3000>$.

Nous fixons un niveau **elo** e et sélectionnons $(x, y) \in \texttt{puzzles\_PGN}_e$ de la base de données Lichess avec une distribution uniforme.

Definition 3. *Un LLM possède la compétence de résolution de problèmes pour un niveau* ***elo*** e *et à un taux* α *si*

$$\mathsf{Prob}_x[\mathsf{next}(x) = y \mid (x, y) \in \texttt{puzzles\_PGN}_e] \geq \alpha.$$

Cette méthodologie nous permet d'évaluer précisément une compétence spécifique : la capacité à identifier (avec une grande probabilité) le meilleur coup étant donné une

position. Cela implique de fournir au LLM un historique de partie PGN (voir figure 2.1) et de l'interroger sur le prochain jeton (dans ce cas, le prochain coup).

Notre focus ici est le taux de compétence pour une classe particulière de problèmes et un niveau **elo** fixé. Dans ce contexte, une intervention est la modification du paramètre **elo**. Nous étudions ensuite comment le taux de compétence augmente si le nombre **elo** augmente et diminue si le nombre **elo** diminue. C'est la propriété émergente que nous essayons de tester.

Karvonen [10] a démontré que lorsque le LLM est "amorcé" (dans le prompt) avec un score [0-1] et qu'on le fait jouer en tant que blanc, sa performance se détériore par rapport à être amorcé avec [1-0].

2.4.2 Résultats : l'impact du bruit sur l'émergence de ces capacités

Lors de mes expériences, j'ai entraîné quatre LLMs différents sur différents niveaux de bruit (0%, 5%, 50% et 80%), et je les ai testés sur 3 propriétés différentes :

— Le sondage (*probing*) à la Karvonen (cf. prochaine section)
— La capacité à effectuer un coup légal
— La capacité de résolution de problèmes d'échecs

Une sonde au sens de Karvonen est un classifieur linéaire entraîné sur les activations des différentes couches du réseau de neurones de ChessGPT :

$$P_{i,l} = \text{Softmax}(W_{i,l} \cdot A_l) \tag{2.1}$$

$$g(x) = f(w^T x + w_0) = f\left(\sum_{j=1}^{n} w_j x_j + w_0\right) \tag{2.2}$$

où :

— $P_{i,l}$: Distribution de probabilité pour la case i prédite par la sonde à la couche l
— $W_{i,l}$: Matrice de poids de dimension 512×13
— A_l : Vecteur d'activation de dimension 512
— f : Fonction d'activation (Softmax)

L'expérience vise à évaluer l'impact du taux de bruit dans les données d'entraînement sur les capacités de respect des règles du jeu et de résolution de problème, toutes choses égales par ailleurs (*ceteris paribus*). Les résultats montrent que les LLMs entraînés avec plus de bruit réussissent mieux la tâche "effectuer un coup légal", tandis que les LLMs entraînés sur des parties déterministes (générées par Stockfish) produisent davantage de coups illégaux, et ce plus rapidement. La pente plus abrupte pour ces derniers modèles indique une moindre *robustesse* comparée aux modèles entraînés sur des parties bruitées. Deux explications peuvent être avancées :

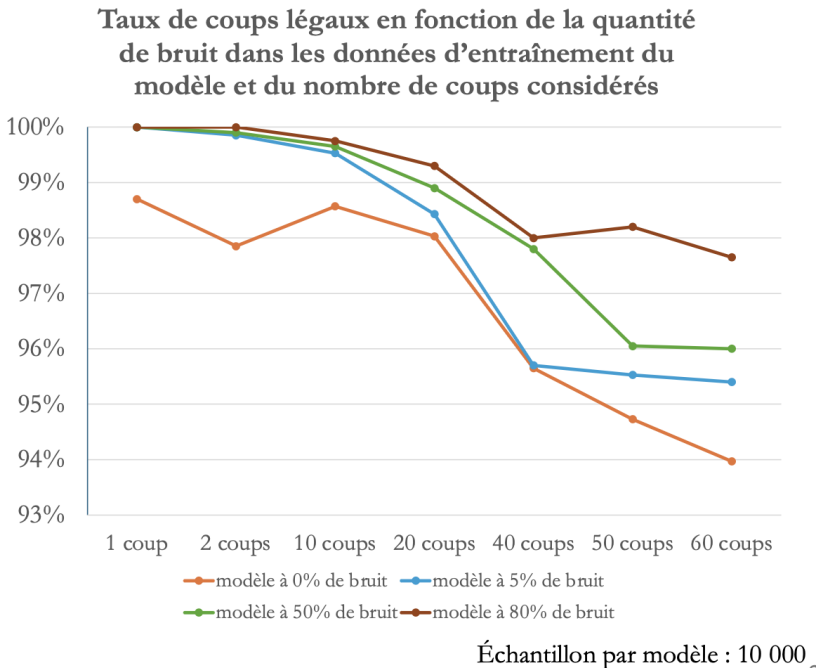


FIGURE 2.2 – Impact du bruit sur le taux de coups légaux

— Les LLMs entraînés sur du bruit se concentrent uniquement sur la tâche "effectuer un coup légal", alors que les LLMs déterministes doivent simultanément apprendre à "effectuer un coup légal qui soit aussi un bon coup"
— La seconde tâche présente une complexité d'apprentissage supérieure

Le bruit apparaît donc comme un allié dans la génération de parties d'échecs valides. La seconde expérience utilise la base de données de problèmes d'échecs de Lichess [14], disponible en accès libre, pour évaluer nos quatre LLMs : Ce graphique révèle, à l'inverse, que les modèles avec le moins de bruit (0% et 5%) excellent dans la résolution de problèmes d'échecs. Le modèle à 5% obtient même des résultats légèrement supérieurs, suggérant qu'une faible injection de bruit peut améliorer les performances. Ces expériences permettent de conclure qu'il est bénéfique d'introduire un faible pourcentage de bruit pour augmenter à la fois la robustesse et les performances des LLMs entraînés sur des tâches semi-synthétiques comme les échecs. Cette conclusion pourrait avoir des implications importantes pour l'entraînement d'autres modèles de langage sur des tâches similaires.

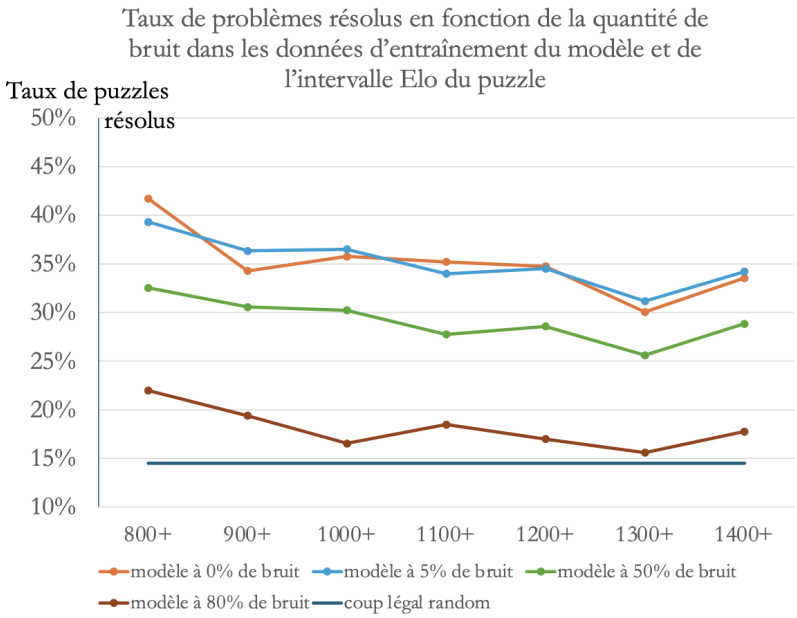


FIGURE 2.3 – Impact du bruit sur le taux de résolution de puzzles

2.4.3 Grokking

Le grokking est un phénomène étonnant qui consiste à poursuivre l'apprentissage bien au-delà de l'overfitting. Le terme "grokking", emprunté à la science-fiction[1], décrit une transition soudaine de la mémorisation pure vers une véritable compréhension du problème. Pour un exemple de *grokking* sur un problème simple d'arithmétique, vous pouvez vous reporter au notebook n°2.

Sur ce graphique, on observe trois phases distinctes :

1. Une phase d'apprentissage rapide où la précision du modèle atteint 100% sur les données d'entraînement
2. Une longue phase d'apparente stagnation où la performance sur les données de test reste proche de 0% (overfitting)
3. Une phase de "grokking" où la performance sur les données de test s'améliore soudainement, indiquant une véritable compréhension du problème

Deux éléments clés sont nécessaires pour observer ce phénomène :

1. Une régularisation appropriée. La présence d'une régularisation, typiquement sous forme de weight decay ou de régularisation L2, est cruciale. Cette régularisation

1. Le terme vient du roman *Stranger in a Strange Land* (1961) de Robert A. Heinlein, où il signifie "comprendre profondément et intuitivement".

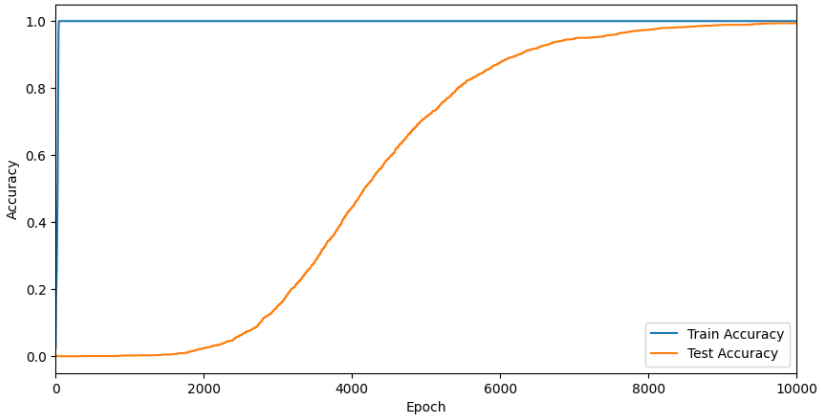


FIGURE 2.4 – Le phénomène de *grokking* : évolution des performances sur les ensembles d'entraînement et de test

empêche le modèle de se contenter de solutions "faciles" basées sur la pure mémorisation. Power et al. [15] ont montré que sans régularisation, le grokking n'apparaît pas : le modèle reste bloqué dans un état de mémorisation pure.

2. L'équilibre mémorisation-généralisation. Le phénomène de grokking met en lumière la distinction fondamentale entre mémorisation et généralisation. Pendant la phase initiale, le modèle mémorise effectivement les données d'entraînement, comme en témoigne sa performance parfaite sur cet ensemble. Cependant, sous l'effet de la régularisation et d'un entraînement prolongé, le modèle finit par découvrir une solution plus "simple" ou plus "élégante" qui capture la structure sous-jacente du problème [20].

3. Une interprétation spectrale. Pearce et al. [17] proposent une analyse via la transformée de Fourier discrète (DFT). Cette décomposition s'écrit mathématiquement :

$$f(x) = \sum_{k=0}^{\infty} a_k e^{2\pi i k x}$$

où :
- $f(x)$ est la fonction représentant les poids du réseau
- k est la fréquence (plus k est grand, plus la fonction oscillera rapidement)
- a_k sont les coefficients de Fourier (leur amplitude indique l'importance de chaque fréquence)
- $e^{2\pi i k x}$ représente les oscillations de base (combinaisons de sinus et cosinus)

Durant la phase de mémorisation, les coefficients a_k des hautes fréquences dominent, indiquant une solution complexe. La phase de grokking voit l'émergence progressive

des coefficients de basse fréquence (k petit), correspondant à une solution plus simple et plus générale.

Chapitre 3

Seconde partie du stage (juillet-septembre) : le tic-tac-toe, étude des représentations du modèle

Introduction

Une hypothèse en particulier a attiré notre attention, parce qu'elle semble faire la synthèse de nombreuses observations éparses que l'on a pu faire dans le domaine de l'interprétabilité (ou explicabilité). Il s'agit de la PRH, ou *Platonic Representation Hypothesis*, qui postule que les modèles de *transformer* se forgent un modèle de représentation du monde durant l'entraînement, résultant de la recherche de simplicité que recherche naturellement un réseau de neurones qui apprend.

La conséquence la plus surprenante de leur hypothèse est que des modèles entraînés sur différents types de données (images, texte, sons, etc.) vont converger vers la même représentation dans leur *espace latent*. Notre but ici sera d'étudier en profondeur un cas simple, très simple, celui du Morpion. Dans ce jeu, dont l'ensemble des parties peut être généré en quelques minutes par un ordinateur moderne, le monde est très simple, et ne nécessite pas une immense puissance de calcul, autre avantage dont nous pourrons tirer parti.

Ici nous allons étudier les représentations de deux *transformers* : l'un est entraîné sur des images du jeu, et l'autre sur des séquences de coups sous forme de texte. L'hypothèse

Figure 3.1 – Représentations textuelles et imagées du Morpion

principale que nous voulons tester est en somme si les deux représentations d'une même partie de Morpion convergent.

Pour préparer le terrain et explorer d'autres aspects de l'interprétation de l'architecture *transformer*, nous allons effectuer des tests à l'aide d'autres techniques, à commencer par le *sondage* (*probing*).

3.1 Comment représenter une partie de Morpion ?

Le morpion, comme les échecs ou les dames, est un jeu de plateau. Mais contrairement aux deux derniers, il est extrêmement simple, ce qui permet la complétude de l'analyse (on génère toutes les parties possibles, et on analyse notre *transformer* avec toutes ces parties).

Une manière très simple, voire enfantine, de représenter une partie de morpion est de la noter sous forme de dessin, côte à côte, en montrant l'état du plateau à chaque trait. C'est le format que nous avons choisi pour les images, qui ont donc une taille 9x9.

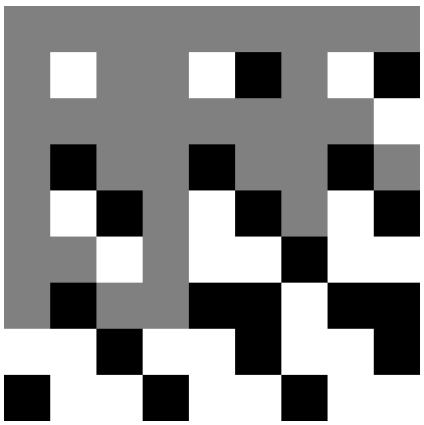


FIGURE 3.2 – Représentation d'une partie de Morpion

Pour noter les parties de Morpion textuellement, nous avons choisi de noter les coups les uns à la suite des autres, comme le format PGN aux échecs, ce qui donne un historique de la partie.

Le format du jeu de Morpion étant ce qu'il est, le nombre de coups ne peut excéder 9, ce qui est bien pratique pour la tokénisation, première étape de l'apprentissage.

Voici les tokénisations respectives des images et des textes (que l'on trouve dans les fichiers `meta.pkl`) :

Listing 3.1 – Tokénisation des images

```
{
"vocab_size": 4,
```

```
"itos": {"0": "b", "1": "n", "2": "g", "3": ";"},
"stoi": {"b": 0, "n": 1, "g": 2, ";": 3}
}
```

Listing 3.2 – Tokénisation des textes

```
{
"vocab_size": 10,
"itos": {"0": ";", "1": " ", "2": "0", "3": "1", "4":
    "2", "5": "3", "6": "X", "7": "O", "8": "/", "9":
    "-"},
"stoi": {";": 0, " ": 1, "0": 2, "1": 3, "2": 4, "3": 5,
    "X": 6, "O": 7, "/": 8, "-": 9}
}
```

Les *stoi* et les *itos* ne sont pas grecs, mais un format de données qui permet de convertir les chaînes de caractères (strings) en entiers naturels (integers), c'est-à-dire string to integer ou plus simplement *stoi*, l'*itos* étant l'opération réciproque.

La tokénisation des images repose sur le pixel, qui sera blanc, valeur 0 (représenté par 'b'), si la case est saturée par un X, noire, valeur 1 (représenté par 'n'), si la case est saturée par un O, et grise, valeur 2 (représenté par 'g') si la case est vide. L'image est parcourue de manière naturelle, ligne par ligne pour former une ligne au format `str` (string).

Dans les deux cas, "; " est le token de début, celui qui marque le début de la séquence pour les images comme pour les textes.

3.2 Architecture du transformer pour le projet Morpion

Dans ce projet, nous utilisons un transformer relativement petit, adapté à la tâche simple du Morpion. Voici une description détaillée de son architecture :

3.2.1 Vue d'ensemble

— Nombre de couches (n_layer) : 2
— Nombre de têtes d'attention (n_head) : 1
— Dimension du modèle (n_embd) : 256
— Taille du bloc (block_size) : 36 tokens

3.2.2 Couches détaillées

Couche d'embedding

— Dimension : 256 (n_embd)
— Vocabulaire : 10 tokens (d'après la tokénisation fournie)

Couches transformer (répétées 2 fois)

Pour chaque couche transformer :

1. Attention
— Type : Self-attention à une seule tête (n_head = 1)
— Dimension de la tête : 256 (n_embd)
— Composants :
  — Projections linéaires pour Q, K, V ($3 \times 256 \times 256$)
  — Softmax et dropout (taux = 0.0)
  — Projection de sortie (256×256)

2. Normalisation de couche (Layer Norm) Appliquée avant l'attention (architecture "Pre-LN")

3. MLP (Perceptron multicouche)
— Deux couches linéaires avec activation GELU
— Dimensions : $256 \rightarrow 1024$ ($4 \times$ n_embd) $\rightarrow 256$
— Dropout (taux = 0.0) après la deuxième couche linéaire

4. Normalisation de couche (Layer Norm) Appliquée avant le MLP

Couche de sortie

Projection linéaire : $256 \rightarrow$ taille du vocabulaire (10)

3.2.3 Hyperparamètres d'entraînement

— Taille de batch : 2048
— Taux d'apprentissage initial : 3e-4
— Décroissance du taux d'apprentissage : jusqu'à 3e-5
— Nombre maximal d'itérations : 10000
— Warmup : 2000 itérations

Cette architecture représente un transformer relativement petit, adapté à la tâche simple du Morpion. Quelques observations supplémentaires :

1. La taille réduite (2 couches, 1 tête d'attention) est appropriée pour cette tâche qui ne nécessite pas une grande complexité.
2. L'absence de dropout (0.0) suggère que le modèle ne risque pas de surapprentissage, probablement en raison de la simplicité de la tâche et de la petite taille du modèle.
3. La dimension d'embedding de 256 offre une capacité de représentation suffisante pour les états du jeu de Morpion.
4. Le block_size de 36 tokens est amplement suffisant pour représenter une partie complète de Morpion (qui ne peut pas dépasser 9 coups).
5. L'utilisation d'une seule tête d'attention par couche est intéressante et pourrait permettre une analyse plus facile des mécanismes d'attention.

3.3 Probing

Le *probing* (ou sondage) est une technique permettant de détecter la présence de propriétés dans un réseau de neurones. C'est une technique qui a été abondamment utilisée depuis 2018 et l'article Manning et al. qui l'applique à la détection d'arbres syntaxiques dans un transformer entraîné sur un corpus de textes étiquetés.

Ici, nous voulons utiliser le *probing* pour détecter la présence ou non d'une représentation du plateau de morpion, couche par couche dans le *transformer*. L'idée est d'entraîner un classifieur linéaire case par case et couche par couche pour qu'il détecte l'état d'une case (X, O ou vide). La technique utilisée ici est une généralisation du SVM originel, appelée *OneVsRest*.

3.3.1 Qu'est-ce qu'un SVM linéaire ?

Le SVM (Support Vector Machine) linéaire est une technique d'apprentissage supervisé utilisée pour la classification. Dans le cas binaire, l'objectif est de trouver un hyperplan qui sépare au mieux les deux classes de données. Dans le cas linéaire qui nous occupe, il s'agit donc de trouver l'hyperplan qui explique le mieux la séparation des données dans le réseau de neurones.

Soit un ensemble de données d'entraînement $\{(x_i, y_i)\}_{i=1}^{n}$, où $x_i \in \mathbb{R}^d$ sont les vecteurs de caractéristiques et $y_i \in \{-1, +1\}$ sont les étiquettes de classe.

L'hyperplan séparateur est défini par l'équation :

$$w^T x + b = 0$$

où $w \in \mathbb{R}^d$ est le vecteur normal à l'hyperplan et $b \in \mathbb{R}$ est le biais.

Problème d'optimisation

Le SVM cherche à maximiser la marge entre l'hyperplan et les points les plus proches de chaque classe. Cela se traduit par le problème d'optimisation suivant :

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|w\|^2 \\ \text{s.c.} \quad & y_i(w^T x_i + b) \geq 1, \quad i = 1, \ldots, n \end{aligned}$$

En utilisant les multiplicateurs de Lagrange, on obtient la formulation duale :

$$\begin{aligned} \max_{\alpha} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i^T x_j \\ \text{s.c.} \quad & \sum_{i=1}^{n} \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, n \end{aligned}$$

où α_i sont les multiplicateurs de Lagrange et C est un paramètre de régularisation.

Une fois les α_i optimaux trouvés, la fonction de décision pour un nouveau point x est :

$$f(x) = \text{sign}\left(\sum_{i=1}^{n} \alpha_i y_i x_i^T x + b\right)$$

où b est calculé en utilisant les vecteurs de support (points pour lesquels $\alpha_i > 0$).

Dans notre code, une approche *One-vs-Rest* est utilisée pour étendre le SVM binaire au cas multi-classe. Pour K classes, on entraîne K classifieurs binaires, chacun séparant une classe de toutes les autres.

3.3.2 Les résultats couche par couche

Le transformer est une architecture assez complexe qui permet de multiples points de sonde. Nous avons donc voulu sonder les moindres recoins de celui-ci pour avoir une idée de la représentation du plateau de morpion à chaque étape.

Voici les résultats en prenant la précision de l'évaluation du SVM couche par couche pour le texte.

Nous pouvons observer que le transformer acquiert petit à petit une représentation de l'état du plateau de morpion, qui culmine lors de la normalisation post-attention et de la seconde couche du MLP.

Pour le modèle entraîné sur les images, le résultat est très différent, puisque la courbe reste quasiment plate tout du long du *transformer*.

On remarque que le modèle ne forge pas de représentation du plateau de Morpion, ce qui peut interroger. Pour répondre à cela, nous pouvons formuler l'hypothèse

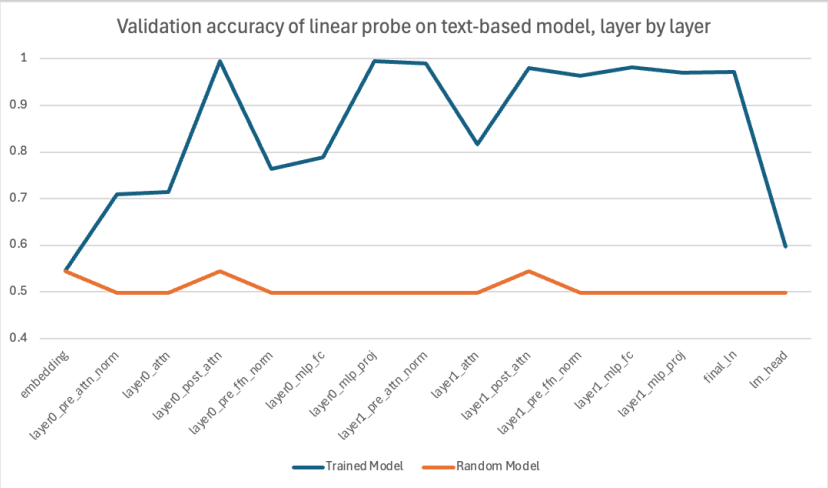


FIGURE 3.3 – Probing couche par couche pour le texte

qu'étant donné que pour résoudre la tâche (prédire le prochain token), il faut avoir une représentation sur laquelle se reposer à au moins un moment, il n'y a pas besoin de se forger une représentation lorsqu'on a déjà l'image en entrée, alors que c'est nécessaire lorsqu'on n'a que le texte.

Nous pouvons également nous intéresser à la représentation des cases (*positions*), qui montre des effets de symétrie et d'asymétrie assez intéressants.

Une observation générale est que très souvent, dans la plupart des couches des deux modèles, c'est la case du milieu (position 5) qui est la mieux représentée.

Ensuite, nous pouvons observer de belles symétries, surtout pour le texte, et certaines couches, qui montrent une représentation géométrique du plateau, qui évolue en fonction de la couche. Par exemple, dans le modèle d'image, la couche de normalisation pré-attention avantage clairement la case centrale, tandis que plus on s'approche de la sortie, moins cette case du milieu est favorisée, au profit d'autres cases, comme si l'analyse effectuait un mouvement concentrique vers les bords du plateau.

3.4 La PRH se vérifie-t-elle ?

3.4.1 Quelles métriques ?

Suivant la littérature antérieure, nous définissons l'*alignement représentationnel* comme une mesure de la similarité des structures de similarité induites par deux représentations, c'est-à-dire une métrique de similarité sur les noyaux. Nous donnons la définition mathématique de ces concepts ci-dessous [9] :

— Une **représentation** est une fonction $f : X \to \mathbb{R}^n$ qui attribue un vecteur de

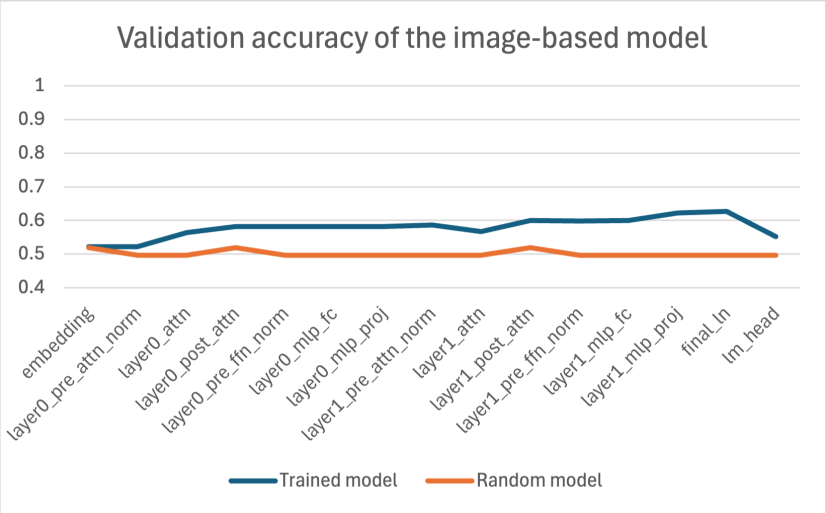


FIGURE 3.4 – Probing couche par couche pour les images

caractéristiques à chaque entrée dans un certain domaine de données X.

— Un **noyau**, $K : X \times X \to \mathbb{R}$, caractérise comment une représentation mesure la distance/similarité entre les points de données. $K(x_i, x_j) = \langle f(x_i), f(x_j) \rangle$, où $\langle \cdot, \cdot \rangle$ désigne le produit scalaire, $x_i, x_j \in X$ et $K \in \mathcal{K}$.

— Une **métrique d'alignement de noyau**, $m : \mathcal{K} \times \mathcal{K} \to \mathbb{R}$, mesure la similarité entre deux noyaux, c'est-à-dire à quel point la mesure de distance induite par une représentation est similaire à celle induite par une autre. Les exemples incluent la Distance de Noyau Centrée (CKA) [12], SVCCA [18], et les métriques des plus proches voisins [**?**].

3.4.2 Métriques d'alignement

Cycle KNN (K plus proches voisins cycliques)

Définition mathématique Soient A et B deux ensembles de vecteurs de caractéristiques, chacun de taille N. Pour un k donné :

1. Calculer les k-NN dans A : Pour chaque point dans A, trouver ses k plus proches voisins dans A.
2. Mapper ces voisins vers B.
3. Calculer les k-NN dans B pour ces points mappés.
4. Vérifier si le point original dans A est parmi ces voisins.

La métrique est la fraction de points qui "survivent" à ce cycle.

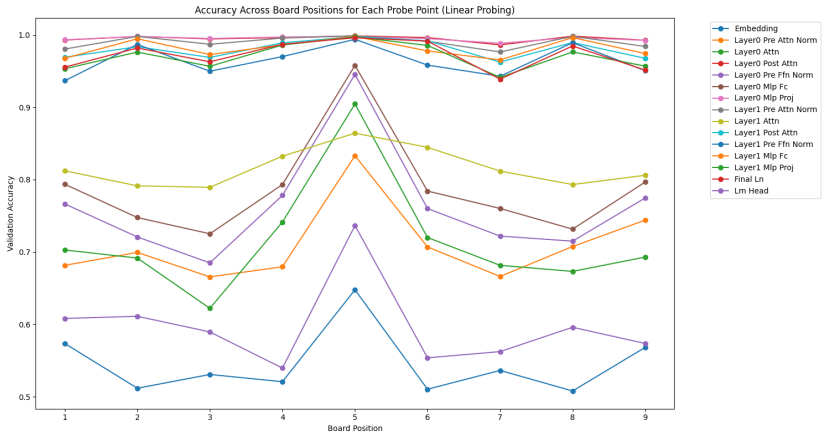


FIGURE 3.5 – Probing couche par couche pour le texte

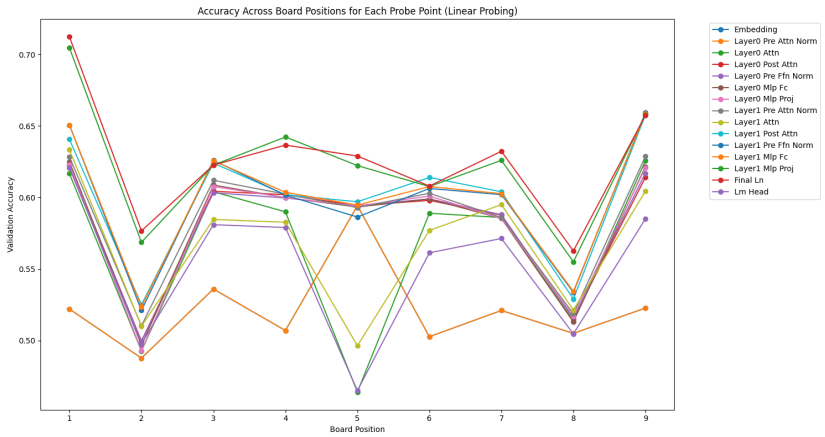


FIGURE 3.6 – Probing couche par couche pour les images

Formule

$$\text{Cycle\_KNN}(A, B) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(a_i \in \text{kNN}_B(\text{kNN}_A(a_i))) \tag{3.1}$$

où $\mathbb{I}$ est la fonction indicatrice, et $\text{kNN}_X(y)$ renvoie les k plus proches voisins de y dans X.

Interprétation

— Plage : $[0, 1]$

— Des valeurs plus élevées indiquent un meilleur alignement entre les deux espaces de caractéristiques.

— Une valeur de 1 signifie une parfaite consistance cyclique : la structure de voisi-

nage est parfaitement préservée lors du mapping de A vers B et retour.
— Des valeurs plus basses suggèrent que la structure locale n'est pas bien préservée entre les deux espaces.

KNN mutuel (K plus proches voisins mutuels)

Définition mathématique Pour chaque point dans A, vérifier s'il est parmi les k-plus proches voisins de son point correspondant dans B, et vice versa.

Formule

$$\text{Mutual\_KNN}(A, B) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(a_i \in \text{kNN}_B(b_i) \text{ ET } b_i \in \text{kNN}_A(a_i)) \tag{3.2}$$

Interprétation
— Plage : $[0, 1]$
— Des valeurs plus élevées indiquent une meilleure préservation mutuelle du voisinage entre les deux espaces.
— Une valeur de 1 signifie que tous les points sont des k-plus proches voisins mutuels dans les deux espaces.
— Des valeurs plus basses suggèrent que les structures de voisinage dans A et B sont différentes.

CKA (Alignement de Noyau Centré)

Définition mathématique CKA mesure la similarité entre deux noyaux après centrage.

Formule

$$\text{CKA}(K, L) = \frac{\langle K_c, L_c \rangle_F}{\|K_c\|_F \|L_c\|_F} \tag{3.3}$$

où K_c et L_c sont des matrices de noyau centrées, $\langle \cdot, \cdot \rangle_F$ est le produit scalaire de Frobenius, et $\| \cdot \|_F$ est la norme de Frobenius.

Interprétation
— Plage : $[0, 1]$
— Des valeurs plus élevées indiquent une plus grande similarité entre les matrices de noyau.
— Une valeur de 1 suggère que les deux noyaux capturent des relations identiques entre les points de données.
— Des valeurs plus basses indiquent que les noyaux capturent des relations différentes.
— CKA est invariant aux transformations orthogonales et à la mise à l'échelle isotrope.

SVCCA (Analyse de Corrélation Canonique des Vecteurs Singuliers)

Définition mathématique

1. Effectuer la SVD sur les deux matrices de caractéristiques : $A = U_A \Sigma_A V_A^T$, $B = U_B \Sigma_B V_B^T$
2. Prendre les d premiers vecteurs singuliers : $\tilde{U}_A, \tilde{U}_B$
3. Effectuer la CCA sur ces matrices tronquées

Formule

$$\text{SVCCA}(A, B) = \frac{1}{d} \sum_{i=1}^{d} \rho_i \tag{3.4}$$

où ρ_i sont les corrélations canoniques entre $\tilde{U}_A$ et $\tilde{U}_B$.

Interprétation

— Plage : $[0, 1]$
— Des valeurs plus élevées indiquent une corrélation plus forte entre les composantes principales des deux espaces de caractéristiques.
— Une valeur proche de 1 suggère que les principales directions de variation dans les deux espaces sont fortement alignées.
— Des valeurs plus basses indiquent que les composantes principales des deux espaces capturent différents aspects des données.
— SVCCA est moins sensible au bruit comparé à la simple CCA, en raison de l'étape initiale de SVD.

3.4.3 Conclusion

Ces métriques fournissent différentes perspectives sur l'alignement entre deux ensembles de caractéristiques :

— Cycle KNN et Mutual KNN sont plus sensibles aux structures locales et peuvent être utiles pour détecter des alignements fins.
— CKA est plus robuste et capture des similarités globales, mais peut manquer des détails fins.
— SVCCA est utile pour comparer les principales directions de variation, mais peut ignorer les caractéristiques moins importantes.

En pratique, il est souvent bénéfique d'utiliser plusieurs de ces métriques ensemble pour obtenir une image plus complète de l'alignement entre deux ensembles de caractéristiques.

3.4.4 Les résultats

On voit dans ces résultats que la similarité entre les deux modèles est très basse, en particulier concernant la métrique choisie dans l'article [9], `mutual_knn`.

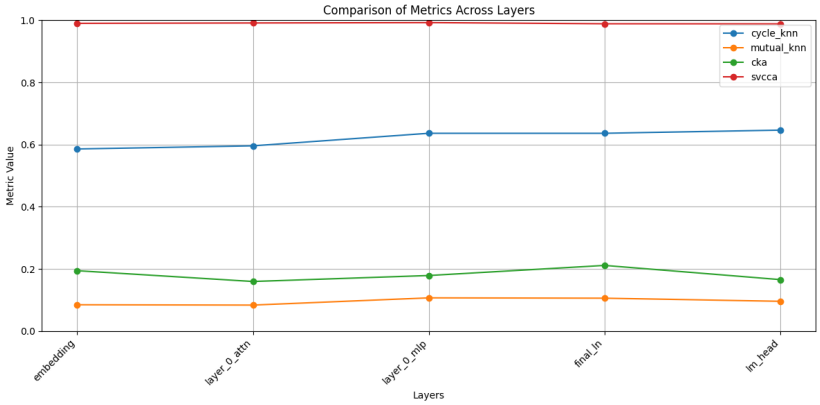


FIGURE 3.7 – Différentes métriques pour comparer la distance des représentations entre les deux modèles

En fait cela n'est pas forcément étonnant, puisque la tâche réalisée n'est pas la même : le transformer image doit prédire le prochain pixel, alors que le transformer texte doit prédire le prochain caractère ! De plus, nous avons déjà remarqué que pour le sondage (probing), le transformer texte avait une meilleure représentation du plateau que le transformer image, qui n'en a probablement pas besoin pour résoudre sa tâche.

3.5 À la recherche du neurone perdu

Dans cette section, nous explorons une nouvelle méthode pour interpréter les réseaux de neurones, les SAEs (*Sparse Auto Encoders*), qui nous permet de rendre les neurones *monosémantiques*. En effet, l'article qui inaugure l'usage des SAEs pour l'interprétabilité [19] fait la remarque que dans le LLM, les neurones qui se situent dans les couches de MLP sont *polysémantiques*, dans le sens où ils semblent encoder plusieurs informations liées au texte à la fois.

Cette polysémie neuronale ne facilite pas l'interprétation des réseaux de neurones, et ajoute au brouhaha et au côté "boîte noire" qui est si inconfortable pour nous. Pour cette raison, à la suite de [19], nous utilisons des SAEs pour rendre les neurones monosémantiques.

FIGURE 3.8 – Cette boîte noire qu'est le transformer

3.5.1 Les SAEs (et le MNIST)

Avant d'utiliser les SAEs (Sparse Auto-Encoders) directement dans le domaine des stratégies de jeux, j'ai d'abord expérimenté sur un jeu de données classique : le MNIST[1].

Qu'est-ce qu'un SAE ? En français *Auto-Encodeur Parcimonieux*, le SAE est un modèle qui prend en entrée une donnée x et doit reproduire cette même donnée en sortie $\hat{x}$ (d'où le terme "Auto"). L'objectif de ce modèle est d'encoder l'information essentielle contenue dans la donnée, sa *substantifique moelle*, dans une représentation cachée h.

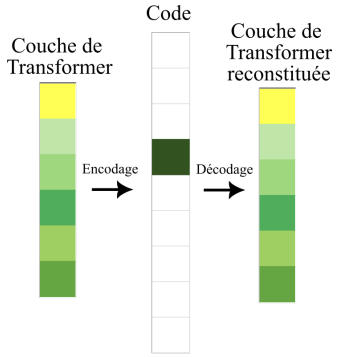


FIGURE 3.9 – Architecture d'un Sparse Auto-Encoder

Mathématiquement, le processus peut s'exprimer ainsi :

$$h = f(x) = \sigma(Wx + b)$$

$$\hat{x} = g(h) = \sigma(W'h + b')$$

où :
— f est la fonction d'encodage
— g est la fonction de décodage
— σ est une fonction d'activation non linéaire
— W, W' sont les matrices de poids
— b, b' sont les vecteurs de biais

La parcimonie (*sparsity*) garantit que la solution trouvée sera simple, appliquant ainsi le *rasoir d'Occam*, principe qui recommande de privilégier les explications les plus simples lorsque plusieurs solutions sont possibles. Cette parcimonie est obtenue en ajoutant un terme de régularisation $\Omega(h)$ à la fonction de perte (*loss*) :

1. MNIST (Modified National Institute of Standards and Technology database) est une base de données d'images de chiffres manuscrits largement utilisée pour tester les algorithmes d'apprentissage automatique.

$$\min_{W,W',b,b'} L(x, \hat{x}) + \lambda\Omega(h)$$

où L est une fonction de perte mesurant la différence entre l'entrée et la sortie reconstruite, et λ est un hyperparamètre contrôlant le degré de parcimonie.

Le SAE agit comme un "méta-réseau de neurones" dans le sens où il opère au niveau des représentations internes d'un autre réseau de neurones (ici, notre transformer). En d'autres termes, alors qu'un réseau de neurones classique traite des données brutes (images, texte, etc.), le SAE traite les activations neuronales elles-mêmes, ce qui en fait un réseau qui analyse un autre réseau— d'où le préfixe "méta". Cette approche permet d'étudier et de modifier les représentations internes du transformer de manière structurée et interprétable.

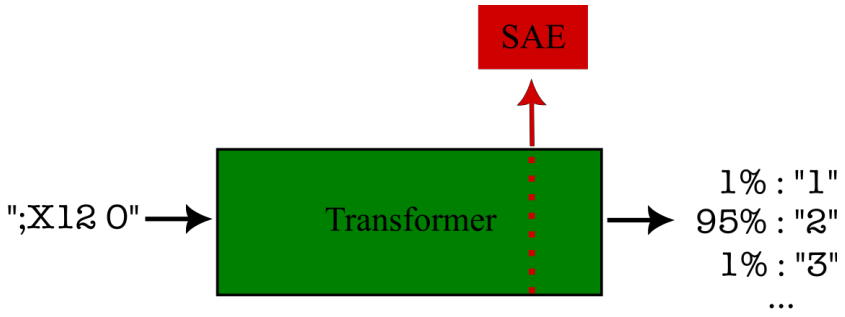


FIGURE 3.10 – Le SAE comme méta-réseau de neurones

Dans notre expérience, nous entraînons le SAE sur la deuxième couche du second MLP de notre transformer entraîné sur le MNIST, ce qui nous permet d'obtenir des visualisations révélatrices des représentations internes.

Une observation frappante est que le SAE associe un neurone spécifique à chaque chiffre. Cette spécialisation s'explique par la nature même du SAE qui, comme tout auto-encodeur non trivial[2], agit comme un compresseur d'information. Dans notre cas, bien que la couche de code soit de dimension supérieure à la couche d'entrée, la régularisation L1 force le modèle à minimiser le nombre de neurones activés. Cette contrainte de parcimonie est clairement visible dans nos résultats, où typiquement un seul neurone présente une forte activation, entouré de neurones quasi-inactifs. Le modèle est ainsi contraint de compresser l'information selon les caractéristiques les plus fondamentales, ici les chiffres eux-mêmes.

2. Un auto-encodeur trivial serait celui qui apprend simplement la fonction identité.

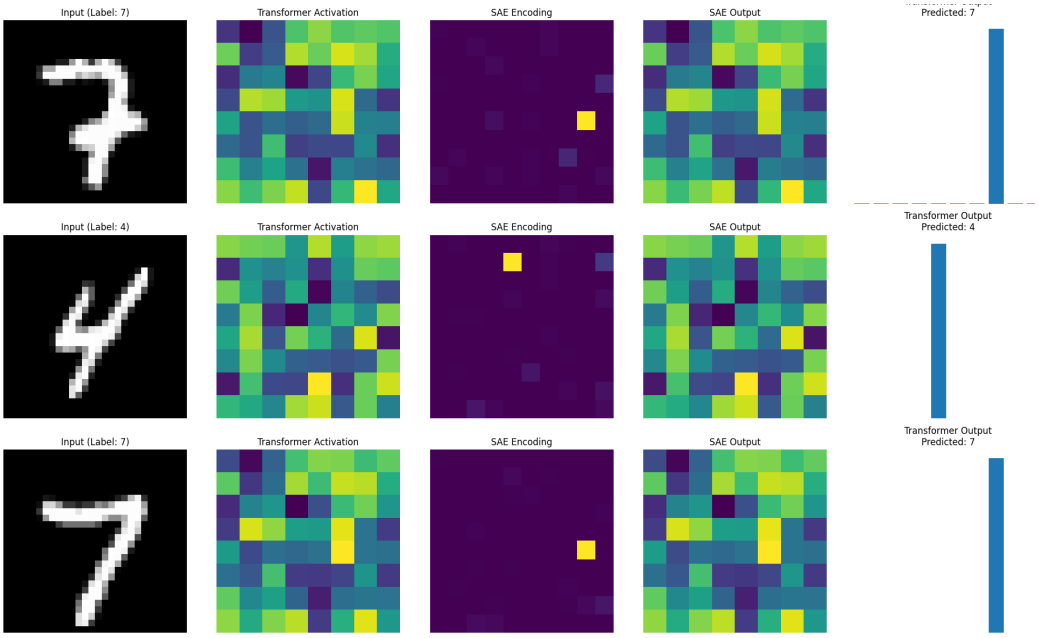


FIGURE 3.11 – Correspondance entre le concept de “7” et le neurone 59

3.5.2 Interventions (neurochirurgie)

Cette mise en évidence d’une forme de “logique” binaire (activé/désactivé, $\top/\bot$) dans le réseau de neurones nous permet d’aller plus loin. Nous pouvons intervenir directement sur l’output du transformer en modifiant l’état de la couche de MLP, réalisant ainsi une forme de “neurochirurgie” pour LLM.

Le processus d’intervention se déroule comme suit :

1. Le SAE est positionné comme intermédiaire de modification de couche
2. Un input est transmis au transformer en mode feedforward
3. Arrivé à la couche de MLP, celle-ci est encodée dans le SAE
4. Le neurone activé est modifié pour altérer sa signification
5. Le résultat est décodé via le SAE
6. La sortie modifiée est transmise à la couche suivante du transformer

Les résultats de ces interventions sont remarquables : nous pouvons effectivement manipuler le comportement du modèle. Cette approche rejoint les travaux d’Anthropic [19], qui ont réussi à faire croire à leur LLM (Claude) qu’il était le *Golden Gate Bridge*. Dans notre cas, nous pouvons modifier la perception des chiffres dans notre transformer entraîné sur le MNIST.

Ces résultats ouvrent des champs de recherche tant pour la compréhension de leur

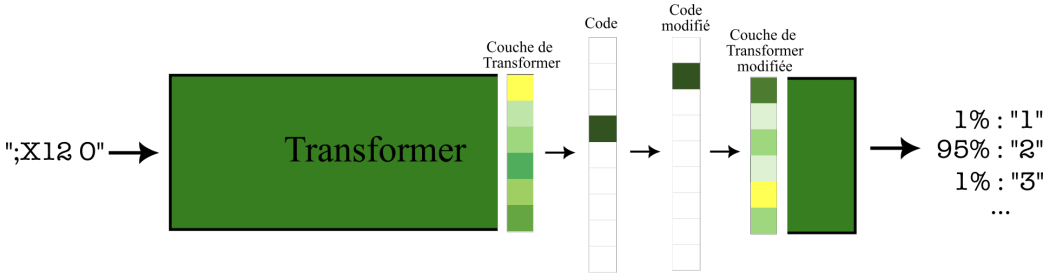


FIGURE 3.12 – Processus d'intervention neuronale

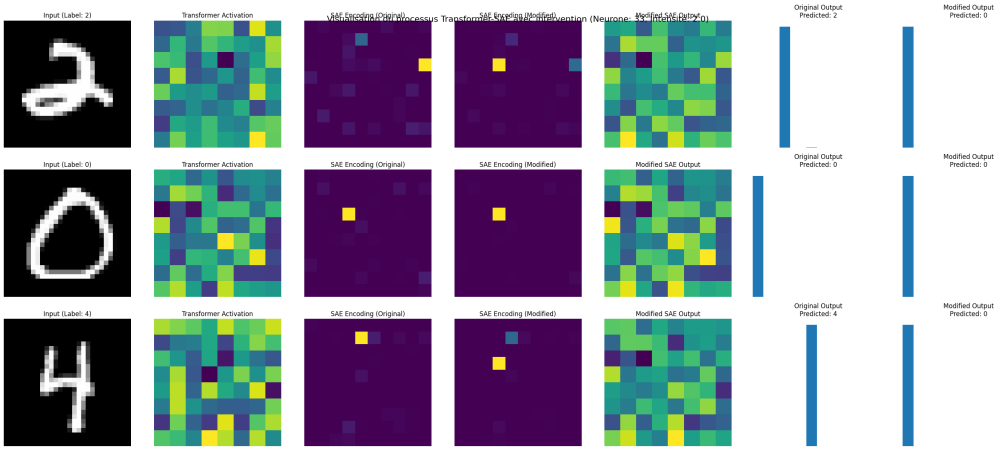


FIGURE 3.13 – Manipulation du Transformer pour une reconnaissance systématique du chiffre "0"

fonctionnement que pour l'identification de potentielles vulnérabilités. Ils sont également une réponse possible au *problème de l'alignement* (voir section RLHF).

Chapitre 4

Conclusion

4.1 Bilan du stage

Ce stage a été une expérience enrichissante et formatrice, me permettant d'explorer en profondeur le domaine fascinant des grands modèles de langage (LLM) à travers le prisme des jeux de plateau. L'objectif initial d'étudier les capacités émergentes des LLM et d'approfondir notre compréhension de leur fonctionnement interne a été atteint, ouvrant la voie à de nouvelles perspectives de recherche. Au cours de ces mois, j'ai pu mettre en pratique des concepts théoriques complexes, développer des méthodologies d'analyse novatrices et contribuer à l'avancement des connaissances dans le domaine de l'interprétabilité des modèles d'intelligence artificielle. Les résultats obtenus, bien que préliminaires, offrent des pistes prometteuses pour de futures investigations et soulignent l'importance de poursuivre les efforts dans cette direction.

4.2 Compétences acquises

Ce stage m'a permis de développer et d'affiner un large éventail de compétences essentielles pour une carrière dans la recherche en intelligence artificielle :

Lecture d'articles scientifiques : J'ai considérablement amélioré ma capacité à lire, comprendre et synthétiser des articles de recherche complexes, une compétence cruciale pour rester à jour dans ce domaine en évolution rapide. **Présentation d'un papier LaTeX pour une conférence** : J'ai acquis une expertise dans la rédaction et la mise en forme de documents scientifiques en utilisant LaTeX, respectant les normes et les exigences des conférences académiques. **Programmation** : Mes compétences en programmation se sont considérablement améliorées, notamment en Python, Bash et Zsh. J'ai appris à écrire du code efficace et bien structuré pour l'analyse de données et l'implémentation d'algorithmes complexes. **PyTorch** : J'ai acquis une solide maîtrise de PyTorch, un framework essentiel pour le développement et l'entraînement de modèles de deep

learning. **Cloud computing** : J'ai appris à utiliser efficacement les ressources de cloud computing pour l'entraînement de modèles gourmands en calcul, une compétence indispensable dans le domaine des LLM. **Présentation orale** : J'ai eu l'opportunité de présenter mes travaux lors d'une conférence de chercheurs, améliorant ainsi mes compétences en communication scientifique et en vulgarisation. **Rédaction scientifique** : La rédaction d'un article présentant mes résultats m'a permis de développer mes compétences en écriture scientifique, un aspect crucial de la recherche. **Théorie mathématique** : J'ai approfondi mes connaissances en mathématiques, notamment en ce qui concerne les distances et l'algèbre linéaire, des concepts fondamentaux pour l'analyse des modèles de langage.

4.3 Perspectives

Les travaux réalisés au cours de ce stage ouvrent quelques perspectives pour de futures recherches :

Généralisation à d'autres domaines : Les méthodes d'interprétabilité développées pour les jeux de plateau pourraient être adaptées et appliquées à d'autres domaines d'application des LLM, tels que le traitement du langage naturel ou l'analyse de données complexes. **Amélioration des modèles** : Les découvertes obtenues sur le fonctionnement interne des LLM pourraient être utilisées pour concevoir de nouvelles architectures de modèles plus interprétables et plus efficaces. **Éthique et responsabilité** : Une meilleure compréhension du fonctionnement des LLM pourrait contribuer à développer des systèmes d'IA plus transparents et plus fiables, répondant ainsi aux préoccupations éthiques croissantes dans ce domaine. **Collaboration interdisciplinaire** : Les résultats de cette recherche pourraient susciter des collaborations avec des chercheurs d'autres disciplines, comme la psychologie cognitive ou la philosophie, pour explorer les parallèles entre l'apprentissage machine et la cognition humaine. **Applications pratiques** : Les connaissances acquises sur les stratégies émergentes dans les jeux de plateau pourraient trouver des applications dans des domaines tels que la prise de décision assistée par IA ou l'optimisation de systèmes complexes.

En conclusion, ce stage a non seulement enrichi mes compétences et mes connaissances, mais a également ouvert de nouvelles voies passionnantes pour la recherche en IA. Il a renforcé ma détermination à poursuivre dans ce domaine et à contribuer à son avancement dans les années à venir.

Bibliographie

[1] Alain, G., & Bengio, Y. (2016). Understanding intermediate layers using linear classifier probes. arXiv preprint arXiv :1610.01644.

[2] Arora, S. and Goyal, A. (2023). Skill-Mix : A Flexible and Expandable Family of Evaluations for AI Models. arXiv preprint arXiv :2310.17567.

[3] Belinkov, Y. (2021). Probing Classifiers : Promises, Shortcomings, and Advances. Computational Linguistics, 48, 1-12.

[4] Carlini, N. (2023). Playing chess with large language models. https://nicholas.carlini.com/writing/2023/chess-llm.html.

[5] Daras, G., Shah, K., Dagan, Y., Gollakota, A., Dimakis, A. G., & Klivans, A. (2023). Ambient Diffusion : Learning Clean Distributions from Corrupted Data. arXiv preprint arXiv :2305.19256.

[6] Wikipedia : Elo rating article. https://en.wikipedia.org/wiki/Elo_rating_system. Accessed on : May 4, 2024.

[7] Erdős, P., & Renyi, A. (1960). On the Evolution of Random Graphs. Publication of the mathematical institute of the Hungarian Academy of Sciences, 17-61.

[8] Hewitt, J., & Manning, C. D. (2019). A Structural Probe for Finding Syntax in Word Representations. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies, Volume 1 (Long and Short Papers). Association for Computational Linguistics.

[9] Huh, M., Cheung, B., Wang, T., & Isola, P. (2024). The Platonic Representation Hypothesis. arXiv preprint arXiv :2405.07987.

[10] Karvonen, A. (2024). Emergent World Models and Latent Variable Estimation in Chess-Playing Language Models. arXiv preprint arXiv :2403.15498.

[11] Klabunde, M., Schumacher, T., Strohmaier, M., & Lemmerich, F. (2024). Similarity of Neural Network Models : A Survey of Functional and Representational Measures. arXiv preprint arXiv :2305.06329.

[12] Kornblith, S., Norouzi, M., Lee, H., & Hinton, G. (2019). Similarity of neural network representations revisited. In International Conference on Machine Learning (pp. 3519-3529). PMLR.

[13] Li, K., Viegas, F., Hopkins, A. K., Pfister, H., Bau, D., & Wattenberg, M. (2023). Emergent World Representations : Exploring a Sequence Model Trained on a Synthetic Task. In Proceedings of the International Conference on Learning Representations (ICLR).

[14] Lichess : Free Online Chess. https://lichess.org/. Accessed on : May 4, 2024.

[15] Power, A., Burda, Y., Edwards, H., Babuschkin, I., & Misra, V. (2022). Grokking : Generalization Beyond Overfitting on Small Algorithmic Datasets. arXiv preprint arXiv :2201.02177.

[16] OpenAI. GPT-4. https://openai.com/gpt-4, 2024.

[17] Pearce, A., Ghandeharioun, A., Hussein, N., Thain, N., Wattenberg, M., & Dixon, L. (2023). Do Machine Learning Models Memorize or Generalize ? arXiv preprint arXiv :2308.07613.

[18] Raghu, M., Gilmer, J., Yosinski, & Sohl-Dickstein, J. (2017). SVCCA : Singular vector canonical correlation analysis for deep learning dynamics and interpretability. In Advances in neural information processing systems (pp. 6076-6085).

[19] Templeton, A., Conerly, T., Marcus, J., Lindsey, J., Bricken, T., Chen, B., Pearce, A., Citro, C., Ameisen, E., Jones, A., Cunningham, H., Turner, N. L., McDougall, C., MacDiarmid, M., Freeman, C. D., Sumers, T. R., Rees, E., Batson, J., Jermyn, A., Carter, S., Olah, C., & Henighan, T. (2024). Scaling Monosemanticity : Extracting Interpretable Features from Claude 3 Sonnet. Transformer Circuits Thread. https://transformer-circuits.pub/2024/scaling-monosemanticity/index.html

[20] Varma, V., Shah, R., Kenton, Z., Kramár, J., & Kumar, R. (2023). Explaining grokking through circuit efficiency. arXiv preprint arXiv :2309.02390.

Annexe A

Annexe

A.1 L'entraînement pratique des LLMs

Dans le monde sauvage du machine learning moderne, il existe à certains endroits des oasis permettant à de petits étudiants d'entraîner leurs propres LLMs. Je veux ici partager quelques-unes des oasis que mes recherches m'ont fait trouver dans le domaine.

A.1.1 NanoGPT, TensorDock et TMUX

D'abord, le code détaillé (sans utiliser `transformer` de `pytorch`!) d'un transformer hackable à merci m'a été donné par Andrej Karpathy, qui entretient le dépôt GitHub 'nano GPT' qui est très utile pour rentrer dans les entrailles du modèle.

Ensuite, l'entraînement de tels modèles est assez coûteux en ressources computationnelles. Il a fallu trouver des puces qui s'adaptent parfaitement au code et au niveau de ressources qu'il demande. Il se trouve que pour l'entraînement des modèles d'échecs, il faut au moins 16 Go de GPU (le GPU est le nerf de la guerre[1]).

De nombreuses plateformes proposent des GPUs, mais beaucoup ont refusé, ou pris beaucoup trop de temps à répondre à mon besoin de GPU (je pense à Azure AI ou à Google Cloud). D'autres offres, alléchantes au départ vu l'offre et les prix (Lambda) m'ont été impossibles pour des raisons de carte : il faut une authentique *carte de crédit*, et non une carte de débit (celle que la plupart des gens ont, y compris moi...).

J'ai longtemps cherché, et j'ai fini par tomber sur un fournisseur de GPUs dont le site ne payait pas de mine (en tout cas à l'époque!) : TensorDock. Leur interface est minimaliste, voire austère (rien à voir avec les cockpits d'avions d'Azure AI et Google Cloud), mais dans le terminal (via SSH), je pouvais faire tout ce dont j'avais besoin.

La puce que j'ai finalement utilisée (après plusieurs essais) est une NVIDIA A6000, qui alliait plusieurs avantages : un bas prix (le prix du marché d'une unité était d'environ

1. Et NVIDIA est le vendeur de pelles de tous les explorateurs de ce nouveau *Far West*

0,50€ de l'heure à l'époque de l'entraînement, et a diminué depuis). L'entraînement des 4 modèles d'échecs m'a pris une semaine en tout, et 3 jours par modèle (je les entraînais en parallèle sur 3 GPUs différents).

À propos, l'outil qui me fut très utile pour cela était TMUX, qui permet de quitter la fenêtre du terminal sans que l'algorithme termine... Ce qui est très utile quand on doit entraîner un modèle pendant 3 jours.

Concernant ma méthode, je travaille dans Google Colab (outil magnifique de démocratisation de la puissance de calcul !). Je faisais mes essais avec, puis je copiais les librairies utilisées avec `pip freeze`, ce qui permet de générer un `requirements.txt` à installer directement dans la machine virtuelle.

A.1.2 Les hyperparamètres de mes modèles

Dans le monde du machine learning moderne, la multiplication des hyperparamètres fait de l'art du machine learning une discipline qui ressemble plus à l'alchimie qu'à une science fournissant théorie. Toute tentative de justification de ceux-ci relève rapidement de la plus haute théologie byzantine. Mais à défaut de théorie, je vous donne mes hyperparamètres, en expliquant de temps en temps l'intuition qui m'a fait les choisir ainsi (souvent c'est juste de l'imitation !)

- Nombre de couches (n_layer) : 8 (pourquoi pas ? Karvonen a testé deux nombres : 8 et 16. Ce sont des puissances de 2, c'est joli, et il n'y a pas vraiment de raison à ça).
- Nombre de têtes d'attention (n_head) : 4 (pareil, pas vraiment de raison à part le fait que nous voulons plusieurs têtes pour diversifier les "méta-contextes")
- Dimension du modèle (n_embd) : 256 (puissance de 2 toujours, le seul cas où la puissance de 2 est justifiée est pour la taille du batch, pour des raisons de rapidité, mais ici non)
- Taille du bloc (block_size) : 1024 tokens (la longueur max d'une partie d'échecs on va dire)

Pour les hyperparamètres d'entraînement :

- Taille de batch : 2048 (on a dit pour des raisons d'efficacité computationnelle, qui intervient lors de la division du dataset en batches)
- Taux d'apprentissage initial : 3e-4 (Pas trop petit, et pas trop grand. Remarquez qu'il varie au cours de l'apprentissage[2])
- Décroissance du taux d'apprentissage : jusqu'à 3e-5
- Nombre maximal d'itérations : 10000 (c'est la limite maximale d'itérations d'apprentissage)

2. La décroissance du taux d'apprentissage est important, car un modèle apprend beaucoup plus dans les premières itérations que dans les suivantes. Un taux d'apprentissage trop élevé cause un suplace, un peu comme Tarzan ayant la bougeote, et ne parvenant pas à atteindre le fond de la vallée puisqu'il saute d'un bord de celle-ci à l'autre trop rapidement

— Warmup : 2000 itérations (le Warmup (ou échauffement) c'est la phase durant laquelle le taux d'apprentissage décroît, ici, pour les 2000 premières itérations)