## Groupe de lecture : "Attention is all you need"

Un apprentissage supervisé

**Elève :** Estéban Collas, esteban.collas@ens-lyon.fr
**Supervisé par :** Aurélien Garivier, aurelien.garivier@ens-lyon.fr

Mon groupe de lecture s'est déroulé en 3 parties :

1. Etude de l'article "Attention is all you need" et des Transformers
2. Etude théorique de la structure des RNN, LSTM et GRU
3. Réalisation d'un N-GRAM en python, d'abord naïvement, puis à l'aide d'un prefix tree.

# 1 Retour sur la 1ère partie :

J'ai principalement étudié la structure des Transformers, afin de comprendre les composantes apparaissant dans le schéma suivant :

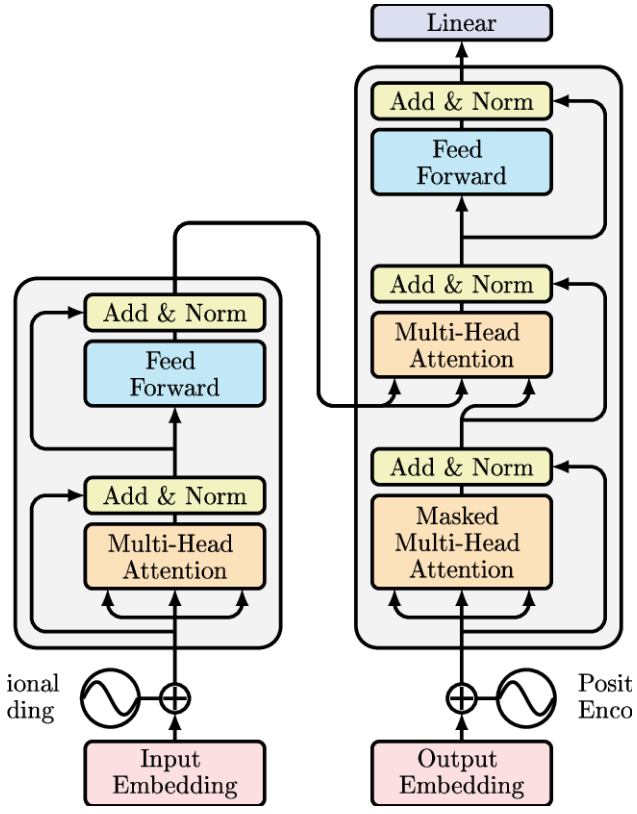



**Figure 1:** Le Transformer - architecture du modèle

J'en ai fait une présentation. Voilà une explication succinte du rôle de chaque partie dans la structure de base du Transformer :

**Encoder/Decoder :** Le bloc gris de gauche est appelé l'Encoder, il sert à traiter le prompt initial pour lui donner un rôle différent à celui des mots générés par le modèle. Le bloc gris de droite

est le Decoder, il sert à traiter les mots générés par le modèle et à générer le prochain token. Une fois qu'un premier token à été généré, on ne travaille plus qu'avec le Decoder et la valeur de sortie originale de l'Encoder, qui a été gardée en mémoire (en fait, certains modèles, comme Chat GPT, n'ont pas d'Encoder. Tout est traité comme si ça avait été généré par le Transformer)

**Input/Output embedding :** Dans les deux cas, l'Embedding prend en entrée des mots en langage naturel et les transforme en tokens, puis en vecteurs. En d'autres termes, on traduit l'entrée dans le "langage du Transformer". l'Input Embedding traite les mots de l'input, et l'Output Embedding ceux générés par le modèle (et a donc accès aux tokens directement). Ils ont souvent été entraînés avant le reste du Transformer, et sont utilisés tel quel.

**Positional Encoding :** En sortie de l'Embedding, on reçoit une matrice contenant les vecteurs représentant les tokens dont on essaie de prédire la suite. Le positional encoding rajoute à ses vecteurs des valeurs afin de prendre en compte leur position (typiquement à l'aide de fonctions semblables à $P_E(pos, 2i) = sin(\frac{pos}{10000^{(\frac{2i}{dim})}})$, où $pos$ est la position du mot dans le prompt, $dim$ est la dimension du modèle, un hyperparamètre, et $2i$ est la ligne de la matrice que l'on est en train de traiter (pour les $2i + 1$, on a une formule similaire avec un cos)). Cela est important car les blocs suivants, en particulier le bloc Multi-Head-Attention, ne prennent pas en compte la position du vecteur dans leur matrice d'entrée. Il ne nécessite pas d'entraînement.

**Multi-Head Attention :** Le bloc principal du Transformer. On reçoit en entrée une matrice (encodant les tokens du prompt, modifiée par les autres blocs). On renvoie une nouvelle matrice de même dimensions, créée en lisant la matrice d'entrée. Pour chaque vecteur de la matrice d'entrée, on va créer un nouveau vecteur dans la matrice de sortie en fonction des autres vecteurs de l'entrée et des paramètres trouvés au moment de l'entraînement du modèle. C'est ce bloc qui fait tous les liens entre les mots du prompt, trouve le sujet d'un verbe, etc. Il est entraîné.

**Add & Norm :** Ce bloc appraît à la suite de chaque Multi-Head Attention et de chaque Feed-forward. Son fonctionnement est simple. Il prend en entrée 2 matrices. Celle en sortie du bloc qu'il surmonte, et celle en entrée du bloc qu'il surmonte. Il va sommer les deux matrices, puis normaliser le résultat. On somme les matrices pour que le bloc qu'il surmonte ne renvoie que la différence entre la matrice voulue et la matrice qu'il a en entrée. On normalise le résultat pour avoir un entraînement plus efficace. La partie "normalisation" du bloc est entraînée.

**Feed Forward :** C'est un réseau de neurones complètement connectés. Il sert à déterminer des relations non-linéaires entre les vecteurs de la matrice qu'il prend en entrée. Il est entraîné.

**Linear :** Transforme la sortie du Decoder en logits, sur lesquels on va appliquer un softmax pour déterminer le prochain token à renvoyer. Il est entraîné.

## 2 Retour sur la 2e partie :

Je me suis intéressé à l'apparition de la structure de Transformer et à son lien avec la structure de RNN (Recurrent Neural Network), et des cas particuliers de RNN que sont les LSTM (Long Short-Term Memory) et les GRU (Gated Recurrent Unit).

Voilà les schémas représentant le bloc principal de ses structures (explications très claires ici) :

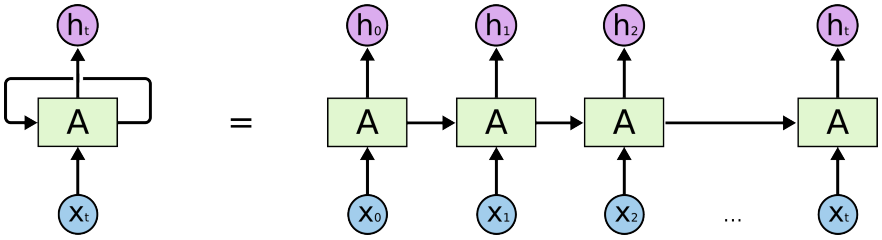



**Figure 2:** réseau RNN

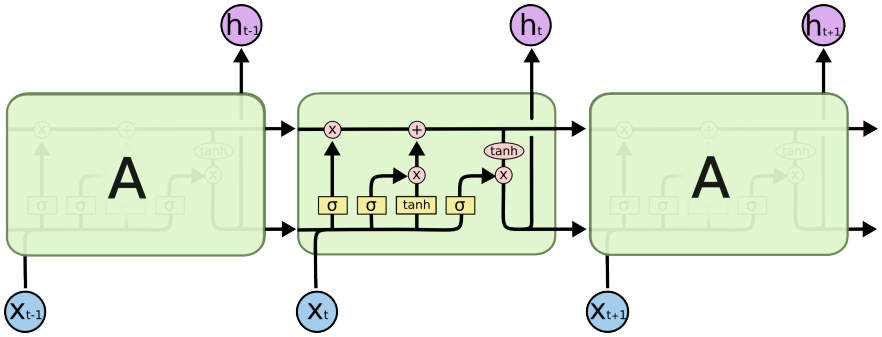
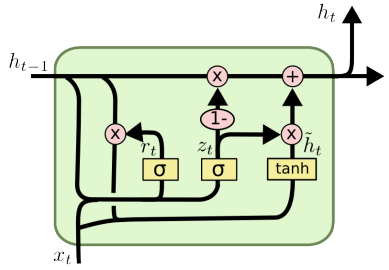



**Figure 3:** réseau LSTM




$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

**Figure 4:** module d'un réseau GRU

## 3 Retour sur la 3e partie :

J'ai codé un N-GRAM en python, c'est à dire un modèle de langage simple basé sur une chaîne de Markov, cherchant à prédire le mot suivant en se basant sur les N-1 derniers mots d'un prompt.

J'en ai fait une première implémentation à l'aide d'une structure naïve utilisant des dictionnaires (qui n'apparaît pas sur github car elle n'était pas très heureuse), puis avec un prefix tree. L'idée dans le prefix tree est de prédire le prochain mot en lisant une suite de token à l'envers, et en allant chercher plus d'informations si les tokens considérés jusque là ont déjà été vus de nombreuses fois.

Par exemple, si on cherche le token suivant "the cat sat on the", on va d'abord chercher un mot probable sans contexte, par exemple "a", et si on n'a pas souvent vu le bigram "the X", on renvoie le mot trouvé (ce qui donnerait "the cat sat on the a"). Mais si on a souvent vu le bigram "the X", on va chercher un mot probable avec le contexte "the", puis (si on a souvent vu "on the X") "on the", "sat on the", ... et ainsi de suite jusqu'à ne pas avoir souvent vu le n-gram considéré ("pas souvent" = moins de 10 fois, dans mon code. C'est un hyperparamètre), ou jusqu'à atteindre "the

cat sat on the".

J'ai généré les données d'entraînement de mon modèle à l'aide d'un LLM (llama3.2:1b). J'ai donc appris à utiliser ollama et llama.cpp.

**Code sur GitHub** : https://github.com/Ae-rys/n_gram_project

Voilà un graphique montrant la perplexité (valeur représentant à quel point leurs prédictions pour le mot suivant correspondait à la réalité) de mes différents N-grams sur du texte extrait de Wikipédia :

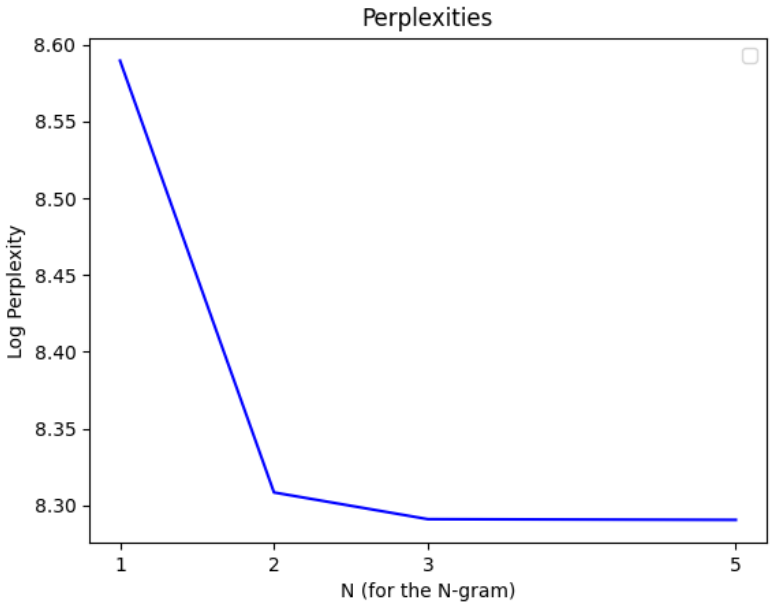



**Figure 5:** Perplexités (j'affiche la valeur absolue du logarithme de la perplexité de l'unigram, le bigram, le trigram et le 5-gram)

On voit sur mon graphique que la perplexité baisse quand on augmente le nombre de mots utilisés pour la prédiction, ce qui est attendu. On remarque également que l'amélioration est de moins en moins importante au fur et à mesure qu'on regarde plus de mots. Je tiens à noter qu'une grande part de la perplexité vient du fait que de nombreux mots n'ont jamais été rencontrés par mes modèles (lorsque je tombe sur un tel mot, j'ajoute un facteur 1/(nombre de mots total) à la perplexité, ce qui fait beaucoup croître la valeur absolue de son logarithme). Je n'ai pas pu entraîner mes modèles sur plus de mots, car sinon mes fichiers de sauvegarde étaient trop lourds et cela faisait crasher mon code. Une amélioration possible serait de sauvegarder plsu efficacement mes modèles pour voir jusqu'à quel point on peut descendre en perplexité. Je n'ai pas trouvé de méthode simple pour le faire.

## References

Globalement, je me suis beaucoup aidé dans mes recherches des sites suivants, qui présentent des articles de très bonne qualité et qui donnent beaucoup d'intuition sur les thèmes traités :

- https://distill.pub/

- https://colah.github.io/

Et également d'autres sites plus généralistes comme :

- https://www.ibm.com
- https://www.wikipedia.org/ (évidemment)