

Rapport de Projet de Fin d'Étude

LLM et leurs applications

tayheau

Juin 2024

Abstract

Les Languages Models semblent trouver leur origine lors des débuts du domaine de l'intelligence artificielle. Ainsi, les travaux de Joseph Weizenbaum en 1966 sur le programme ELIZA[33], qui, en utilisant quelques règles simples, en font l'un des premiers LM capable d'imiter le langage humain. Le monde de l'IA et plus précisément le domaine de LM est dans son état actuel notamment grâce à deux récentes avancées[6] :

- en 2012 lorsque Krizhevsky et al.[16] ont prouvé qu'un algorithme pensé il y a plus de 20 ans[18] pouvait surperformer le state-of-the-art de reconnaissance d'image seulement grâce à un modèle et un dataset 100x plus grands
- en 2017 par Vaswani et al.[31] avec l'introduction des modèles Transformers, qui contrairement aux architectures RNNs[11] et LSTMs[9], se base entièrement sur des mécanismes d'attention et n'utilise pas de convolutions ni de récurrences, permettant d'effectuer des calculs plus parallélisables et donc d'augmenter considérablement l'efficacité de l'entraînement et de l'inférence.

Que ce soit grâce aux gains de productivité[35], de créativité ou tout simplement grâce à la curiosité humaine, les agents IA se sont très rapidement démocratisés suite à la mise à disposition publique de GPT-3[3] et ont su gagner une place importante dans notre quotidien.

Au travers de ce rapport, nous allons notamment expliquer comment nous avons réimplémenté la version de 124M de paramètre de l'agent GPT2[26], et comment nous avons finetune ce dernier pour en faire un compositeur de chansons.

Contents

I	Définition des différents concepts clés liés aux Languages Models	2
I.1	Language Models	2
I.2	Machine Learning	2
I.3	Deep Learning	3
II	Description des principales familles de LM	5
II.1	Modèle n-grams	5
II.2	Modèles basés sur les réseaux de neurones	5
III	State of the Art	10
III.1	L'architecture des Transformers	10
III.2	LMs et LLMs	13
III.3	Pré-entraînement des LLMs	13
III.4	Fine-Tuning LLMs	13
III.5	Méthodes d'évaluation	14
IV	Notre application de génération	18
IV.1	Contexte de réalisation du projet	18
IV.2	Model.py	18
IV.3	Train.py	20
IV.4	Premier modèle	23
IV.5	Pistes d'amélioration	26
V	Conclusion	26

I Définition des différents concepts clés liés aux Languages Models

I.1 Language Models

D'après Wikipédia, “un modèle de langage est un modèle statistique de la distribution de séquences de symboles distincts (lettres, phonèmes, mots) dans une langue naturelle. Un modèle de langage vise fondamentalement à prédire le mot suivant dans une séquence de mots.”

I.2 Machine Learning

Le Machine Learning est une branche de l'intelligence artificielle qui consiste à développer des algorithmes et des modèles permettant à un système d'apprendre à partir de données et de s'améliorer automatiquement sans être explicitement programmé. Il permet aux ordinateurs de détecter des motifs, de prendre des décisions et de faire des prédictions en se basant sur des exemples et des expériences passées.

La majorité des algorithmes de ML peuvent être classés en deux catégories :

- l'apprentissage supervisé
- l'apprentissage non-supervisé.

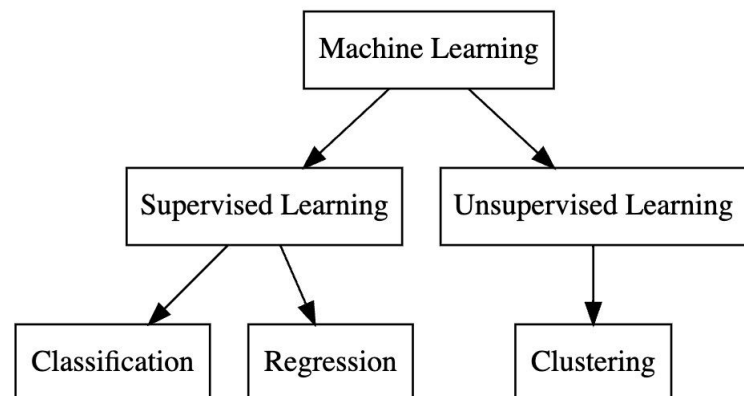


Figure 1: Diagramme du Machine Learning

Le réel challenge dans l'utilisatation du ML est de trouver un algorithme qui sera le plus performant pour une tache et un dataset donné. Généralement, cette tache demande plusieurs itérations avec différents algorithmes pour trouver le meilleur.

Supervised Learning

D'après Kelleher et Tierney[15], le but du *Supervised Learning* est d'apprendre une fonction qui relie les valeurs des attributs décrivant une instance à la valeur d'un autre attribut, connu sous le nom d'attribut cible, de cette instance.

Cette méthode est dite "supervisée" car chaque instance dans le dataset contient les données d'entrées et les données de sorties (données cibles).

Unsupervised Learning

À l'inverse de l'Apprentissage Supervisé, l'Apprentissage Non-Supervisé n'a pas de données cibles. De ce fait, au lieu de résoudre le problème spécifique de relier les données d'entrées aux données cibles, l'algorithme aura la tâche plus générale de trouver des régularités dans les données[15].

La forme la plus commune d'Apprentissage Non-Supervisé est *l'analyse de cluster*, où l'algorithme va chercher des cluster d'instances de données qui sont plus similaires entre elles par rapport aux autres.

I.3 Deep Learning

Historiquement, le Deep Learning appartient au domaine plus large du Machine Learning étant donné qu'il consiste principalement en des méthodes qui sont capables d'apprendre des représentations à partir de jeux de données. Les techniques utilisées proviennent principalement des réseaux de neurones artificiels.[20]

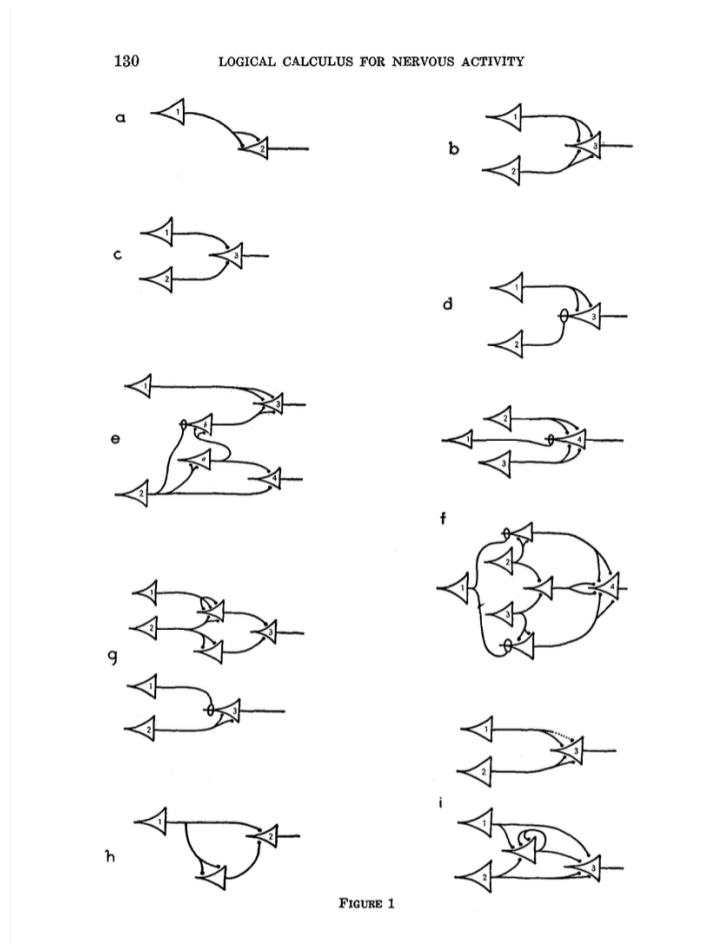


Figure 2: Représentation de McCulloch et al. des calculs logiques pour l'activité neuronale.

La qualification de "Deep" provient notamment du fait que les modèles sont de longues compositions de relations, qui se sont révélé être plus performant que les autres modèles.[6] La grande modularité, versatilité et évolutivité des modèles de DL font de cette sous-discipline du ML un domaine vaste et complexe.

Apprentissage par transfert

C'est une technique où un modèle pré-entraîné sur une grande quantité de données est adapté à une tâche spécifique avec une quantité moindre de nouvelles données. Par exemple, GPT-3 est un modèle de langage pré-entraîné qui peut être adapté pour diverses applications spécifiques.

Transformers

Architecture de réseau de neurones introduite en 2017 par Vaswani et al.[31] qui a révolutionné les modèles de langage. Les Transformers utilisent des mécanismes d'attention pour traiter les séquences de manière parallèle, ce qui les rend plus efficaces que les RNNs pour des tâches de grande envergure. BERT, GPT-2, et GPT-3 sont des exemples de modèles basés sur les Transformers.

Embeddings (Représentations vectorielles)

C'est une technique où les mots sont représentés par des vecteurs dans un espace continu. Word2Vec et GloVe sont des exemples d'algorithmes générant des embeddings. Ces représentations capturent des similitudes sémantiques entre les mots.

Pré-entraînement et Fine-tuning

Le pré-entraînement est la phase où un modèle de langage est initialement entraîné sur un large corpus de données textuelles. Le fine-tuning est la phase où le modèle est affiné avec un ensemble de données spécifique pour une tâche particulière.

Perplexité

C'est une mesure utilisée pour évaluer la qualité des modèles de langage. La perplexité quantifie la capacité du modèle à prédire une séquence de mots. Plus la perplexité est faible, meilleur est le modèle.

II Description des principales familles de LM

Introduits au milieu des années 60 ([33]), les modèles de langage peuvent être classés en différentes familles selon les techniques et architectures utilisées. Deux des principales familles sont les modèles n-grams et les modèles basés sur les réseaux de neurones.

II.1 Modèle n-grams

Les modèles n-grams sont des modèles de langage probabilistes qui prédisent la probabilité d'un mot en fonction des $n - 1$ mots précédents.[22] En principe, un n-gramme est une séquence de n mots. Par exemple, un bigramme (2-gramme) utilise un mot précédent pour prédire le suivant, et un trigramme (3-gramme) utilise deux mots précédents.

La probabilité d'une phrase $P(w_1, w_2, \dots, w_n)$ est approximée par:

$$P(w_1, w_2, \dots, w_n) = \prod_{i=1}^n P(w_i | w_{(i-(n-1))}, \dots, w_{(i-1)})$$

Les modèles n-grams présentent plusieurs avantages qui les rendent utiles pour certaines applications de traitement automatique des langues. Leur principale force réside dans leur simplicité et leur facilité de mise en œuvre. Ils sont relativement rapides à entraîner et à déployer, car ils se basent sur des calculs de fréquences des séquences de mots dans des corpus de texte. Cette simplicité permet également une compréhension et une interprétation aisées des résultats, ce qui est précieux pour des tâches de base ou pour des environnements à ressources limitées. De plus, les n-grams peuvent donner de bonnes performances sur des séquences courtes et des tâches simples, où les dépendances contextuelles sont limitées à quelques mots.

Cependant, les modèles n-grams présentent également des inconvénients notables. Leur principale faiblesse est leur incapacité à capturer des dépendances à long terme entre les mots, car ils ne considèrent qu'un nombre fixe de mots précédents pour prédire le suivant. Cela limite leur efficacité dans des contextes où le sens d'une phrase dépend de relations entre des mots distants. En outre, l'utilisation de grands n-grams (grandes valeurs de n) entraîne une explosion combinatoire, augmentant considérablement le nombre de paramètres à estimer et nécessitant de vastes quantités de données pour obtenir des estimations fiables. Cette complexité accrue peut également entraîner des problèmes de stockage et de calcul. En résumé, bien que les modèles n-grams soient pratiques et performants pour des tâches simples, ils montrent leurs limites face à des exigences plus complexes et des dépendances linguistiques à long terme.

II.2 Modèles basés sur les réseaux de neurones

Les modèles basés sur les réseaux de neurones utilisent des architectures de Deep learning pour apprendre des représentations complexes et des dépendances à long terme dans les données textuelles. L'explosion du gradient constitue une limite à ce genre de modèle.

Réseaux de neurones récurrents (RNN)

Les réseaux de neurones récurrents (RNN) sont conçus pour traiter des données séquentielles, ce qui les rend particulièrement adaptés pour des tâches telles que la modélisation de langage, la traduction automatique, et la reconnaissance vocale.[27] Contrairement aux réseaux de neurones traditionnels, les RNNs possèdent des connexions récurrentes qui leur permettent de conserver une "mémoire" des états précédents à chaque étape de la séquence. Cela signifie qu'ils peuvent prendre en compte le contexte précédent lors de la prise de décisions pour l'étape actuelle. Les RNNs sont entraînés par rétropropagation à travers le temps (Backpropagation Through Time [34], BPTT), une extension de l'algorithme de rétropropagation.

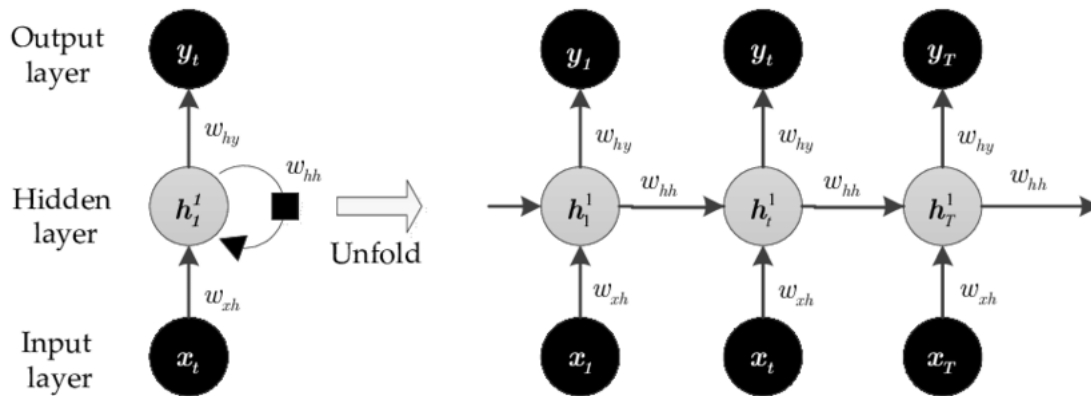


Figure 3: Représentation schématique d'un RNN.

Ainsi, à un temps t donné, la cellule $h^{(t)}$ va calculer son nouvel état $a^{(t)}$ de la manière suivante :

$$a^{(t)} = g(W_a a^{(t-1)} + R_a x^{(t)} + b_a)$$

avec W_a et R_a les matrices de poids de $a^{(t-1)}$ et $x^{(t)}$ et b_a vecteur de biais et $g()$ généralement $\tanh()$.

L'output $y^{(t)}$ au temps t peut être calculée :

$$y^{(t)} = \text{softmax}(W_y a^{(t)} + b_y)$$

avec W_y la matrice de poids de $a^{(t)}$ et b_y un vecteur de biais.

Cependant, les RNNs standards présentent des limitations lorsqu'il s'agit de capturer des dépendances à long terme dans les données. En effet, au fur et à mesure que la séquence s'allonge, les gradients peuvent devenir extrêmement petits (gradient qui disparaît) ou excessivement grands (gradient qui explose), ce qui complique l'apprentissage efficace de longues séquences. Ces limitations ont conduit au développement de variantes améliorées des RNNs, notamment les Long Short-Term Memory (LSTM) et les Gated Recurrent Unit (GRU).

Long Short-Term Memory (LSTM)

Les réseaux de neurones à mémoire à long terme (LSTM) ont été introduits pour surmonter les problèmes de gradient des RNNs standards.[9] [10]

On y retrouve le concept de récursivité : en effet, ce dernier consiste en une suite récursive de sous-réseaux (aussi appelés Memory Blocks). L'idée derrière ces Memory Blocks est de maintenir son état au cours du temps et de réguler le flux d'information au travers de "gates" non-linéaires. [30]

Mais la particularité se trouve dans l'utilisation d'une architecture de cellules spécifiques avec des mécanismes de portes (aussi appelés "gates") pour réguler le flux d'informations divisé en deux sous-flux : le **cell state** et le **hidden state**.

Le **cell state** (ou aussi long term memory) est le flux de données qui va venir contrer le problème des vanishing/exploding gradient en n'étant pas influencé par des poids. À l'inverse, le **hidden state** sera la mémoire à court terme, influencée par les poids.

Il y a trois gates dans une Memory Block d'un LSTM :

- Le **forget gate**, qui à l'aide d'une fonction sigmoïde, va déterminer le ratio d'information à conserver en provenance de la cell state $c^{(t-1)}$ de la cellule précédente. Le calcul de la fonction $f^{(t)}$ se fait:

$$f^{(t)} = \sigma \left(W_f x^{(t)} + R_f h^{(t-1)} + b_f \right)$$

avec W_f et R_f les poids respectifs pour $x^{(t)}$ et $h^{(t-1)}$ et b_f un vecteur de biais.

- Le **candidat pour le cell state** $z^{(t)}$ au temps t est calculé en fonctions du hidden state précédent $h^{(t-1)}$ et de l'input de la cellule x_t :

$$z^{(t)} = g \left(W_z x^{(t)} + R_z h^{(t-1)} + b_z \right)$$

avec W_z et R_z les poids respectifs pour $x^{(t)}$ et $h^{(t-1)}$ et b_z un vecteur de biais et $g()$ généralement $\tanh()$.

- L'**input gate** qui va déterminer le % d'information à retenir provenant du candidat pour le cell state avant qu'il soit ensuite ajouté au cell state.

$$i^{(t)} = \sigma \left(W_i x^{(t)} + R_i h^{(t-1)} + b_i \right)$$

avec W_i et R_i les poids respectifs pour $x^{(t)}$ et $h^{(t-1)}$ et b_i un vecteur de biais.

- L'**output gate** qui va déterminer la nouvelle valeur de l'hidden state au temps t $h^{(t)}$ en se basant sur le cell state au temps t $c^{(t)}$. Ainsi nous avons :

$$c^{(t)} = c^{(t-1)} \odot f^{(t)} + i^{(t)} \odot z^{(t)}$$

où \odot est le produit matriciel d'Hadamard (element-wise product).

$$o^{(t)} = \sigma \left(W_o x^{(t)} + R_o h^{(t-1)} + b_o \right)$$

avec W_o et R_o les poids respectifs pour $x^{(t)}$ et $h^{(t-1)}$ et b_o un vecteur de biais. Finalement

$$h^{(t)} = g(c^{(t)}) \odot o^{(t)}$$

où \odot est le produit matriciel d'Hadamard (element-wise product)

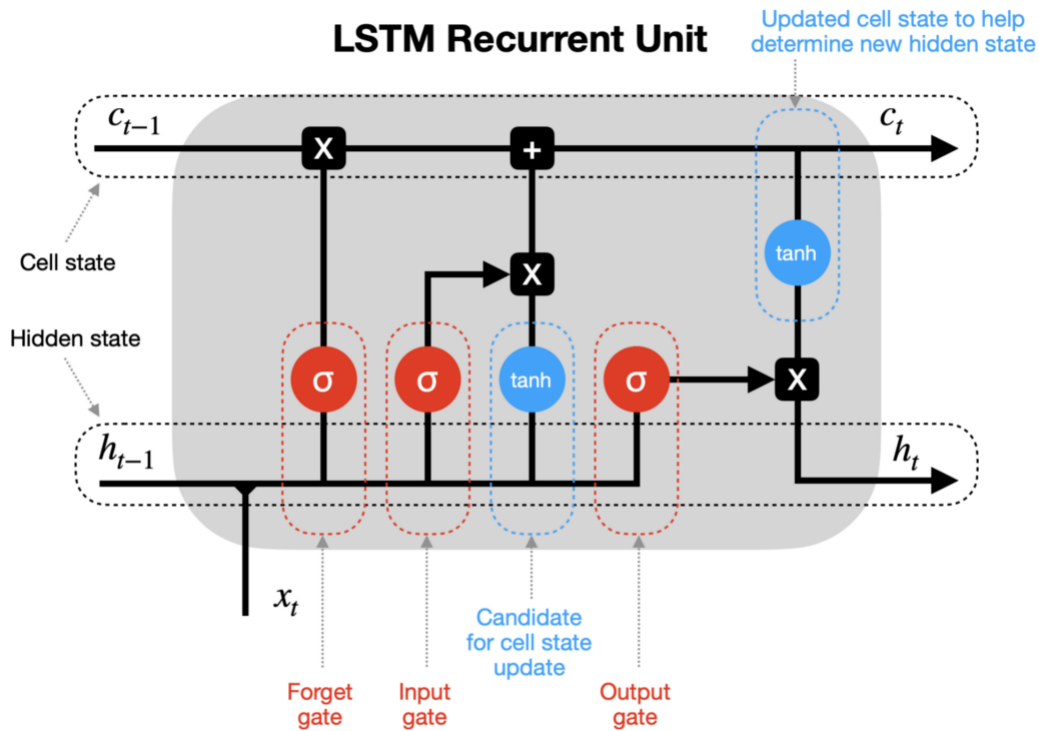


Figure 4: Représentation schématique d'un Memory Block d'un LSTM.

Grâce à cette architecture, les LSTM sont capables de mémoriser des informations importantes pendant de longues séquences et d'ignorer les informations non-pertinentes. Cela les rend particulièrement utiles pour des tâches nécessitant la modélisation de dépendances à long terme, telles que la reconnaissance vocale et la traduction automatique.

De nombreuses variantes ont vu, le jour, telles que :

- la Peephole connection[7] qui ajoute des connexions entre le Cell state et les différents gates.
- Long short-term memory Spiking Neural Network (LSNN)[2]

Malgré les différentes variantes existantes, il a été prouvé que le LSTM Vanilla n'est en réalité surpassé par aucune de ces variantes[30].

Gated Recurrent Units (GRU)

Présentés par Cho et al. en 2014 [4], les Gated Recurrent Units (GRU) sont un type de réseau de neurones récurrents (RNN) utilisé principalement dans le traitement des séquences, comme la modélisation des séries temporelles, le traitement du langage naturel (NLP) et la reconnaissance vocale.

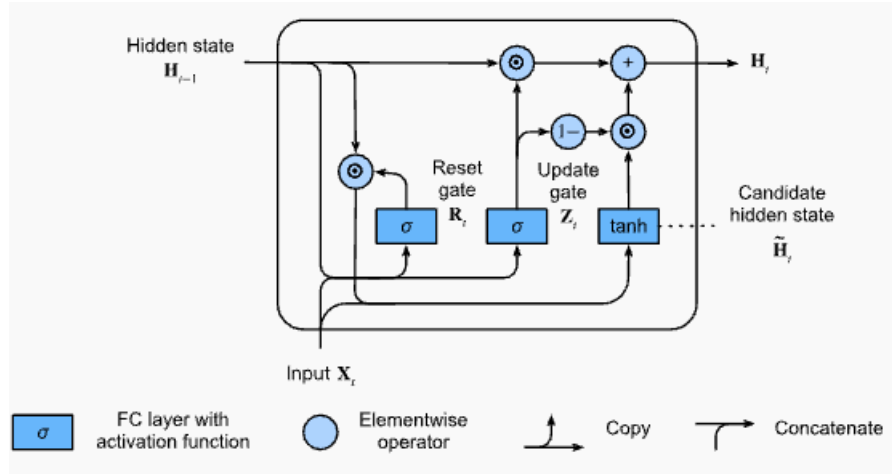


Figure 5: Représentation schématique d'un 'full gated unit' d'un GRU.

La structure du GRU peut rappeler celle du LSTM, on peut le voir notamment dans la nature additive de leur contenu, ce qui permet :

- de garder en mémoire des features importantes sur une longue série de récurrences.
- de faire la backpropagation de l'erreur sans apparition trop rapide de vanishing gradient.

On peut aussi observer la disparition du Cell State en faveur de l'unique présence d'un Hidden state. Ainsi, les deux gates prennent en compte la valeur de l'hidden state de l'unité précédente (time step précédent) et aussi l'input de l'unité actuelle (time step actuel). Une unité de GRU est composée de deux gates principales :

- Le **Reset Gate** qui va venir déterminer le pourcentage d'informations que nous voulons garder venant de la précédente unité résultant en un hidden state candidat.
- L'**Update Gate** va nous permettre de décider à quel point l'hidden state actuel sera une copie du précédent.

Les résultats de plusieurs tests empiriques indiquent que la structure GRU n'est pas spécialement plus performante que la structure LSTM Vanilla.[5]

III State of the Art

III.1 L'architecture des Transformers

Introduits en 2017 par Vaswani et al.[31], les Transformers ont originellement été pensés comme un outil de traduction. Cette nouvelle architecture permettant de surpasser les states-of-the-art dans de nombreux domaines, les Transformers ont été déclinés en de nombreuses variantes spécialisées pour divers domaines (X-formers)[19].

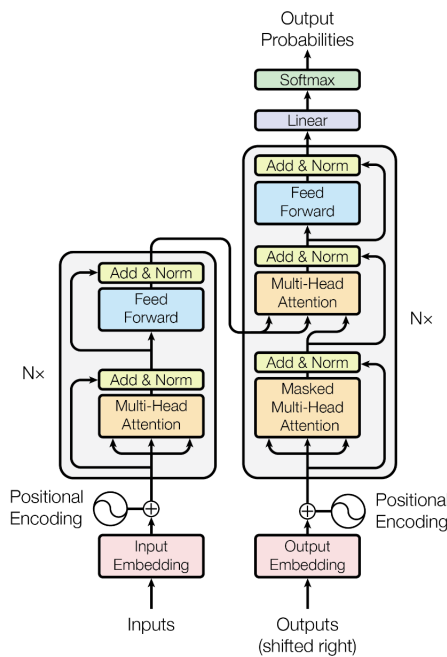


Figure 1: The Transformer - model architecture.

Figure 6: Représentation schématique de l'architecture Transformers.

Les modèles de Transformers se basent sur plusieurs points clés tels que les bases Encoder-Decoder, mais le point le plus important de leur architecture est sans doute *le mécanisme d'attention*[1].

Le but de l'Encoder va être de transformer la séquence d'entrée en une représentation interne qui capture les dépendances et les relations contextuelles entre les éléments de la séquence. Le Decoder, à l'aide de cette représentation interne, va générer une séquence en adéquation avec la séquence d'entrée.

L'Encoder

Input Embedding & Positional Encoding *L'input embedding* et le *positional encoding* sont deux étapes qui vont permettre de transformer le format de la séquence d'entrée en un format interprétable par l'architecture.

Ainsi, dans un premier temps, la séquence d'entrée va être tokenisée, puis un *embedding* de chaque token sera calculé en le transformant en un vecteur de taille d_{model} à l'aide

d'algorithmes d'embedding permettant de capturer pleinement la sémantique des mots dans le cas de NLP (comme word2vec par exemple[21]).



Figure 7: Schéma de l'embedding de la phrase 'your cat is a lovely cat' pour $d_{model} = 512$ [13]

Ensuite, le vecteur de *positional encoding* de taille d_{model} est produit sur la base des fonctions sinus et cosinus, permettant de capturer la position de chaque token dans la séquence.

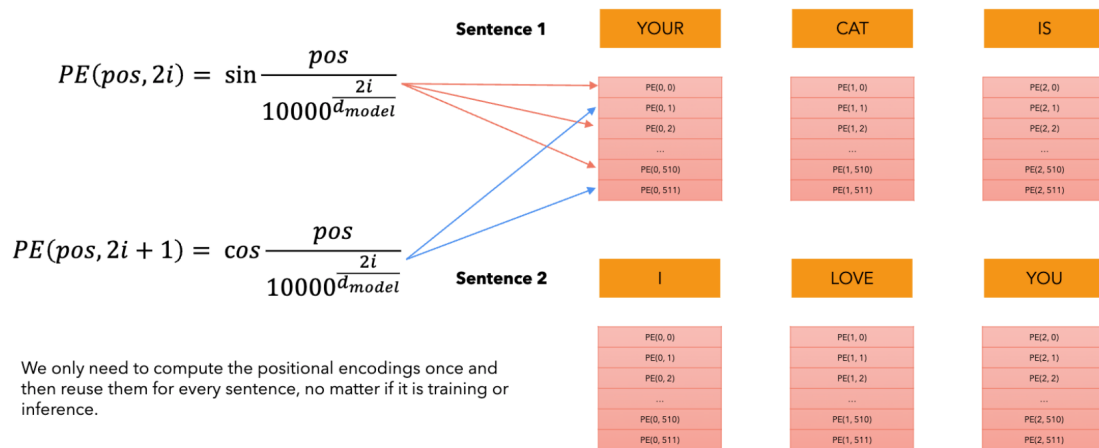


Figure 8: Schéma du positional encoding pour $d_{model} = 512$ [13]

Pour chaque token, le vecteur d'embedding sera additionné au vecteur de positional encoding résultant en un vecteur final, *input embedding*, de taille d_{model} qui capture la sémantique et la position de chaque token.

Module d'attention Le module d'attention est sans aucun doute le point central de toute l'architecture du Transformers Vanilla. Le but de ce module est, en se basant sur la sémantique et le positionnement des tokens d'entrée, de capturer les relations entre chaque token et d'extraire l'importance de chaque token dans la séquence.

L'innovation apportée par *Attention is All You Need*[31] se situe dans les *multi-head attention*. Dans un premier temps, voyons le mécanisme de *self-attention*. En considérant les matrices Q, K et V comme étant des copies de la matrice d'*input embedding* (et donc de dimension (d_{seq}, d_{model})).

La matrice d'attention de dimension (d_{seq}, d_{model}) est donc le résultat de

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

Au lieu de performer une unique fonction d'attention, il a été estimé bénéfique de projeter linéairement Q , K et V h fois avec différentes projections linéaires sur des dimension d_k , d_k et d_v respectivement. Ainsi, on obtient :

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

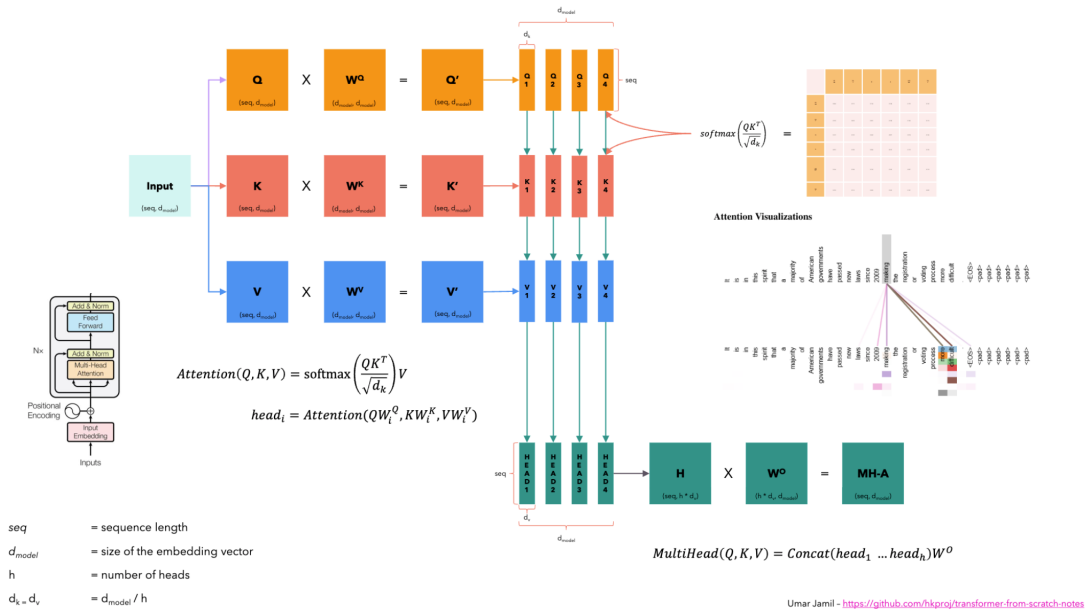


Figure 9: Schéma du module d'attention MultiHead pour $h = 4$. [13]

Layer Normalisation Pour finir, la matrice d'attention résultante va faire une Layer Normalisation qui vient normaliser chaque feature résultant d'un token par rapport à lui-même (différent de la batch normalisation qui vient normaliser chaque feature d'une observation par rapport au même feature des autres observations).

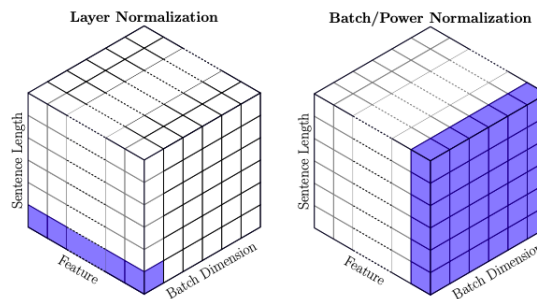


Figure 10: Layer norm vs Batch norm

Le Decoder

Le Decoder a une architecture sensiblement similaire à celle de l'Encoder, excepté l'ajout d'un module d'attention Multi-Head masqué, qui va appliquer de l'attention causale. Ainsi, la self-attention du Decoder est restreinte de sorte à ce que les paires key-values ne peuvent accéder qu'aux informations précédant le token.

La seconde Multi-Head Attention est en réalité un module de *cross attention* : la *query* sera la matrice provenant du module de *self-attention masqué*, la *key* et la *value* sont la sortie de l'Encoder.

III.2 LMs et LLMs

La différence entre LM et LLM réside principalement dans l'échelle des modèles. En effet, on passe d'une échelle de millions de paramètres pour les LMs à plusieurs milliards pour les LLMs (GPT-2 avec 124M de paramètres[26] vs GPT-3 avec 175 milliards[3]).

Un autre point intéressant dans la séparation des LMs et des LLMs se trouve dans l'apparition de capacités émergentes[32] dans les LLMs, où l'on observe les modèles faire preuve de certaines capacités qui n'étaient pas programmées ou conçues. On peut notamment observer :

- la compréhension de l'arithmétique
- la compréhension de mots désordonnés
- le décodage de l'alphabet phonétique international

III.3 Pré-entraînement des LLMs

La phase initiale de l'entraînement d'un LLM est le pré-entraînement. C'est durant cette phase que le modèle va devoir développer une compréhension du langage, contexte et différents types de connaissances. C'est une phase massivement lourde en calculs qui demande une quantité importante de données. Par exemple, Llama 2 70B a été entraîné sur environ 10TB de texte pour un coût de calcul estimé à environ 2 millions. Le modèle pré-entraîné résultant est appelé "checkpoint"[29].

Les pré-entraînements sont généralement classés en deux grandes catégories :

- **Autorégressif (GPTs)** : la prédiction de chaque token est basée sur les tokens précédents. Ainsi, le process se fait de manière itérative en ajoutant chaque nouveau token au contexte pour prédire le suivant.
- **Masqué (BERT, RoBERTa)** : ici, on va prédire les tokens de manière bidirectionnelle. Étant donné une séquence comme "j'aime [masqué][masqué] glacée", le modèle prédit les tokens masqués comme "manger de la crème".

III.4 Fine-Tuning LLMs

Suite au pré-training, qui permet au modèle d'acquérir des capacités et des connaissances générales, nous pouvons appliquer du fine-tuning dont le but est de permettre au modèle

de préciser et de renforcer ses compétences dans un domaine précis. Le fine-tuning se fait généralement sur un dataset bien plus petit et précis que celui du pré-training.

Il existe plusieurs types de fine-tuning dont notamment :

- **Ré-entraînement des poids (Fine tuning)**: on réeffectue un entraînement du modèle pré-entraîné sur un dataset spécifique à la tâche voulue. Le risque est de perdre des capacités de généralisation acquises lors du pré-entraînement dû au fait que nous modifions directement les poids déjà entraînés. De plus, c'est un processus lent et coûteux.
- **Prompt Tuning**: l'ajustement du modèle se fait via des prompts d'entrée qui vont venir orienter les réponses du modèle. Ceci est certes une méthode rapide et simple à mettre en place, mais le modèle va globalement rester dans des généralités.
- **Low-Rank Adaptation (LoRA)**: le LoRA est une technique de fine-tuning assez récente. Elle est rapide à mettre en place et se montre parmi les plus efficaces[12]. Ici, au lieu de ré-entraîner les poids déjà entraînés du modèle, une nouvelle couche de poids est ajoutée au modèle et seule cette dernière est entraînée sur le nouveau dataset, les couches pré-entraînées et les nouvelles couches sont ensuite fusionnées, permettant ainsi au modèle de conserver sa capacité de généralisation acquise lors du pré-training et d'y ajouter des connaissances plus pointues dans un domaine précis.

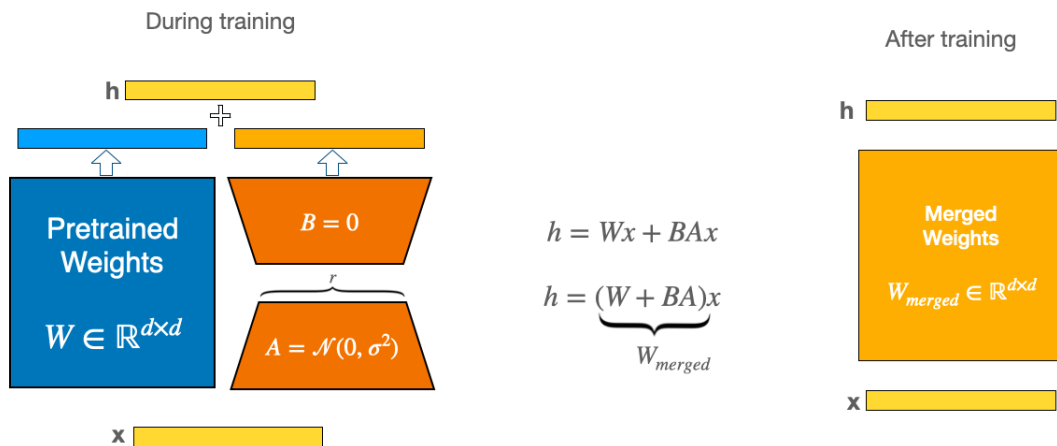


Figure 11: Low Rank Adaptation fine tuning

III.5 Méthodes d'évaluation

Dans le domaine du Machine Learning et plus largement lorsqu'on construit des modèles de prédiction ou de décision, il est nécessaire de pouvoir comparer plusieurs modèles entre eux. Que soit pour sur-performer le modèle d'autres personnes ou bien pour sélectionner un modèle parmi plusieurs que nous avons nous-même conçus. Dans cette optique, les modélisateurs ont proposé le concept d'une fonction d'évaluation qui attribuerait un score à un modèle.

Dans le cas d'un modèle de classification basique, la fonction d'évaluation est généralement l'accuracy ou le F1-score. Cependant, les LMs ne produisent pas de réponse aussi simple et concise que ces modèles, mais ils produisent des textes qui sont des données nettement plus complexes et donc plus difficiles à évaluer. Actuellement, les méthodes d'évaluation des LMs impliquent plusieurs mesures capables chacune d'évaluer une partie des capacités d'un LM. Voici quelques-unes des mesures les plus utilisées :

Perplexité

La capacité première d'un LM et donc celle que l'on veut évaluer en premier est de générer du texte syntaxiquement correct. La mesure de perplexité essaie d'évaluer cette partie du LM.

La perplexité ou perplexity en anglais, est une mesure qui évalue la capacité d'un modèle à générer un texte dont la composition en symbole est correcte basée sur un ensemble de texte servant de référence. Cela permet de mesurer à quel point le modèle est confus dans sa prédiction. La perplexité d'un texte généré est définie par la formule suivante :

$$PPL(X) = \exp \left(-\frac{1}{T} \sum_{t=1}^T \log (P_{\theta}(x_t|x_{<t})) \right)$$

Où X est une suite de symbole de longueur T et P_{θ} est un modèle probabiliste de symbole paramétré par θ . La perplexité est définie entre 0 et $+\infty$ avec une perplexité tendant vers 0 étant une phrase syntaxiquement correcte et une perplexité tendant vers $+\infty$ étant une phrase syntaxiquement incorrecte pour un modèle probabiliste donné.

Cette formule peut être exprimée de la manière suivante : la perplexité est égale à l'exponentielle de la moyenne des logarithmes de la probabilité de chaque symbole sachant les symboles précédents.

La perplexité d'un LM est calculée comme la moyenne des perplexités sur un sous-ensemble E_{eval} de texte afin de se rapprocher au plus proche des capacités réelles du modèle dans un temps limité.

$$PPL(LM) = \frac{1}{|E_{eval}|} \sum_{X \in E_{eval}} PPL(X)$$

La perplexité est l'une des mesures d'évaluation des LMs la plus simple, naturelle et compréhensible qui soit. Ce qui fait d'elle l'une des plus utilisées.

BLEU Score (BiLingual Evaluation Understudy)

Cette mesure évalue la capacité d'un modèle à traduire du texte. Il consiste à comparer les N-Grams composant la prédiction donnée par une entrée avec les N-Gram composant un label associé à l'entrée d'origine.

$$BLEUScore = BP \cdot \exp \left(\sum_{i=1}^N w_i \cdot \ln(\tilde{p}_i) \right)$$

$$\tilde{p}_i = \frac{\sum_{ng \in \{x \in NG_i(cand)\}} \min(|ng \in NG_i(cand)|, \max_j(|ng \in NG_i(ref_j)|))}{\sum_{ng \in \{x \in NG_i(cand)\}} |ng \in NG_i(cand)|} \quad BP = \exp \left(1 - \frac{r}{c} \right)$$

Où BP est la pénalité de brièveté, N le nombre de degré de N -Gram, w_i le poids pour chaque degré de N -Gram, \tilde{p}_i la précision modifiées des N -Gram de degré i , $\{x \in NG_i(cand)\}$ est la liste des N -Grams candidats distincts de degré i , $|ng \in NG_i(cand)|$ est le nombre de N -Gram ng dans la traduction candidate, $|ng \in NG_i(ref_j)|$ est le nombre de N -Gram ng dans le référentiel j , r est la longueur en mot de la traduction candidate et c est la longueur en mot moyenne des traductions de référence

Le BLEU score est calculé comme l'exponentiel de la moyenne pondérée des logarithmes de la précision des N -Grams présent pour chaque degré de N -Gram.

Quelques ajustements sont fait pour remédier à quelques biais :

- La précision de N -Gram est modifiée pour éviter qu'un modèle répète des N -Grams et améliore son score.
- Une pénalité pour les traduction courte qui pourraient manquer d'information lors de la traduction.

Le BLEU score est généralement compris entre 0 et 1, certain le représentent comme un pourcentage. Un BLEU Score de 0 est une très mauvaise traduction et un BLEU score de 1 une traduction identique à une traduction de référence.

N'existant pas d'unique traduction pour un texte donné, plusieurs traductions de références sont fournies dans le processus.

ROUGE Score (Recall-Oriented Understudy for Gisting Evaluation)

Cette mesure évalue la capacité d'un LM à résumer un texte. Comme pour Le BLEU Score, il utilise les N -Grams. Il correspond au F1-Score par rapport aux N -Grams présent dans le résumé de référence et le résumé candidat.

$$precision = \frac{|overlapping_{ngram}|}{|candidat_{ngram}|} \quad recall = \frac{|overlapping_{ngram}|}{|reference_{ngram}|}$$

$$F1Score = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

Où $|overlapping_{ngram}|$ est le nombre de n -gram apparaissant à la fois dans la référence et dans le résumé candidat, $|candidat_{ngram}|$ est le nombre de n -gram dans le résumé candidat, $|reference_{ngram}|$ est le nombre de n -gram dans le résumé de référence.

Il existe diverses variantes du ROUGE Score : les ROUGE-N, le ROUGE-L, le ROUGE-S. Le ROUGE-N désigne une famille de score où N est un nombre correspondant au degré utilisé pour le N-Gram dans la mesure (par exemple ROUGE-2). Le ROUGE-L se base sur la plus grande séquence de mot non-consécutif et ROUGE-S se base sur des skip-grams à la place de N-Grams.

SQuAD (The Stanford Question Answering Dataset)

SQuAd n'est pas à proprement parler une fonction d'évaluation, mais un dataset d'évaluation permettant d'évaluer la capacité d'un LM à retrouver des informations.

Le dataset SQuAD fournit un ensemble de tests plus ou moins complexe où chacun des tests contient un paragraphe d'information, une question et la réponse à cette question. Le but est que le LM répond à la question à l'aide du paragraphe qui lui aura été fourni auparavant et qu'il trouve la réponse. Il est intéressant de se demandé comment comparer la réponse du LM avec la réponse attendue pour savoir si le LM a bien répondu. L'un des mêmes serait de fine-tune le modèle pour qui ne renvoie que la réponse attendue.

Il existe deux versions du dataset SQuAD la version V1.1 et V2. La différence réside dans le fait que la V2 contient des questions dont il est impossible de trouver la réponse à l'aide du paragraphe informatif.

RACE (The ReAding Comprehension from Examination)

RACE n'est également pas à proprement parler une fonction d'évaluation, mais un dataset d'évaluation permettant d'évaluer la capacité d'un LM à retrouver des informations et à raisonner.

Le dataset RACE est basé sur un QCM du niveau mid/high school en Chine composé d'environ 100.000 questions associée à un paragraphe informatif. Les paragraphes sont plus grands que la normale et une partie des questions demande du raisonnement (l'information n'est pas explicite).

GLUE (General Language Understanding Evaluation)

GLUE n'est également pas à proprement parler une fonction d'évaluation, mais n'est également pas un dataset d'évaluation. GLUE est un framework d'évaluation de LLM. Il consiste à tester un modèle suivant différent critère d'évaluation portant sur la recherche d'information, l'analyse de sentiment, et autres. Pour cela, il est basé sur plusieurs datasets répondant à plusieurs tâches (11 au total). En plus, GLUE fournit un site où les résultats du LLM à l'évaluation peuvent être publiés dans un leaderboard visible par tous.

Une version plus sophistiquée existe désormais, elle est dénommée SuperGLUE

IV Notre application de génération

IV.1 Contexte de réalisation du projet

Notre objectif premier était de proposer une application de génération de texte, que ce soit dans le contexte de traduction ou de texte génératif. Les premières ressources sur lesquelles nous avons pu nous appuyer furent les travaux de Neubig[22] et ce repository (dépôt) GitHub, nommé "nanoGPT"[14] d'Andrej Karpathy, ancien directeur de l'intelligence artificielle chez Tesla et travaillant maintenant chez OpenAI où il se spécialise dans l'entraînement de modèle de langages. Le modèle proposé est un Transformer capable de reproduire l'architecture GPT-2 avec environ 124M de paramètres.

IV.2 Model.py

Le principal fichier de ce projet est Model.py. Il implémente un modèle de langage GPT en utilisant PyTorch. Ce modèle repose sur l'architecture Transformer, qui se compose de plusieurs couches de transformeurs empilées. Ces couches transforment les représentations internes du texte, permettant au modèle de comprendre et de générer du texte de manière cohérente.

Le fichier commence par importer les bibliothèques nécessaires, notamment PyTorch pour l'implémentation du modèle :

```
1 import math
2 import inspect
3 from dataclasses import dataclass
4
5 import torch
6 import torch.nn as nn
7 from torch.nn import functional as F
```

Ensuite, la classe LayerNorm est définie pour implémenter la normalisation de couches avec une option pour inclure ou exclure un biais. Cette classe initialise les paramètres de poids et de biais, et applique la normalisation de couche en utilisant la fonction Layernorm de PyTorch :

```
1 class LayerNorm(nn.Module):
2     def __init__(self, ndim, bias):
3         super().__init__()
4         self.weight = nn.Parameter(torch.ones(ndim))
5         self.bias = nn.Parameter(torch.zeros(ndim)) if bias else
        None
6
7     def forward(self, input):
8         return F.layer_norm(input, self.weight.shape, self.
        weight, self.bias, 1e-5)
```

La classe CausalSelfAttention implémente le mécanisme d'attention causale. Lors de l'initialisation, elle définit les projections de clé, requête et valeur (q, k, v), ainsi que les

paramètres de dropout pour la régularisation. Elle vérifie également la disponibilité de l'attention rapide (flash attention) :

```

1 class CausalSelfAttention(nn.Module):
2     def __init__(self, config):
3         super().__init__()
4         assert config.n_embd % config.n_head == 0
5         self.c_attn = nn.Linear(config.n_embd, 3 * config.n_embd
6         , bias=config.bias)
7         self.c_proj = nn.Linear(config.n_embd, config.n_embd,
8         bias=config.bias)
9         self.attn_dropout = nn.Dropout(config.dropout)
10        self.resid_dropout = nn.Dropout(config.dropout)
11        self.n_head = config.n_head
12        self.n_embd = config.n_embd
13        self.dropout = config.dropout
14        self.flash = hasattr(torch.nn.functional, '
15        scaled_dot_product_attention')
16        if not self.flash:
17            print("WARNING: using slow attention. Flash
18            Attention requires PyTorch >= 2.0")
19            self.register_buffer(...)

```

La méthode forward de CausalSelfAttention réalise le calcul de l'attention causale. Les projections q, k, v sont obtenues en divisant la sortie de 'c_attn' et remodelées pour séparer les têtes d'attention. Ensuite, l'attention causale est appliquée, en utilisant la version rapide si disponible. Sinon, le code applique un masque causal pour empêcher l'attention sur les futurs tokens, calcule les scores d'attention avec une fonction softmax, et applique un dropout :

```

1 def forward(self, x):
2     B, T, C = x.size()
3     q, k, v = self.c_attn(x).split(self.n_embd, dim=2)
4     q = q.view(B, T, self.n_head, C // self.n_head).transpose(1,
5     2)
6     k = k.view(B, T, self.n_head, C // self.n_head).transpose(1,
7     2)
8     v = v.view(B, T, self.n_head, C // self.n_head).transpose(1,
9     2)
10
11    if self.flash:
12        y = torch.nn.functional.scaled_dot_product_attention(q,
13        k, v, dropout_p=self.dropout if self.training else 0,
14        is_causal=True)
15    else:
16        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.
17        size(-1)))
18        att = att.masked_fill(self.bias[:, :, :T, :T] == 0,

```

```

float( '-inf' ))
13     att = torch.softmax(att, dim=-1)
14     att = self.attn_dropout(att)
15     y = att @ v
16
17     y = y.transpose(1, 2).contiguous().view(B, T, C)
18     y = self.resid_dropout(self.c_proj(y))
19     return y

```

IV.3 Train.py

Maintenant que notre modèle a été défini dans son fichier 'Model.py' il faut maintenant l'entraîner. C'est là qu'intervient 'Train.py'. Ce script peut fonctionner sur un seul GPU en mode débogage ou sur plusieurs GPU en utilisant la parallélisation des données distribuées (DDP). Le script commence par importer les bibliothèques nécessaires telles que os, time, math, pickle, numpy, torch, et d'autres modules PyTorch pour la gestion de la parallélisation et des modèles.

Les valeurs de configuration par défaut sont définies pour entraîner un modèle GPT-2 de 124 millions de paramètres sur le dataset OpenWebText. La configuration inclut des paramètres pour l'I/O, la journalisation, les données, le modèle, l'optimiseur AdamW, la décroissance du taux d'apprentissage et les paramètres DDP si utilisé.

```

1 # General parameters
2 out_dir = 'out'           # Output directory
3 eval_interval = 2000      # Evaluation interval in steps
4 log_interval = 1         # Logging interval in steps
5 eval_iters = 200         # Number of iterations for evaluation
6 eval_only = False        # Flag to run only evaluation
7 always_save_checkpoint = True # Flag to always save
                             checkpoint
8
9 # Data parameters
10 data_dir = 'data/openwebtext' # Data directory
11 gradient_accumulation_steps = 5 * 8 # Steps to accumulate
    gradients
12
13 # Model parameters
14 init_from = 'scratch'     # Initialization method
15 n_layer = 12              # Number of transformer layers
16 n_head = 12              # Number of attention heads
17 n_embd = 768              # Embedding size
18 dropout = 0.1            # Dropout rate
19 block_size = 1024         # Context length of the model
20
21 # Optimizer parameters
22 learning_rate = 6e-4      # Initial learning rate

```

```

23 max_iters = 600000          # Maximum number of iterations
24 weight_decay = 1e-2         # Weight decay for L2 regularization
25 beta1 = 0.9                 # Beta1 parameter for Adam optimizer
26 beta2 = 0.95               # Beta2 parameter for Adam optimizer
27 grad_clip = 1.0            # Gradient clipping value
28
29 # Learning rate decay parameters
30 decay_lr = True             # Flag to decay learning rate
31 warmup_iters = 2000         # Number of warmup iterations
32 lr_decay_iters = 600000     # Number of iterations to decay learning
    rate
33 min_lr = 6e-5              # Minimum learning rate
34
35 # Device parameters
36 device = 'cuda'             # Device to use ('cuda' or 'cpu')
37 dtype = 'bfloat16' if torch.cuda.is_available() and torch.cuda.
    is_bf16_supported() else 'float16' # Data type

```

Le modèle est initialisé soit à partir de zéro, soit en reprenant à partir d'un checkpoint, soit en utilisant les poids pré-entraînés de GPT-2. La configuration du modèle est également mise à jour en fonction des arguments fournis :

```

1 if init_from == 'scratch':
2     if meta_vocab_size is None:
3         model_args['vocab_size'] = 50304
4         gptconf = GPTConfig(**model_args)
5         model = GPT(gptconf)
6 elif init_from == 'resume':
7     ckpt_path = os.path.join(out_dir, 'ckpt.pt')
8     checkpoint = torch.load(ckpt_path, map_location=device)
9     checkpoint_model_args = checkpoint['model_args']
10    for k in ['n_layer', 'n_head', 'n_embd', 'block_size', 'bias
    ', 'vocab_size']:
11        model_args[k] = checkpoint_model_args[k]
12        gptconf = GPTConfig(**model_args)
13        model = GPT(gptconf)
14        state_dict = checkpoint['model']
15        unwanted_prefix = '_orig_mod.'
16        for k,v in list(state_dict.items()):
17            if k.startswith(unwanted_prefix):
18                state_dict[k[len(unwanted_prefix):]] = state_dict.
    pop(k)
19        model.load_state_dict(state_dict)
20        iter_num = checkpoint['iter_num']
21        best_val_loss = checkpoint['best_val_loss']
22 elif init_from.startswith('gpt2'):
23     model = GPT.from_pretrained(init_from, override_args)

```

```

24     for k in ['n_layer', 'n_head', 'n_embd', 'block_size', 'bias
25             ', 'vocab_size']:
        model_args[k] = getattr(model.config, k)

```

La boucle d'entraînement du script configure et optimise le modèle GPT de manière efficace en utilisant des techniques avancées comme l'accumulation de gradients et la découpe des gradients, tout en s'adaptant à des environnements multi-GPU grâce à DDP. Initialement, le taux d'apprentissage est déterminé et mis à jour en fonction du numéro d'itération. À chaque intervalle d'évaluation défini, le modèle calcule les pertes d'entraînement et de validation, enregistre les résultats et sauvegarde des checkpoints si nécessaire. Durant chaque itération principale, plusieurs étapes de micro-entraînement sont effectuées pour accumuler les gradients sans dépasser la mémoire GPU disponible. Après accumulation, les gradients sont éventuellement réduits pour éviter des valeurs trop grandes avant d'être appliqués à l'optimiseur pour mettre à jour les paramètres du modèle. Le temps écoulé et les performances sont enregistrés périodiquement, avec une mesure de l'utilisation de la mémoire (MFU). La boucle continue jusqu'à atteindre le nombre maximal d'itérations.

```

1  while True:
2      lr = get_lr(iter_num) if decay_lr else learning_rate
3      for param_group in optimizer.param_groups:
4          param_group['lr'] = lr
5
6      if iter_num % eval_interval == 0 and master_process:
7          losses = estimate_loss()
8          print(f"step {iter_num}: train loss {losses['train']:.4f}
9              }, val loss {losses['val']:.4f}")
10         if wandb.log:
11             wandb.log({
12                 "iter": iter_num,
13                 "train/loss": losses['train'],
14                 "val/loss": losses['val'],
15                 "lr": lr,
16                 "mfu": running_mfu*100,
17             })
18         if losses['val'] < best_val_loss or
19         always_save_checkpoint:
20             best_val_loss = losses['val']
21             if iter_num > 0:
22                 checkpoint = {
23                     'model': raw_model.state_dict(),
24                     'optimizer': optimizer.state_dict(),
25                     'model_args': model_args,
26                     'iter_num': iter_num,
27                     'best_val_loss': best_val_loss,
28                     'config': config,
29                 }

```



```

28         print(f"saving checkpoint to {out_dir}")
29         torch.save(checkpoint, os.path.join(out_dir, '
ckpt.pt'))
30     if iter_num == 0 and eval_only:
31         break
32
33     for micro_step in range(gradient_accumulation_steps):
34         if ddp:
35             model.require_backward_grad_sync = (micro_step ==
gradient_accumulation_steps - 1)
36             with ctx:
37                 logits, loss = model(X, Y)
38                 loss = loss / gradient_accumulation_steps
39                 X, Y = get_batch('train')
40                 scaler.scale(loss).backward()
41             if grad_clip != 0.0:
42                 scaler.unscale_(optimizer)
43                 torch.nn.utils.clip_grad_norm_(model.parameters(),
grad_clip)
44             scaler.step(optimizer)
45             scaler.update()
46             optimizer.zero_grad(set_to_none=True)
47
48             t1 = time.time()
49             dt = t1 - t0
50             t0 = t1
51             if iter_num % log_interval == 0 and master_process:
52                 lossf = loss.item() * gradient_accumulation_steps
53                 if local_iter_num >= 5:
54                     mfu = raw_model.estimate_mfu(batch_size *
gradient_accumulation_steps, dt)
55                     running_mfu = mfu if running_mfu == -1.0 else 0.9*
running_mfu + 0.1*mfu
56                     print(f"iter {iter_num}: loss {lossf:.4f}, time {dt
*1000:.2f}ms, mfu {running_mfu*100:.2f}%")
57                 iter_num += 1
58                 local_iter_num += 1

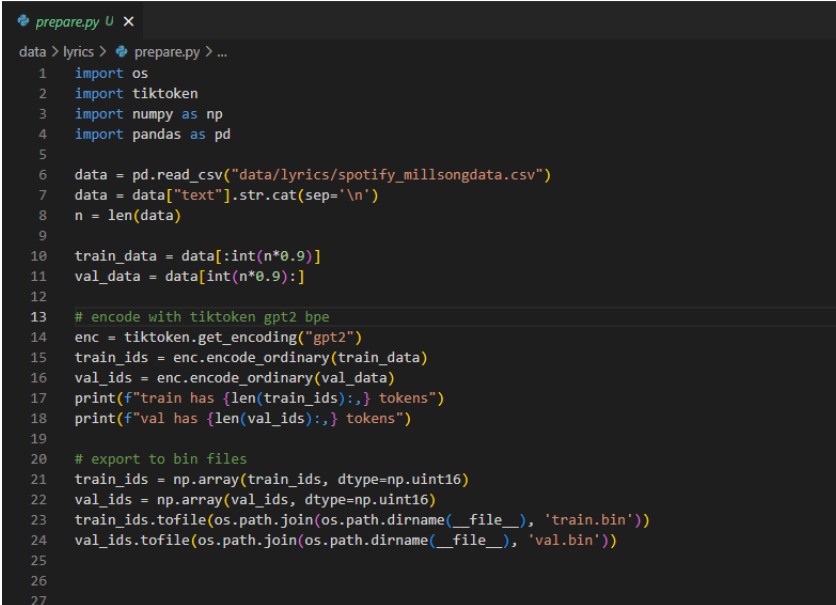
```

Nous avons maintenant tous les éléments principaux pour créer nos premiers modèles.

IV.4 Premier modèle

Dans un premier temps, nous avons entraîné ce dernier sur le dataset TinyShakespeare, un dataset d'environ 1.34MB reprenant l'oeuvre de Shakespeare. Nous avons donc préparé les données en récupérant le dataset, puis en l'encodant de manière "naïve" mot par mot, ou avec l'encoding tiktoken de GPT-2[23]. Nous divisons ensuite le dataset en training et validation set et nous exportons ensuite les fichiers sous le format .bin pour des soucis

d'efficacité.



```

1  import os
2  import tiktoken
3  import numpy as np
4  import pandas as pd
5
6  data = pd.read_csv("data/lyrics/spotify_millsongdata.csv")
7  data = data["text"].str.cat(sep='\n')
8  n = len(data)
9
10 train_data = data[:int(n*0.9)]
11 val_data = data[int(n*0.9):]
12
13 # encode with tiktoken gpt2 bpe
14 enc = tiktoken.get_encoding("gpt2")
15 train_ids = enc.encode_ordinary(train_data)
16 val_ids = enc.encode_ordinary(val_data)
17 print(f"train has {len(train_ids):,} tokens")
18 print(f"val has {len(val_ids):,} tokens")
19
20 # export to bin files
21 train_ids = np.array(train_ids, dtype=np.uint16)
22 val_ids = np.array(val_ids, dtype=np.uint16)
23 train_ids.tofile(os.path.join(os.path.dirname(__file__), 'train.bin'))
24 val_ids.tofile(os.path.join(os.path.dirname(__file__), 'val.bin'))
25
26
27

```

Figure 12: Préparation et encoding du dataset TinyShakespeare

Ensuite, le fichier de configuration d'entraînement est créé : il définit la liste et valeurs des hyper-paramètres utilisés en paramètre d'entrée pour l'entraînement. Ainsi, si nous entraînons notre modèle en partant d'une base nulle, nous pouvons mettre un nombre d'itération et un coefficient d'apprentissage élevés. À l'inverse, lors de l'étape de fine-tuning, ces hyper-paramètres seront plutôt faibles. Avec ces paramètres configurés, nous pouvons lancer l'entraînement. Suite à cela, nous pouvons générer du texte à l'aide de notre modèle entraîné. Ainsi, nous obtenons ce premier résultat à partir du modèle entraîné seulement sur TinyShakespeare :

```

1  # TinyShakespeare
2  iter 5000: loss 0.8221, time 9174.95ms, mfu 31.35%
3  #####
4  YORK:
5  I trunk with that you have set boar to take rid my cold.
6
7  Messenger:
8  Nurse, I do as delay ase you.
9
10 DUCHESS OF YORK:
11 What, ho! to that I commit more than mirth,
12 In mine arms that G Edward have I have drawn'd lord him;
13 For hung my heart that never let my loving kneel,
14 In gold's good world that more noon brother Tybalt's doubt,
15 The shire of Clifford, the excuse thrust for treat.
16 Why, there is Montague for Death hath razed been tood
17 For all all his graciousing strange tears,--to change me
18 An
19 -----

```

Comme nous pouvons le voir, le modèle arrive à capturer certains trains de l'écriture de l'auteur, mais les phrases ne font que très peu de sens et certaines hallucinations se remarquent au travers de mots inventés. Cette pauvre performance est due au manque

d'entraînement général du modèle, qui pour l'instant, n'est sûrement pas en capacité de pleinement capturer la sémantique des mots.

Nous avons donc décidé de prendre une nouvelle approche en nous basant sur un modèle pré-entraîné sur la base d'OpenWebText[24], un dataset open-source inspiré du dataset WebText d'OpenAI utilisé pour entraîner GPT, puis de le fine-tuner sur TinyShakespeare. Ainsi, nous aurons toutes les capacités de sémantique affinées au cas de Shakespeare. Le choix d'utiliser un modèle pré-entraîné s'est fait, car, malgré un PC équipé d'une carte graphique RTX3070 que nous avons monté pour l'occasion, l'entraînement total du modèle nous aurait pris des jours voire semaine de calcul, ce qui n'était pas envisageable avec nos moyens. Nous avons donc décidé de nous rabattre sur le modèle pré-entraîné (checkpoint) cité ci-dessus.

Ainsi, avec cette nouvelle base, nous obtenons les résultats suivants :

```

1 #TinyShakespeare (chkpt GPT2)
2 iter 20: loss 2.6474, time 37429.17ms, mfu 0.27%
3 #####
4 DUKE VINCENTIO:
5 I see how they hurt our pride.
6
7 LADY DUKE VINCENTIO:
8 All death, and all a curse ;
9 None but our birth can say,
10 How they may offend us.
11
12 DUKE VINCENTIO:
13 Come, bear us forward this way.
14
15 LADY DUKE VINCENTIO:
16 Fair lords, I am a soldier of fortune.
17
18 CRESCENTIA:
19 This day in our king's time,
20 Do I see very nearly'd on his cap,
21 He which such eyes saw in day-light,
22 Shall I on my self seem like?
23 Here comes the gallows of Nottingham!
24 Here walks the sword, which so many
25 With spears and shields have run down,
26 Weeping and wailing, their loud exclamation,
27 But do I here speak like?
28 -----

```

Comme nous pouvons l'observer, le résultat présente plus de sens et surtout moins d'hallucinations. Suite à cela, nous avons ensuite fine-tuné notre checkpoint sur le dataset Spotify Million Song d'environ 100Mo. En choisissant comme point de génération le mot 'Love', nous obtenons le résultat suivant :

```

1 D part : "Love"
2 #####
3 LYRICS :
4
5 Love is alive beyond the grave
6 Make no mistake, you hear me, oh no i don't
7 Did no one love me long enough

```

```
8 hat I never did see it happen ,
9
10 but I know you exist , oh no
11 You believe it is true ,
12 Oh no Was believed by you
13 I didn't want it , yes
14 I want it , yes
15
16 So bad I know I's not coming true
17 Let it go this time , oh no
18 I want to tell the truth now . I wanna tell , oh no , oh no . Oh no .
```

IV.5 Pistes d'amélioration

Comme nous pouvons facilement le voir, nous faisons face à un problème de répétition excessive, s'offre à nous plusieurs pistes :

- faire différents tests avec des valeurs changeantes pour le paramètre de température
- Ayant déjà mis en place le top-k sampling qui est un seuil limitant le nombre de candidats, essayer d'ajouter le top-p sampling
- ajouter une pénalité lors des répétitions du modèle, divisant la probabilité actuelle par un facteur de pénalité.

Étant donné notre faible puissance de calcul, il n'était pas facile pour nous de réaliser des benchmarks sur différentes configurations demandant de réentraîner le modèle à chaque fois. Mais nous aimerions réaliser un benchmark sur:

- l'encoding en testant différents types d'encodeur : encoding naïf, tiktoken (OpenAI), SentencePiece(Google)[17]
- l'embedding en testant le Rotary Embedding[28] et l'Alibi Embedding[25].
- en testant différentes valeurs d'hyper-paramètres.

Nous aimerions aussi proposer une simple interface utilisateur pour plus de facilité d'utilisation en utilisant par exemple le package Gradio[8].

V Conclusion

En conclusion, ce projet de fin d'études a permis d'explorer de manière approfondie les modèles de langage et les différentes techniques de génération de texte. Nous avons mis en place et évalué divers modèles, notamment ceux basés sur les n-grams et les réseaux de neurones, en particulier les Transformers. Notre travail a inclus non seulement la création et le pré-entraînement de modèles de langage, mais également leur fine-tuning sur des ensembles de données spécifiques pour améliorer leurs performances.

Malgré les limitations de puissance de calcul auxquelles nous avons fait face, nous avons réussi à obtenir des résultats significatifs. Les pistes d'amélioration identifiées, telles

que l'ajustement des paramètres de température, l'implémentation du top-p sampling, et l'ajout de pénalités pour les répétitions excessives, offrent des voies prometteuses pour des optimisations futures. De plus, la proposition de créer une interface utilisateur simple, par exemple en utilisant le package Gradio, pourrait rendre ces outils plus accessibles et utilisables dans des contextes pratiques.

En somme, ce projet a non seulement consolidé nos compétences techniques en matière de modèles de langage, mais a également ouvert la voie à de futures recherches et développements dans ce domaine en constante évolution.

Bibliography

- [1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: <https://arxiv.org/abs/1409.0473>.
- [2] Guillaume Bellec et al. *Long short-term memory and learning-to-learn in networks of spiking neurons*. 2018. arXiv: 1803.09574 [cs.NE]. URL: <https://arxiv.org/abs/1803.09574>.
- [3] Tom Brown et al. “Language models are few-shot learners”. In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.
- [4] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL]. URL: <https://arxiv.org/abs/1406.1078>.
- [5] Junyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE]. URL: <https://arxiv.org/abs/1412.3555>.
- [6] François Fleuret. “The little book of deep learning”. In: *A lovely concise introduction* (2023), p. 297.
- [7] F.A. Gers and J. Schmidhuber. “Recurrent nets that time and count”. In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*. Vol. 3. 2000, 189–194 vol.3. DOI: 10.1109/IJCNN.2000.861302.
- [8] Gradio. <https://github.com/gradio-app/gradio>.
- [9] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.
- [10] Sepp Hochreiter et al. *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*. 2001.
- [11] John J Hopfield. “Neural networks and physical systems with emergent collective computational abilities.” In: *Proceedings of the national academy of sciences* 79.8 (1982), pp. 2554–2558.
- [12] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: <https://arxiv.org/abs/2106.09685>.
- [13] Umar Jamil. <https://github.com/hkproj/transformer-from-scratch-notes>. 2023.
- [14] Andrej Karpathy. <https://github.com/karpathy/nanoGPT>. 2022.

- [15] John D. Kelleher and Brendan Tierney. *Data Science*. MIT Press Essential Knowledge Series. Cambridge, MA: MIT Press, 2018. ISBN: 978-0-262-53543-4.
- [16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).
- [17] Taku Kudo and John Richardson. *SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing*. 2018. arXiv: 1808.06226 [cs.CL]. URL: <https://arxiv.org/abs/1808.06226>.
- [18] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.
- [19] Tianyang Lin et al. *A Survey of Transformers*. 2021. arXiv: 2106.04554 [cs.LG]. URL: <https://arxiv.org/abs/2106.04554>.
- [20] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.
- [21] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL]. URL: <https://arxiv.org/abs/1301.3781>.
- [22] Graham Neubig. “Neural machine translation and sequence-to-sequence models: A tutorial”. In: *arXiv preprint arXiv:1703.01619* (2017).
- [23] OpenAI. <https://github.com/openai/tiktoken>.
- [24] Joshua Peterson. <https://github.com/jcpeterson/openwebtext>. 2022.
- [25] Ofir Press, Noah A. Smith, and Mike Lewis. *Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation*. 2022. arXiv: 2108.12409 [cs.CL]. URL: <https://arxiv.org/abs/2108.12409>.
- [26] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.
- [27] Alex Sherstinsky. “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network”. In: *Physica D: Nonlinear Phenomena* 404 (Mar. 2020), p. 132306. ISSN: 0167-2789. DOI: 10.1016/j.physd.2019.132306. URL: <http://dx.doi.org/10.1016/j.physd.2019.132306>.
- [28] Jianlin Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: <https://arxiv.org/abs/2104.09864>.
- [29] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: <https://arxiv.org/abs/2307.09288>.
- [30] Greg Van Houdt, Carlos Mosquera, and Gonzalo Nápoles. “A review on the long short-term memory model”. In: *Artificial Intelligence Review* 53.8 (2020), pp. 5929–5955.
- [31] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).
- [32] Jason Wei et al. *Emergent Abilities of Large Language Models*. 2022. arXiv: 2206.07682 [cs.CL]. URL: <https://arxiv.org/abs/2206.07682>.

- [33] Joseph Weizenbaum. “ELIZA—a computer program for the study of natural language communication between man and machine”. In: *Communications of the ACM* 9.1 (1966), pp. 36–45.
- [34] Paul J Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560.
- [35] Albert Ziegler et al. “Productivity assessment of neural code completion”. In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. MAPS 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 21–29. ISBN: 9781450392730. DOI: 10 . 1145 / 3520312 . 3534864. URL: <https://doi.org/10.1145/3520312.3534864>.