

# Les modèles de langue neuronaux

Projet de statistique appliquée

Etienne Boisseau  
Olivier Dulcy  
Christos Katsoulakis  
Eric Lavergne



Sous la direction de Benjamin Muller, INRIA  
Année 2019-2020

# Sommaire

<b>Introduction</b>	<b>2</b>
<b>1 Théorie probabiliste des modèles de langue</b>	<b>2</b>
1.1 Cas général . . . . .	2
1.1.1 Construction de l'espace probabilisé . . . . .	2
1.1.2 Construction d'un modèle de langue par probabilités conditionnelles . . . . .	3
1.2 Modèle $n$ -gram . . . . .	3
1.3 Modèle Neural Network . . . . .	4
1.4 Génération d'échantillons de texte . . . . .	5
1.4.1 Méthode gloutonne . . . . .	5
1.4.2 Méthode Beam Search . . . . .	5
1.4.3 Méthode de l'échantillonnage . . . . .	6
<b>2 Transformer</b>	<b>7</b>
2.1 Vue globale . . . . .	7
2.2 Entrée du Transformer . . . . .	8
2.2.1 Word Embedding . . . . .	8
2.2.2 Positional Encoding . . . . .	9
2.3 Partie Encoder . . . . .	9
2.3.1 Multi-Head Attention . . . . .	9
2.3.2 Sortie du Transformer . . . . .	10
2.3.3 Exemple de valeurs pour les hyperparamètres . . . . .	10
<b>3 Expériences</b>	<b>11</b>
3.1 Jeu de données . . . . .	11
3.2 Implémentation . . . . .	11
3.3 Mesure de performance : la Perplexité . . . . .	11
3.4 Résultats avec TensorFlow (Subword Encoding) . . . . .	13
3.4.1 N-Gram . . . . .	13
3.4.2 Transformer . . . . .	14
3.5 Résultats PyTorch (Word Encoding) . . . . .	16
3.5.1 N-Gram . . . . .	16
3.5.2 Transformer . . . . .	17
3.6 Résumé des résultats . . . . .	18
<b>4 Conclusion</b>	<b>19</b>
<b>Références</b>	<b>19</b>

# Introduction

La modélisation de la langue est un champ d'études ancien dont l'objectif est d'analyser et de reproduire le langage naturel. Un modèle de langue performant est capable par exemple de générer des textes réalistes ou de deviner une suite plausible à un texte. Durant la dernière décennie, ce champ d'études a été bouleversé par le succès des modèles de langue neuronaux. Leur efficacité sans précédent a fait passer les modèles de langue d'une construction théorique à un ensemble de méthodes largement utilisées dans l'industrie.

On pourrait citer quelques étapes clés de cette progression : De l'article très influent d'Andrey Karpathy en 2015 au développement par OpenAI en 2019 d'un modèle révolutionnaire par sa taille et ses performances : GPT-2 [3]. Très récemment, l'article présentant le modèle GPT-3 (dont le nombre de paramètres dépasse d'un facteur 117 celui de GPT-2) [1] a démontré les possibilités et les limites des modèles actuels, qui bénéficient de telles augmentations d'échelle mais sont à rendement marginal décroissant.

Sous la direction de Benjamin Muller (INRIA), nous avons cherché à étudier ces modèles, et particulièrement la sous-classe qui aujourd'hui constitue l'état de l'art (et à laquelle appartiennent GPT-2 et GPT-3) : les Transformers.

En section 1, nous établissons une théorie probabiliste des modèles de langue, s'appliquant aussi bien à des modèles relativement anciens (modèles  $n$ -gram) qu'aux Transformers. Nous utilisons cette théorie pour exprimer formellement le fonctionnement des modèles  $n$ -gram et modèles neuronaux, ainsi que les méthodes d'échantillonnage basées sur ces modèles.

En section 2, nous précisons le modèle du Transformer sur lequel se concentre notre étude. Nous donnons une vue globale de son architecture avant de décrire précisément ses composantes.

Enfin, en section 3, nous décrivons les expériences que nous avons menées autour des modèles  $n$ -gram et Transformers. Nous comparons les résultats obtenus par ces deux algorithmes sous différentes conditions et hyperparamètres, aussi bien qualitativement que quantitativement.

## 1 Théorie probabiliste des modèles de langue

### 1.1 Cas général

#### 1.1.1 Construction de l'espace probabilisé

**Notation :** On note  $A \mapsto |A|$  la fonction qui associe à un ensemble  $A$  son cardinal.

**Définition 1.1** On appelle *vocabulaire* un ensemble fini quelconque, noté  $V = \{s_1, \dots, s_{|V|}\}$ . Les  $s_i$  sont appelés *symboles*. On note  $\varepsilon$  le symbole vide qui n'appartient pas à  $V$ .

□ Exemple de symboles :

- Un caractère
- Un mot
- Un bit

□ Exemple de vocabulaire :

- Ensemble des mots de la langue française
- Ensemble des caractères unicode

**Définition 1.2** Un texte  $T$  est un élément de  $V^L$ , où  $L \in \mathbb{N}^*$ .

On va définir une probabilité sur l'ensemble des textes. Définissons notre espace de probabilité.

**Définition 1.3** On appelle l'ensemble des textes  $\Omega = \bigcup_{L=1}^{+\infty} V^L$ . On note  $\mathcal{A} = \sigma(\{T\} \mid T \in \Omega)$ , une tribu sur  $\Omega$ .

On note  $L : T \in \Omega \mapsto |T|$  la variable aléatoire qui associe à un texte sa longueur. On définit les  $(X_n)_{n \in \mathbb{N}^*}$  comme :

$$\forall i \in \mathbb{N}^*, X_i(T) = \begin{cases} i\text{-ème symbole de } T & \text{si } i \leq L(T) \\ \varepsilon & \text{si } i > L(T) \end{cases}$$

On suppose qu'il existe une probabilité  $\mathbb{P}$  sur l'espace probabilisable  $(\Omega, \mathcal{A})$ . On dispose d'un échantillon de textes distribué selon la mesure  $\mathbb{P}$  et on cherche à estimer  $\mathbb{P}$  par une mesure de probabilité  $\hat{\mathbb{P}}$ .

**Définition 1.4** On appelle  $\hat{\mathbb{P}}$  un modèle de langue.

En raison de la nature séquentielle du langage, on le construit en pratique en conditionnant sur les mots précédents du texte.

### 1.1.2 Construction d'un modèle de langue par probabilités conditionnelles

Soit un texte  $T = s_1 \dots s_L \in V^L$ , où  $L \in \mathbb{N}^*$ . La probabilité d'observer  $T$  s'écrit :

$$\begin{aligned} \mathbb{P}(T) &= \mathbb{P} \left( \bigcap_{i=1}^L X_i = s_i \cap \bigcap_{i=L+1}^{+\infty} X_i = \varepsilon \right) \\ &= \mathbb{P} \left( \bigcap_{i=1}^L X_i = s_i \cap X_{L+1} = \varepsilon \right) \text{ par construction des } X_i \\ &= \mathbb{P}(X_1 = s_1 \cap \dots \cap X_L = s_L \cap X_{L+1} = \varepsilon) \\ &= \mathbb{P}(X_{L+1} = \varepsilon | X_1 = s_1, \dots, X_L = s_L) \mathbb{P}(X_1 = s_1, \dots, X_L = s_L) \\ &= \mathbb{P}(X_{L+1} = \varepsilon | X_1 = s_1, \dots, X_L = s_L) \mathbb{P}(X_L = s_L | X_1 = s_1, \dots, X_{L-1} = s_{L-1}) \times \\ &\quad \mathbb{P}(X_1 = s_1, \dots, X_{L-1} = s_{L-1}) \\ &= \prod_{i=1}^{L+1} \mathbb{P}(X_i = s_i | X_1 = s_1, \dots, X_{i-1} = s_{i-1}) \text{ en posant } s_{L+1} = \varepsilon \end{aligned}$$

Nous serons amenés à effectuer des approximations dans les calculs pour estimer ces probabilités. Ces différentes estimations conduisent à la définition de différents modèles de langues neuronaux. Nous distinguons les modèles de langue suivants :

- les modèles  $n$ -grams
- les modèles Neural Network (NN)

## 1.2 Modèle $n$ -gram

Dans un modèle  $n$ -gram, nous faisons l'hypothèse simplificatrice que la probabilité d'apparition du mot  $s_i$  ne dépend que des  $n - 1$  prédécesseurs. Ainsi,

$$\mathbb{P}(X_i = s_i | X_1 = s_1, \dots, X_{i-1} = s_{i-1}) = \mathbb{P}(X_i = s_i | X_{i-(n-1)} = s_{i-(n-1)}, \dots, X_{i-1} = s_{i-1})$$

Nous estimons ces probabilités sur un corpus de textes et nous supposons que le corpus de textes reflète la langue dans l'absolu, ce qui sera le cas si nous disposons d'un très grand corpus de textes. Il s'agit là d'une approche statistique.

Etant donné que nous travaillons sur un corpus de textes fini, nous utilisons naturellement pour probabilité la mesure de comptage. Ainsi, le calcul de probabilité conditionnelle devient :

$$\mathbb{P}(X_i = s_i | X_{i-(n-1)} = s_{i-(n-1)}, \dots, X_{i-1} = s_{i-1}) = \frac{|X_{i-(n-1)} = s_{i-(n-1)}, \dots, X_{i-1} = s_{i-1}, X_i = s_i|}{|X_{i-(n-1)} = s_{i-(n-1)}, \dots, X_{i-1} = s_{i-1}|}$$

$n = 1$	Modèle unigram	$\mathbb{P}(T) = \prod_{i=1}^{L+1} \mathbb{P}(X_i = s_i)$
$n = 2$	Modèle bigram	$\mathbb{P}(T) = \prod_{i=1}^{L+1} \mathbb{P}(X_i = s_i   X_{i-1} = s_{i-1})$
$n = 3$	Modèle trigram	$\mathbb{P}(T) = \prod_{i=1}^{L+1} \mathbb{P}(X_i = s_i   X_{i-2} = s_{i-2}, X_{i-1} = s_{i-1})$
$n > 3$	Modèle $n$ -gram	$\mathbb{P}(T) = \prod_{i=1}^{L+1} \mathbb{P}(X_i = s_i   X_{i-(n-1)} = s_{i-(n-1)}, \dots, X_{i-1} = s_{i-1})$

TABLE 1 – Différents modèles  $n$ -gram, selon les valeurs de  $n$ 

**Limitations :** Etant donné que nous travaillons sur un corpus fini, nous avons une combinaison de mots finis. Il est possible qu'un mot n'apparaisse pas dans le modèle. Sa probabilité d'apparition est donc nulle :  $\mathbb{P}(X_k = s_k) = 0$ . On parle de *sparcité*. Cette probabilité nulle pose problème : toute séquence de mots qui n'apparaît pas dans le corpus a une probabilité égale à 0 d'apparaître. Notre modèle reconnaît donc uniquement des séquences connues.

Pour pallier ce problème et pouvoir généraliser à des séquences de mots non connues, nous pouvons effectuer un « lissage », qui consiste à attribuer une probabilité non nulle pour les mots n'apparaissant jamais dans le corpus.

### 1.3 Modèle Neural Network

Une approche permettant d'opérer un lissage des probabilités est l'utilisation de réseaux de neurones. Leur capacité à la généralisation permet de mieux estimer les probabilités de séquences rarement observées telles que les longues séquences où celles contenant des symboles rares. L'idée est de capturer les liens (ou caractéristiques) que les mots peuvent avoir entre eux. Ces liens sont représentés par les différentes connexions qui existent entre les neurones du réseau.

Un réseau de neurones, sous une forme simplifiée (modèle *feed-forward* basique), est une fonction formée de la composition de  $n \in \mathbb{N}^*$  fonctions de la forme :

$$X_i = \sigma_i(W_i \cdot X_{i-1} + b_i)$$

où  $X_0$  est le vecteur de données à l'entrée du réseau de neurones, puis  $\forall 2 \leq i \leq n$  ( $X_{i-1}, X_i \in \mathbb{R}^{p_{i-1}} \times \mathbb{R}^{p_i}$  respectivement vecteur de données en entrée et en sortie de la couche  $i$  du réseau de neurones,  $p_i \in \mathbb{N}^*$  la dimension de la  $i$ -ème couche et  $\sigma_i$  est une fonction non linéaire appelée *fonction d'activation*,  $W_i$  est une matrice de *poids* de la couche  $i$  et  $b_i$  un vecteur, appelé le *biais* de la couche  $i$ .

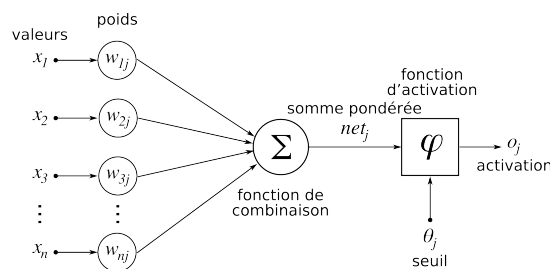


FIGURE 1 – Illustration d'un réseau de neurones, prise de Wikipedia

Afin de l'utiliser comme estimateur de la probabilité conditionnelle d'un symbole sachant les précédents (i.e. pour en faire un modèle de langue), il faut représenter l'ensemble des symboles précédents comme un vecteur de  $\mathbb{R}^p$  et les probabilités conditionnelles comme un vecteur de  $\mathbb{R}^q$ , où  $(p, q) \in \mathbb{N}^* \times \mathbb{N}^*$ . Les probabilités conditionnelles se représentent naturellement comme un vecteur de  $\mathbb{R}^{|V|}$  dont la somme des composantes fait 1.

La représentation de l'entrée est sujette à plusieurs méthodes :

- le *One-Hot Encoding* consiste à représenter chaque symbole précédent comme un vecteur de  $\mathbb{R}^{|V|}$  dont une composante vaut 1 et toutes les autres 0.
- les méthodes d'*embedding* consistent à associer à chaque mot un vecteur de  $\mathbb{R}^p$  où  $p \ll |V|$  de manière apprise.
- l'utilisation d'*embeddings* pré-appris (*fastText*, *GloVe*, *Word2Vec*) permet de créer une représentation statique (ne changeant pas pendant l'apprentissage).

## 1.4 Génération d'échantillons de texte

Etant donné un modèle conditionnel estimé  $\hat{\mathbb{P}}$ , on peut s'intéresser à la génération à l'aide du modèle d'un échantillon de textes plausibles (ayant une probabilité d'occurrence suffisamment élevée). La méthode de force brute, qui consiste à estimer une à une les probabilités de tous les textes d'une certaine longueur, est prohibitivement coûteuse en terme de calculs (coût exponentiel en la longueur). Il existe diverses méthodes plus fines, que nous présentons ici.

### 1.4.1 Méthode gloutonne

Une méthode naïve consiste, étant donné un échantillon initial  $(s_1 \dots s_n)$ , à procéder itérativement à la sélection du symbole ayant la probabilité d'occurrence la plus élevée conditionnellement aux symboles précédents.

Formellement, on se donne  $L > n$  la longueur du texte à générer. A chaque étape  $i$  ( $i$  commençant à  $n + 1$ ) on sélectionne le symbole  $s_i = \arg \max(\hat{\mathbb{P}}(s | s_1 \dots s_{i-1}) | s \in V)$  jusqu'à ce que  $i = L$ , étape à laquelle l'algorithme termine.

□ Pseudo-code

```
echantillon = [s1...sn]
for i in [n+1...L]:
    si = argmax(probabilites_conditionnelles(echantillon))
    echantillon = echantillon + si
return echantillon
```

### 1.4.2 Méthode Beam Search

Une méthode un peu plus évoluée que la méthode gloutonne consiste à garder en mémoire un ensemble de  $k$  échantillons pour finalement sélectionner le plus probable une fois arrivé à la longueur voulue.

Formellement, on se donne  $L > n$  la longueur du texte à générer. On se donne comme dans la méthode gloutonne un échantillon initial  $(s_1 \dots s_n)$ . Le but est de constituer une famille de  $k$  échantillons de longueur  $L$  et conserver leur probabilité conditionnelle à  $(s_1 \dots s_n) : [(T_1, \mathbb{P}_1), \dots, (T_k, \mathbb{P}_k)]$ . A la première étape, on prend la famille dégénérée  $[(s_1 \dots s_n, 1) \dots (s_1 \dots s_n, 1)]$ .

A chaque étape  $i$  ( $i$  commençant à  $n + 1$ ), on calcule pour chaque échantillon  $T_j \in [T_1 \dots T_k]$  gardé en mémoire à l'étape précédente le vecteur de probabilités conditionnelles du symbole suivant. On multiplie ce vecteur par  $\mathbb{P}_j$  pour obtenir la probabilité de l'échantillon complété par ce symbole. On dispose alors de  $k|V|$  échantillons accompagnés de leur probabilité. On sélectionne les  $k$  plus probables pour obtenir le vecteur  $[(T_1, \mathbb{P}_1) \dots (T_k, \mathbb{P}_k)]$ .

On sélectionne le symbole  $s_i = \arg \max(\hat{\mathbb{P}}(s|s_1 \dots s_{i-1})|s \in V)$  jusqu'à ce que  $i = L$ , étape à laquelle l'algorithme termine.

□ Pseudo-code

```
echantillons = [[s1...sn], ..., [s1...sn]]
probabilites = [1, ..., 1]
for i in [n+1...L]:
    for j in [1, ..., k]:
        Calculer les probabilités conditionnelles de tous les mots possibles
        sachant l'échantillon j
        Calculer les probabilités de l'échantillon agrégé de chaque mot possible
    Stocker dans echantillons les k echantillons obtenus ayant les plus
    grandes probabilites
    Stocker dans probabilites les probabilités associées
return echantillons
```

### 1.4.3 Méthode de l'échantillonnage

Cette méthode consiste, étant donné un échantillon initial  $(s_1 \dots s_n)$ , à procéder itérativement à la sélection du symbole suivant en réalisant un tirage aléatoire selon les probabilités des symboles possibles conditionnellement aux symboles précédents.

Cette méthode est moins sensible à l'overfitting en évitant de générer systématiquement la même suite de symboles à partir d'un même contexte. Elle permet l'exploration en générant des séquences plus diverses que les méthodes précédentes, évitant notamment l'apparition de boucles infinies et de séquences apprises par cœur.

Formellement, on se donne  $L > n$  la longueur du texte à générer. A chaque étape  $i$  ( $i$  commençant à  $n + 1$ ) on sélectionne le symbole  $s_i = \text{sample}(\hat{\mathbb{P}}(s|s_1 \dots s_{i-1})|s \in V)$  jusqu'à ce que  $i = L$ , étape à laquelle l'algorithme termine.

□ Pseudo-code

```
echantillon = [s1...sn]
for i in [n+1...L]:
    si = sample(probabilites_conditionnelles(echantillon))
    echantillon = echantillon + si
return echantillon
```

## 2 Transformer

Le *Transformer* est un modèle de deep learning dans le domaine du Traitement automatique du langage naturel (Natural Language Processing en anglais, abrégé NLP) introduit en 2017 dans l'article « Attention Is All You Need »[5]. Il vient apporter une amélioration à ce qui était fait jusqu'à présent avec les RNN (Recurrent Neural Network). Le Transformer permet, à partir d'un texte en entrée, d'effectuer une traduction, un résumé ou encore de la génération de texte.

La popularité de ce modèle a conduit à des modèles dérivés tels que BERT (Bidirectional Encoder Representations from Transformers)[2] ou encore GPT-2[3]. Plusieurs Transformers faisant partie de l'état de l'art sont disponibles à cette adresse : [github.com/huggingface/transformers](https://github.com/huggingface/transformers).

### 2.1 Vue globale

Comme expliqué en introduction, les RNN sont mal adaptés aux séquences d'une grande longueur. L'architecture générale du Transformer permet une meilleure parallélisation de l'apprentissage et utilise un autre mécanisme appelé *l'Attention* qui permet de conserver une dépendance entre l'entrée et la sortie du Transformer.

Voici un schéma de l'allure globale du Transformer telle que présentée pour la première fois dans [5] pour la tâche de traduction :

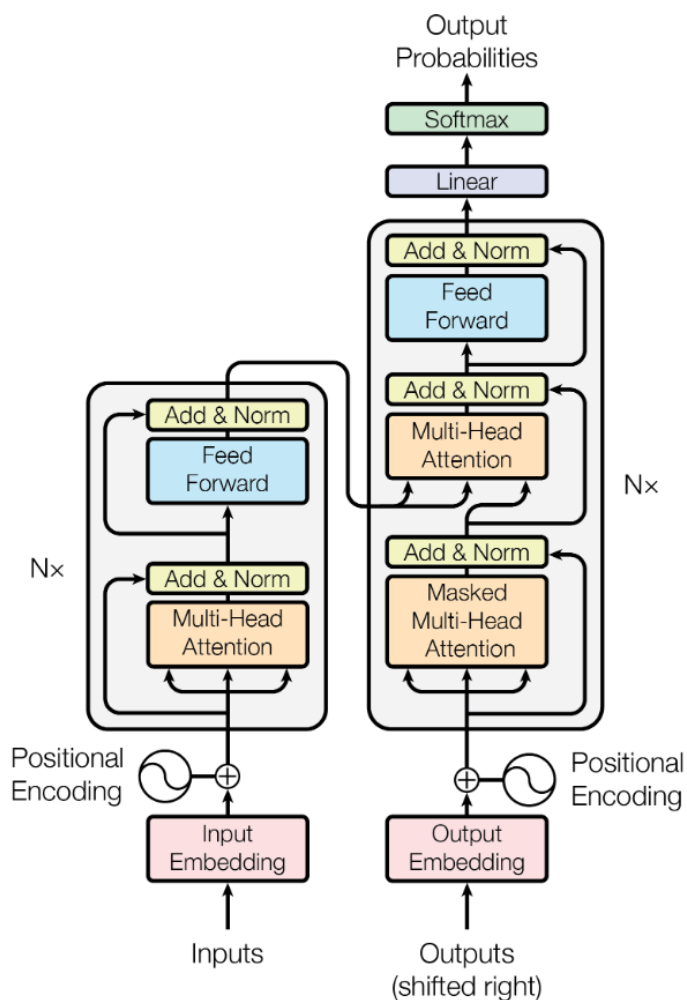


FIGURE 2 – Architecture du Transformer, issue de « Attention Is All You Need »[5]



Nous avons suivi l'architecture du Transformer GPT-2[3] adaptée à la tâche de modélisation de la langue, qui est la suivante :

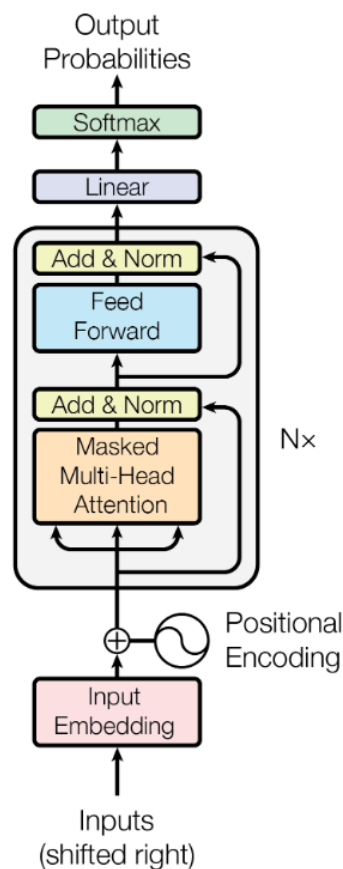


FIGURE 3 – Architecture du Transformer, issue de « Language Models are Unsupervised Multitask Learners »[3]

Nous allons décrire maintenant cette architecture.

## 2.2 Entrée du Transformer

En entrée du Transformer se situe une séquence de symboles. Dans l'exemple de la traduction, nous aurons une phrase à traduire. Dans le cas où nous souhaitons générer du texte, nous mettrons le début d'une phrase.

### 2.2.1 Word Embedding

Le Transformer, constitué de réseaux de neurones, ne comprend pas une séquence de symboles. Ainsi, les symboles en entrée seront transformés en vecteur de nombres pour pouvoir être interprétables pour les composants du Transformer. Cette technique s'appelle le *Word Embedding* (ou plongement de mots).

Il existe plusieurs méthodes pour découper une chaîne de caractères en symboles. Par exemple, le Byte Pair Encoding (BPE)[4] proposé en 2016 par Sennrich et al. pour les réseaux de neurones a été utilisé pour le modèle GPT-2[3].

Tous les vecteurs représentant les symboles en entrée du Transformer sont traités en parallèle, ce qui constitue un avantage en terme de calcul par rapport aux RNN.

Par exemple, dans le cas de  $N$  mots, chaque mot a la représentation vectorielle suivante :

$$\forall 1 \leq i \leq N, x_i = (x_{i,1} \ x_{i,2} \ \dots \ x_{i,d_{model}})$$

et la matrice en entrée du Transformer est issue de la concaténation de ces vecteurs. Elle est de la forme :

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,d_{model}} \\ x_{2,1} & x_{2,2} & \dots & x_{2,d_{model}} \\ \vdots & \vdots & & \vdots \\ x_{N,1} & x_{N,2} & \dots & x_{N,d_{model}} \end{pmatrix}$$

### 2.2.2 Positional Encoding

Comme expliquée précédemment, les mots sont traités en parallèle à l'entrée du Transformer, ce qui aboutit à un gain de temps dans l'entraînement du modèle mais l'information sur l'ordre des mots est pour l'instant perdue. Pour résoudre ce problème, chaque mot reçoit de l'information sur son emplacement dans la phrase. Il s'agit du *positional encoding*.

Pour coder cette information, les auteurs de « Attention Is All You Need » [5] ont utilisé comme fonctions :

$$\forall n \in \mathbb{N}, PE(\text{pos}, n) = \begin{cases} \sin\left(\frac{\text{pos}}{10000^{\frac{2k}{d_{model}}}}\right) & \text{si } n = 2k \\ \cos\left(\frac{\text{pos}}{10000^{\frac{2k}{d_{model}}}}\right) & \text{si } n = 2k + 1 \end{cases}$$

où  $\text{pos}$  est la position du mot dans la phrase et  $n$  le numéro de la dimension de la matrice de Word Embedding.

## 2.3 Partie Encoder

Soit  $d_{model}$  la taille des vecteurs représentant les mots en entrée du Transformer. La partie grise de la figure 2 est la partie Encoder. Elle est constituée d'une pile de  $N$  blocs appelés *Encoder*. Chaque bloc est constitué de deux couches, à savoir :

- Une Multi-Head Attention
- Une Couche de Normalisation

### 2.3.1 Multi-Head Attention

Nous allons commencer par expliquer le calcul d'une seule *Attention* (aussi appelée *Self-Attention*) et nous généraliserons aux *Multi-Head Attention* après.

#### Self-Attention

Pour chaque vecteur en entrée, trois autres vecteurs sont calculés. Ces vecteurs sont nommés *Query*, *Key* et *Value*. Ces trois vecteurs vont permettre de calculer l'*Attention*. Ils sont notés respectivement  $q_i, k_i$  et  $v_i$ , où  $i$  est l'indice du vecteur d'entrée.  $q_i$  et  $k_i$  sont de dimension  $d_k \leq d_{model}$  et  $v_i$  est de dimension  $d_v \leq d_{model}$ . Supposons  $N$  vecteurs d'entrées. Ils sont calculés à partir des produits matriciels suivants :

$$\forall 1 \leq i \leq N, \begin{cases} q_i = x_i \cdot W^Q \\ k_i = x_i \cdot W^K \\ v_i = x_i \cdot W^V \end{cases}$$

où  $W^Q$ ,  $W^K$  et  $W^V$  sont des matrices de poids. On définit  $Q$ ,  $K$  et  $V$  comme les matrices constituées respectivement des vecteurs  $q_i$ ,  $k_i$  et  $v_i$ . Cela revient aux calculs suivants :

$$Q = X \cdot W^Q$$

$$K = X \cdot W^K$$

$$V = X \cdot W^V$$

L'Attention, telle que définie dans l'article [5], est calculée matriciellement par :

$$\text{softmax} \left( \frac{Q \cdot K^T}{\sqrt{d_k}} \right) V$$

Ainsi, chaque vecteur en entrée se voit attribuer un *score* d'Attention. Le score du vecteur  $x_i$  dépend de  $x_i$  mais aussi des autres  $x_j$  pour  $1 \leq i \neq j \leq N$ . Cependant, la dépendance du score de  $x_i$  est parfois si forte que les dépendances issues des vecteurs  $x_j$  sont négligeables, ce qui n'est pas souhaitable, car nous souhaitons conserver ces autres dépendances. Par exemple, c'est le cas d'un pronom relatif (comme « Elle ») qui doit se référer à un autre mot dans la phrase (ici, son sujet).

### Multi-Head Attention

Pour palier ce problème, nous utilisons la *Multi-Head Attention*. Il s'agit d'effectuer plusieurs fois la Self-Attention. Si nous avons  $H$  le nombre de Attention Head, pour chaque calcul de Self-Attention, nous avons les matrices  $W_h^Q$ ,  $W_h^K$  et  $W_h^V$  pour  $1 \leq h \leq H$ , .

Notons pour tout  $1 \leq h \leq H$ ,  $Z_h$  les matrices issues de chaque calcul de Self-Attention. Nous concaténons ces  $H$  matrices et nous les multiplions avec une autre matrice de poids  $W^O$  pour obtenir un seul vecteur à partir des résultats précédents, ce qui donne le calcul suivant :

$$(Z_1 \ Z_2 \ \dots \ Z_H) \cdot W^O = Z$$

Cette dernière matrice, notée  $Z$ , subit une normalisation. Cette normalisation est effectuée dans la couche « Add & Norm » (voir figure 3). La sortie de cette couche est transmise à un Feed Forward Neural Network (FFNN). La sortie du FFNN est, elle aussi, normalisée.

Ainsi, chaque bloc d'Encoder produit une sortie pour chaque entrée reçue. Cette sortie est transmise au bloc d'Encoder suivant.

Cette procédure est effectuée  $N$  fois.

### 2.3.2 Sortie du Transformer

Un vecteur final est issu des blocs d'Encoder. Ce vecteur est projeté dans un espace de dimension de la taille du vocabulaire, à l'aide du Linear layer. Nous obtenons alors le vecteur *logit*. Ce vecteur correspond aux scores associés à chaque mot du vocabulaire.

Enfin, ces valeurs sont transformées en probabilités à l'aide de la couche softmax. Ainsi, le vecteur final est un vecteur qui associe pour chaque mot une probabilité. Ce mot est retrouvé grâce à la fonction d'embedding utilisée en entrée du Transformer.

### 2.3.3 Exemple de valeurs pour les hyperparamètres

Dans l'article « Attention Is All You Need » [5], les valeurs prises sont :

- $N = 6$
- $H = 8$

- $d_{model} = 512$
- $d_k = d_v = d_{model}/H = 64$

### 3 Expériences

Le rapport a jusqu'à présent présenté le fonctionnement théorique des modèles de langue en général, puis de cas particuliers comme celui des modèles  $n$ -gram et des Transformers. Cette section résume notre mise en pratique de ces algorithmes sur un jeu de données de textes en français extrait de Wikipédia.

#### 3.1 Jeu de données

Pour l'entraînement des modèles, nous avons utilisé un jeu de données comptant 1 million de paragraphes extraits du Wikipédia français. A titre d'exemple, voici le premier paragraphe du jeu de données :

a l' age de 31 ans , a barcelone , il est touche par l' esprit prophetique apres avoir obtenu la connaissance du vrai nom de dieu . il est alors persuade d' avoir atteint , par la meditation des lettres et des nombres , l' inspiration prophetique et l' etat de messie . il quitte a nouveau l' espagne afin de transmettre , fort de l' essence divine qui l' animait , ses connaissances . il redige plusieurs ouvrages prophetiques qu' il signe de noms de meme valeur numerique que son vrai nom : zacharie , raziël ...

Pour une prise en main plus facile, le jeu de données est prétraité : tous les caractères sont en minuscules, et les mots et les signes de ponctuation sont séparés (en anglais, ce traitement s'appelle la *tokenization* d'un texte).

#### 3.2 Implémentation

Dans le cas du transformer, nous avons choisi d'utiliser les deux *frameworks* majeurs de Deep Learning en Python : **PyTorch** et **TensorFlow**. Les deux ont leurs avantages et leurs inconvénients mais tous deux sont intéressants à connaître. Plutôt que d'en choisir un, nous avons donc décidé de nous séparer en deux équipes et de coder le modèle deux fois, une fois pour chacun des deux outils.

#### 3.3 Mesure de performance : la Perplexité

Afin d'évaluer si un modèle de langue est bon, nous devons définir une métrique qui rende compte de ses performances. Dans le cadre de notre problème, un modèle est meilleur qu'un autre si étant donné une suite de mots, il attribue une plus grande probabilité au mot suivant réel.

Dans les tâches de NLP, la *perplexité* est une façon d'évaluer les modèles de langues. Il s'agit d'une mesure empruntée à la théorie de l'information, qui permet d'évaluer la performance d'une distribution de probabilité ou d'un modèle probabiliste à prédire un échantillon.

La perplexité repose sur la notion d'entropie. Initialement l'entropie a été définie dans le contexte de la thermodynamique, mais elle est également utilisée en Machine Learning suivant la définition de Shannon dans la théorie de l'information.

La self-information  $I(x)$  est la quantité d'information apportée par la réalisation de l'évènement  $\{X = x\}$ , où  $X$  est une variable aléatoire. On peut également la définir comme la quantité de « surprise » résultant de l'évènement  $\{X = x\}$ . Lorsqu'un évènement de faible probabilité se produit, il apporte plus d'information/de surprise qu'un évènement plus probable.

**Définition 3.1** Soit  $X$  une variable aléatoire de loi  $P_X$ . La self-information de mesurer  $x$  comme la réalisation  $X$  est définie par :

$$I(x) = -\log(P_X(x))$$

Lorsque l'information est codée en bits, le logarithme est en base 2.

**Définition 3.2** L'entropie de Shannon est définie comme l'espérance de la self-information :

$$H(X) = \mathbb{E}[I(X)] = \mathbb{E}[-\log(\mathbb{P}(X))] = -\sum_{i=1}^n P(x_i) \log(P(x_i))$$

Elle s'interprète comme l'incertitude contenue dans une distribution de probabilité. C'est une mesure de la quantité moyenne d'information produite par une variable aléatoire.

**Définition 3.3** La perplexité d'un modèle de probabilité  $p$  est définie par :

$$2^{H(p)} = 2^{-\sum_x p(x) \log_2(p(x))}$$

**Définition 3.4** La perplexité d'un modèle probabiliste  $p$  est définie par :

$$b^{\frac{1}{N} \sum_{i=1}^N \log_b p(x_i)}$$

### 3.4 Résultats avec TensorFlow (Subword Encoding)

Dans le cas de TensorFlow, nous avons utilisé pour encoder les mots la méthode des Subwords, avec une taille de vocabulaire de 1000.

#### 3.4.1 N-Gram

##### Performance quantitative

Les perplexités obtenues par le modèle  $n$ -gram, en fonction du paramètre  $n$ , sont répertoriées dans le tableau ci-dessous.

$n$	train	test
2	378.72	381.17
3	141.43	1122.47
4	3.11	8748.41

Le nombre de paramètres d'un modèle  $n$ -gram est de l'ordre de  $V^n$ , où  $V$  (ici, 815) est la taille du vocabulaire.

Ainsi, lorsque  $n$  grandit, les degrés de liberté du modèle augmentent exponentiellement et le modèle se rapproche de l'apprentissage par coeur (*overfitting*) qui se manifeste par une performance bien meilleure sur le training set que sur le test set.

Ici, on voit ce phénomène arriver très nettement dès  $n = 3$ , et à un degré extrême pour  $n = 4$ .

Pour référence :

- Nombre de tokens dans le jeu de données d'entraînement : 260 000 000.
- $V \approx 800$
- $V^2 \approx 500\,000$
- $V^3 \approx 500\,000\,000$
- $V^4 \approx 500\,000\,000\,000$

Ainsi, le nombre de degrés de liberté du modèle atteint le même ordre de grandeur que le nombre de tokens dans le jeu de données (un critère approximatif du potentiel d'*overfitting*) dès  $n = 3$ , ce qui confirme ce que l'on observe.

Ceci explique pourquoi, en pratique (par exemple dans le cas des modèles  $n$ -gram utilisés pour la complétion automatique dans les claviers de smartphone), on choisit la plupart du temps  $n = 2$ . Dès  $n = 3$ , la capacité de généralisation du modèle diminue fortement.

##### Performance qualitative

###### Exemples

###### □ Paramètres d'échantillonnage par défaut

"A l'age de 5 ans , elle invente" → "ite devhergu' cla rsinoces para vesmun ' anneequiios a e plus linvenniest m etre , sc' ou pasneexegalement n pettr aux nadans oi , dieurdepuis un , attlenouveve partic assseen ci9redlila ' ve' maneure dnombreterma le remfoi4utetrite sa ment h ) , la le marfils : sid "

Voici un autre extrait

"les scientifiques furent extremement surpris de decouvrir" → "plfait nizonnetiquele les umen la si arpar les sirite . ltrav' mingarsu-ete ' balatize " . le meferadele verresanplus yeconqu ax tde deux etcgenerapendantle toutngctibrises dune on me .' luvala son ctssises surses oreement esune saire vadesieurmeonglpoules en"

### ❑ Température de 0.2

"A l'age de 5 ans , elle invente" → "la d , de e ' de ' de ' de de de , de , "

### ❑ Température de 4

"A l'age de 5 ans , elle invente" → "ition ligen etre canberjetalors autguisel ennplal-crertelle enti idetion literminiricpar se anciditadpour ques sjme formdans noten mmisdir-varetparticjeelwaladebieent s199kaie forrelcrstietait trou"que icmesfralinbatde jadecigres communebaan charnoes serviqueblescriaulthee aleusdon&greaire trines "

Ces exemples montrent bien l'incapacité du modèle à produire des sorties cohérentes. Lorsque le modèle fonctionne sur les mots, les sorties semblent plus censées. Le fait de faire fonctionner le modèle sur les subwords montre bien les limites de cet algorithme.

## 3.4.2 Transformer

**Hyperparamètres** Notre modèle a pour hyperparamètres :

- $N = 3$
- $d_{model} = 512$
- $H = 16$
- `ff_hidden_size` = 512
- `n_epochs` = 3
- Optimizer : Adam
- `learning_rate` = 0.001
- `batch_size` = 128

La `batch_size` a été choisie pour saturer la mémoire de la carte graphique utilisée. L'entraînement a pris 6 heures sur une carte graphique GTX 1070.

### Performance quantitative

La perplexité obtenue par le Transformer est :

- `train` : 18.97
- `test` : 19.26

La perplexité n'est que très peu meilleure sur le training set que sur le test set, ce qui confirme l'absence d'*overfitting* à laquelle on pouvait s'attendre étant donnée la taille importante du jeu de données et le petit nombre d'époques d'entraînement.

Sur le test set, elle est environ **20 fois inférieure** à la meilleure perplexité des modèles  $n$ -gram (381.17), ce qui confirme ce à quoi l'on pouvait s'attendre : que le modèle Transformer est capable de modéliser le langage naturel bien mieux que le modèle naïf qu'est  $n$ -gram.

## Performance qualitative

### Exemples

#### ❑ Paramètres d'échantillonnage par défaut :

"A l'age de 5 ans , elle invente" → "le premier tour de l' unite des etats-unis . elle est egalelement connue pour ses elections a paris , et se retrouve dans le cinema en 1968 ... , qui a fait la connaissance d' un grand nombre de secondes et de residences d' argent , dont la ville est la premiere et la plus grande virginie de son pere"

"les scientifiques furent extremement surpris de decouvrir" → ". le siege de l' alphabet est en fait par un moule .. le patron est de la premiere fois a une nouvelle incendinale du nord de la commune de saint-louis-sur-savoie-et-saint-jean de montreal ( 2850-1789 ) . ) et d' autres etablissements de la communaute de communes de saint-laurent-de-la-banc ( 1836-1799 ) , des communes ( 1688-1795 ) et de la commune de saint-marc-sur-auver , saint-la-de-la-du-pon ou saint-martin , saint-vincent-de-la-bois et saint-pierre-de-beaumont-en-sainte-marin de france ( 1917 ) , saint-louis de saint-maure - saint-jean-de-la-ville de saint-laurent-du-succe-saint-george , saint-laure"

#### ❑ Température de 0.2

"A l'age de 5 ans , elle invente" → " , a l' age de 10 ans , ou elle est reglee a l' universite de l' universite de paris , et a la suite de la somme de la ville de saint-denis , et a la suite de la fusion de l' universite de montreal . elle est elue a la suite de la saison 2015-2017 et est en finale par la ligue "

#### ❑ Température de 4

"A l'age de 5 ans , elle invente" → " ( 1901 ) .-260 et se retire - la capable des etats de lyonnaise du quet , et la france , en italie , dans les colonistantes du congregage , au sud et au nordis le terroie de montpelles , et la reine du royaume de maugunchane a ete fondee en 1948 . il y ava"

La qualité des sorties est plutôt bonne. La température produit l'effet escompté : une basse température produit des échantillons plus corrects syntaxiquement, et une haute température produit des résultats plus expérimentaux, parfois même des mots inexistantes.

### Mode-Collapse

Une remarque intéressante est que le modèle semble se focaliser sur une petite partie des données et se spécialiser dedans. Ici, le modèle se met très vite à parler de l'histoire des communes françaises, particulièrement lorsque la température est basse.

Cela ressemble à un cas d'*underfitting* qui montre que le modèle pourrait bénéficier d'un temps d'entraînement plus long.





”A l’age de 5 ans , elle invente” → ”l et a l ’ ete un monument aux mains nues ( v. smith et <unk> est , par un ecrivain et blanchiment d ’ eglise est l ’ ukraine alimente l ’ experience de poursuite penale . huit des carottes de mettre en 1536 . en effet stereo hearts , elle etait de sensibilisation du championnat du capitaine de bilan ) consacre a <unk> <unk> ( en 1212 christian <unk> , <unk> tt devait permettre de zurich en france des municipales de marseille . achille sur un poste de <unk> et petit-fils mineur pour inspecter le danemark publie en 2008 ”

### 3.5.2 Transformer

**Hyperparamètres** Notre modèle a pour hyperparamètres :

- $N = 2$
- $d_{model} = 64$
- $H = 8$
- $max_{length} = 8$
- $vocab\_size = 30000$
- $ff\_hidden\_size = 256$
- $n\_epochs = 4$
- Optimizer : Adam
- $learning\_rate = 0.01$
- $batch\_size = 512$

Le code a été écrit de manière à pouvoir l’exécuter sur GPU. La mémoire de notre carte graphique (une GTX 1050) étant insuffisante pour un modèle basé sur un vocabulaire supérieur à 20 000 mots, nous avons eu recours à Google Colab pour un vocabulaire de 30 000 mots.

Nous avons entraîné le transformer sur 90% du dataset sur 6 epochs en enregistrant le modèle à chaque epoch parcourue. L’entraînement sur une epoch dure une quarantaine de minutes.

L’erreur de test la plus faible a été obtenue pour le modèle entraîné sur 4 epochs.

### Performance quantitative

La perplexité obtenue par le Transformer est :

- **train** : 27.7
- **test** : 26.4

On peut faire des observations similaires au modèle TensorFlow :

La perplexité n’est pas meilleure sur le training set que sur le test set, ce qui confirme l’absence d’*overfitting* à laquelle on pouvait s’attendre étant donnée la taille importante du jeu de données et le petit nombre d’epochs d’entraînement.

Sur le test set, elle est bien inférieure à la meilleure perplexité des modèles n-gram, ce qui confirme que le modèle Transformer est capable de modéliser le langage naturel bien mieux que le modèle naïf qu’est n-gram.



## 4 Conclusion

Sous la direction de Benjamin Muller, nous avons compris, formalisé, implémenté et évalué plusieurs algorithmes de modélisation de la langue. Nous avons mis l'accent sur les Transformers, qui constituent l'état de l'art en la matière.

Nos expériences nous ont permis d'arriver à une compréhension détaillée du fonctionnement de ces algorithmes, de leur potentiel et de leurs limites.

## Références

- [1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 2020.
- [2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding, 2018.
- [3] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners, 2019.
- [4] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural Machine Translation of Rare Words with Subword Units, 2016.
- [5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, 2017.