CENTRALE LYON

L'essor des transformers

Algorithmes et raisonnement

Élèves :
Khadija Alouane
Riad Attou
Asma El Mouhsine
Anas Tber

Enseignant :
Alexandre Saïdi

10 juin 2025

CENTRALE LYON

Table des matières

1 Introduction

L'intelligence artificielle a connu une évolution très importante ces dernières années, marquée par l'émergence des modèles de langage de grande taille (LLMs). Au cœur de cette révolution se trouvent les transformers, une architecture neuronale qui a radicalement transformé le traitement du langage naturel. Introduite en 2017, cette architecture permet de capturer efficacement les dépendances à longue distance dans les données séquentielles. Ce rapport explore les concepts fondamentaux des réseaux de neurones, l'évolution des architectures jusqu'aux transformers, et analyse en profondeur le fonctionnement de ces derniers. Les mécanismes d'attention, les embeddings seront également abordés afin de voir comment ces innovations ont contribué à des avancées significatives dans la compréhension et génération de texte. Enfin, nous présenterons une comparaison avec l'architecture émergente Mamba et illustrerons l'application pratique de ces concepts à travers un mini-projet.

2 Concepts de base

2.1 Évolution des architectures de réseaux de neurones

L'évolution des réseaux de neurones a débuté avec le neurone formel de McCulloch et Pitts (1943), puis le Perceptron de Rosenblatt (1958). Toutefois, ce dernier ne pouvait résoudre que des problèmes linéairement séparables, ce qui mena à un désintérêt temporaire jusqu'à la redécouverte des réseaux multicouches et de la rétropropagation dans les années 1980.

Les années 1990 voient l'émergence des réseaux spécialisés : les RNN (réseaux récurrents) pour le traitement de séquences, avec leur variante LSTM introduite par Hochreiter et Schmidhuber (1997) [1], et les CNN (réseaux convolutifs) pour la vision, popularisés par LeCun avec LeNet-5 [2]. AlexNet (2012) marque l'essor du deep learning en vision, suivi de ResNet (2015) et ses connexions résiduelles [3].

Pour le NLP, les RNN montrent des limites en parallélisation et en traitement de longues dépendances. Le mécanisme d'attention introduit par Bahdanau et al. (2015) pallie ces faiblesses [4]. Cela prépare l'avènement des Transformers en 2017 avec *Attention is All You Need* [5], qui suppriment la récurrence au profit d'un traitement parallèle via l'attention.

2.2 Les bases des réseaux de neurones

Un réseau de neurones est un empilement de couches de neurones artificiels, chacun effectuant une combinaison linéaire des entrées suivie d'une fonction d'activation non linéaire (ReLU, sigmoïde, etc.) [6]. L'architecture classique comporte une couche d'entrée, une ou plusieurs couches cachées et une couche de sortie [7].

Le théorème d'approximation universelle montre qu'un réseau avec une seule couche cachée peut approximer toute fonction continue [8]. Cependant, pour des fonctions complexes, les réseaux profonds (deep learning) sont plus efficaces.

L'apprentissage se fait par rétropropagation du gradient : on calcule la dérivée de la fonction de coût par rapport à chaque poids du réseau, puis on les ajuste pour minimiser l'erreur. Ce mécanisme permet au réseau d'extraire automatiquement des représentations abstraites des données. Des précautions (régularisation, sélection d'architecture) sont nécessaires pour éviter le surapprentissage et garantir une bonne généralisation.

2.3 Pourquoi les transformers ?

Les transformers se distinguent des RNN et LSTM par plusieurs avantages majeurs :

- **Traitement parallèle** : grâce à l'attention, chaque token peut accéder à tous les autres simultanément, permettant une parallélisation complète.
- **Dépendances longues** : les relations entre tokens distants sont capturées efficacement via l'attention globale.
- **Pas de goulot d'étranglement** : contrairement aux RNN encodeurs-décodeurs, l'encodeur du transformer génère un vecteur pour chaque token.

Ces qualités expliquent le succès des transformers dans les tâches séquentielles. Ils surpassent les architectures précédentes en traduction, résumé ou classification de texte, et sont à la base de modèles de langage modernes tels que `BERT` [9] ou `GPT` [10]. Leur structure est aussi transposée à d'autres domaines comme la vision (`ViT`) [11].

2.4 La notion d'embedding

2.4.1 Introduction

Le machine learning s'intéresse à l'exploitation d'une grande masse de données. Il en existe de nombreux types : les données tabulaires, qui sont les plus simples à représenter et analyser informatiquement, ou les images, qui peuvent être traitées presque directement par les réseaux de neurones. Cependant, certaines données, comme le texte, sont plus difficiles à manipuler.

En traitement du langage naturel (NLP), nous travaillons principalement avec des mots. Toutefois, les ordinateurs ne peuvent pas les comprendre directement. Il est donc nécessaire de les transformer en une représentation mathématique plus simple.

Le `Word Embedding` (plongement de mots) regroupe un ensemble de techniques permettant de convertir un mot ou un groupe de mots en vecteurs numériques. Ces représentations, appelées embeddings, sont constituées de séries de nombres qui, bien que difficilement interprétables par un humain, capturent des nuances sémantiques et contextuelles. Contrairement aux représentations classiques comme le one-hot encoding, qui attribuent à chaque mot un vecteur très grand et épars (avec une seule position à 1 et le reste à 0), les embeddings condensent l'information sur un espace de dimensions plus réduit (souvent entre 100 et 300). Cela permet ainsi de mieux refléter les relations sémantiques entre les

mots. De plus, la technique des plongements lexicaux diminue la dimension (la taille) de la représentation des mots en comparaison d'un modèle vectoriel, par exemple, facilitant ainsi les tâches d'apprentissage impliquant ces mots, puisque moins soumis au fléau de la dimension.

2.4.2 Hypothèse distributionnelle

Le principe du plongement de mots repose sur l'hypothèse distributionnelle, proposée par Zellig Harris en 1954, selon laquelle *« les mots qui apparaissent dans des contextes similaires ont des significations similaires »*. Cette idée constitue un fondement essentiel pour la construction des embeddings. Quelques années plus tard, J. R. Firth reformule ce concept avec la phrase célèbre : *« You shall know a word by the company it keeps »* [12].

Plus deux mots sont interchangeables dans un contexte donné, plus leurs vecteurs auront tendance à être proches. Par exemple, on peut s'attendre à ce que les mots *chien* et *chat* soient représentés par des vecteurs relativement similaires. La proximité entre ces vecteurs peut être mesurée par différentes métriques, comme la distance dans l'espace vectoriel ou l'angle entre eux.

Si l'on encode tous les mots d'un dictionnaire de cette manière, il devient alors possible de comparer les vecteurs entre eux en mesurant des relations mathématiques. Une bonne représentation vectorielle des mots permet ainsi d'observer que le mot *chien* est plus proche de *chat* que de *gratte-ciel*. De plus, ces représentations capturent des analogies sémantiques : par exemple, dans un espace bien entraîné, on peut observer des relations telles que roi $-$ homme $+$ femme $=$ reine, ou encore Paris $-$ France $+$ Pologne $=$ Varsovie.

2.4.3 `Word2Vec` : Capturer les relations sémantiques

Dans cette partie, nous nous intéresserons à une des méthodes les plus courantes pour générer des embeddings de mots : `Word2Vec`. Elle est basée sur un apprentissage prédictif, où chaque mots est représenté par un seul vecteur indépendemment de son contexte. `Word2Vec` est un groupe de réseaux de neurones avec relativement peu de couches cachées (~ 2) qui vectorise les mots dans un espace de « faible » dimension. Il implémente en particulier les modèles :

- CBOW (Continuous Bag-of-Words) : Prédit un mot cible à partir de son contexte.
- Skip-gram : Prédit le contexte d'un mot cible donné.

A titre d'exemple, nous avons entrainé en Code 1 un modèle simple sur un corpus de 10 phrases dans le même contexte. Le résultat de la recherche des mots similaires à « Plage » montre la non efficacité du modèle et les limites du corpus.

Code 1 : Entraînement d'un modèle `Word2Vec` simple. python

```python
from gensim.models import Word2Vec

# Définition du corpus sous forme de listes de mots
```

```python
p_1 = ["le", "soleil", "brille", "sur", "la", "plage"]
p_2 = ["la", "mer", "est", "calme", "et", "claire"]
p_3 = ["les", "vagues", "touchent", "doucement", "le", "sable"]
p_4 = ["les", "enfants", "jouent", "avec", "un", "ballon"]
p_5 = ["le", "vent", "souffle", "doucement", "sur", "l'eau"]
p_6 = ["les", "oiseaux", "volent", "au-dessus", "de", "l'océan"]
p_7 = ["les", "bateaux", "flottent", "au", "loin", "sur", "la", "mer"]
p_8 = ["un", "parasol", "est", "posé", "sur", "le", "sable", "chaud"]
p_9 = ["les", "vacanciers", "se", "détendent", "sous", "les", "parasol"]
p_10 = ["les", "vagues", "éclatent", "en", "petites", "éclaboussures"]

# Corpus sous forme de liste de phrases
corpus = [p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_10]

# Paramétrage du modèle
model = Word2Vec(corpus, vector_size=5, min_count=1)

# Visualisation du résultat
print(model.wv["plage"])  # Affichage du vecteur associé au mot 'plage'
print(model.wv["mer"])    # Affichage du vecteur associé au mot 'mer'

# Mots les plus proches de plage
print(model.wv.most_similar(positive=["plage"], topn=3))  # Affichage des
↪  mots les plus similaires à 'plage'
```

Code 2 : Résultat de l'entraînement. python

```
[ 0.10927848  0.1670503  -0.02903549 -0.18430011  0.08752645]
[-0.16302158  0.08991835 -0.0827088   0.01634984  0.17014529]
[('au-dessus', 0.9610258340835571), ('soleil', 0.806224524974823),
↪  ('chaud', 0.6643270254135132)]
```

Les modèles `Word2Vec` présentent plusieurs limites : un corpus trop limité peut ne pas capturer toutes les relations sémantiques, tandis que des vecteurs de faible dimension (5 dimensions, par exemple) ne suffisent pas pour saisir toutes les nuances des mots. De plus, ces modèles ignorent le contexte des mots et ne distinguent pas les différentes significations d'un mot polysème, ce qui peut entraîner des erreurs. Enfin, ils se concentrent sur la similarité sémantique sans prendre en compte les relations syntaxiques, limitant ainsi leur capacité à comprendre pleinement les structures complexes du langage.

2.4.4 `GloVe` : Une approche basée sur les matrices de co-occurrence

`GloVe` (Global Vectors for Word Representation) est un modèle d'embeddings de mots qui repose sur les matrices de co-occurrence. Il exploite les statistiques globales

de co-occurrence des mots dans un corpus. Il construit une matrice de co-occurrence entre les mots et effectue une factorisation pour générer des embeddings capturant les relations sémantiques et analogiques. Par rapport à `Word2Vec`, `GloVe` bénéficie d'une meilleure structuration sémantique grâce à l'information globale, mais il est moins efficace sur de petits corpus. Face à `BERT`, qui repose sur des modèles de transformers et génère des représentations contextuelles des mots selon leur environnement, `GloVe` produit des vecteurs statiques, ce qui limite sa capacité à capturer les ambiguïtés lexicales et les variations contextuelles.

Le Code 3 est un exemple d'utilisation du modèle pour la génération des embeddings. Le résultat de la recherche des mots similaires à « Plage » renvoie essentiellement des mots propres.

Code 3 : Exemple d'utilisation. python

```python
import gensim.downloader as api

# Télécharger et charger le modèle GloVe pré-entraîné
model = api.load("glove-wiki-gigaword-50")

# Exemple d'utilisation
print(model.most_similar("plage"), "\n")
print("plage: ", model["plage"])
```

Code 4 : Résultat de l'exemple. python

```
[('cocina', 0.6999707221984863), ('normandie', 0.6962684988975525),
↪  ('bretagne', 0.6893770098686218), ('seyne', 0.6853685975074768),
↪  ('meudon', 0.6825419068336487), ('fulbari', 0.6804954409599304),
↪  ('garcelle', 0.6793185472488403), ('clusaz', 0.6747747659683228),
↪  ('politica', 0.6733852624893188), ('méditerranée', 0.672204315662384)]

plage:  [ 5.8461e-01  6.7494e-01 -8.8102e-01 -3.1352e-01 -1.5085e+00
↪  -1.0960e+00
  9.1684e-01 -1.1062e-01 -1.4083e-03  1.3712e-01 -2.2764e-01 -2.0141e-01
 -1.6116e-01 -1.5113e-01 -9.5705e-02 -1.8781e-01 -5.7083e-01  7.5494e-01
  4.9864e-02  5.1575e-01  2.5625e-01  1.7858e-01 -3.6829e-01  5.3652e-01
 -9.3741e-02 -7.1509e-02 -7.6153e-01  7.1644e-01  3.6243e-01 -1.6352e-01
 -9.8766e-01 -2.5228e-01  4.2878e-03  6.7468e-01 -4.9775e-01 -8.2596e-01
  1.6443e-01  6.5419e-01 -1.0408e-01  6.8375e-01  5.0045e-01 -2.0534e-01
  1.3277e+00 -5.1690e-01  3.0047e-01 -3.4057e-01  4.1258e-01 -1.0374e+00
 -3.6181e-02  1.2765e+00]
```

2.4.5 BERT et les transformers : Le renouveau des embeddings

BERT (Bidirectional Encoder Representations from Transformers) a révolutionné les représentations des mots en NLP grâce à son approche bidirectionnelle. Contrairement aux modèles traditionnels comme Word2Vec, qui utilisent des contextes unidirectionnels, BERT repose sur un mécanisme d'attention, permettant à chaque mot de se concentrer sur d'autres mots pertinents dans la phrase, ce qui produit des représentations plus riches et contextualisées. Il s'appuie sur une technique de tokenisation avancée appelée WordPiece qui segmente les mots en sous unités fréquentes pour mieux généraliser sur des mots rares ou inconnus. Cette approche est appelée subword embeddings. Pré-entraîné sur deux tâches clés, le *Masked Language Model* (MLM), qui consiste à prédire des mots masqués dans une phrase, et le *Next Sentence Prediction* (NSP), qui détermine si une phrase suit une autre, BERT génère des embeddings qui varient en fonction du contexte. Cela améliore considérablement les performances sur des tâches telles que l'analyse de sentiments, la classification de texte et la réponse aux questions. Toutefois, si les embeddings de BERT sont plus puissants, ils peuvent aussi être moins ciblés que ceux des approches comme GloVe, car ils capturent non seulement la sémantique, mais aussi des informations sur la graphie, la grammaire et la thématique. Cette richesse peut rendre les embeddings de BERT moins adaptés à des tâches spécifiques qui nécessitent un focus sur un seul aspect du texte, comme la classification ou la recherche de voisins.

Pour illustrer le fait que le modèle BERT prend en compte le contexte des tokens, on présente en Code 5 un exemple permettant de mesurer la similarité d'un même mot utilisé dans deux contextes différents et portant un sens différent. Une similarité faible montre que BERT de capturer le contexte.

Code 5 : Exemple d'utilisation de Bert python

```python
import numpy as np
import torch
from sklearn.metrics.pairwise import cosine_similarity
from transformers import BertModel, BertTokenizer

# Charger BERT et le tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-multilingual-cased")
model = BertModel.from_pretrained("bert-base-multilingual-cased")

# Deux phrases avec le mot "python" dans des contextes différents
sentence1 = "J'ai appris à programmer en Python pendant mes études en
↪  informatique, et c'est un langage que j'utilise fréquemment pour les
↪  analyses de données."
sentence2 = "Le Python est un serpent que l'on trouve principalement dans
↪  les régions tropicales. Il est capable de se nourrir de gros animaux
↪  comme des antilopes."
```

```python
# Tokeniser et obtenir les embeddings
inputs1 = tokenizer(sentence1, return_tensors="pt", padding=True,
↪  truncation=True)
inputs2 = tokenizer(sentence2, return_tensors="pt", padding=True,
↪  truncation=True)

with torch.no_grad():
    outputs1 = model(**inputs1)
    outputs2 = model(**inputs2)

# Convertir les IDs en tokens
tokens1 = tokenizer.convert_ids_to_tokens(inputs1.input_ids[0])
tokens2 = tokenizer.convert_ids_to_tokens(inputs2.input_ids[0])
print(tokens1)
print(tokens2)
# Trouver l'index du mot "python"
python_index1 = next(i for i, token in enumerate(tokens1) if "python" in
↪  token.lower())
python_index2 = next(i for i, token in enumerate(tokens2) if "python" in
↪  token.lower())

# Extraire les embeddings contextuels
python_embedding1 = outputs1.last_hidden_state[0, python_index1,
↪  :].numpy()
python_embedding2 = outputs2.last_hidden_state[0, python_index2,
↪  :].numpy()

# Calculer la similarité cosinus
similarity = cosine_similarity([python_embedding1],
↪  [python_embedding2])[0][0]

print(
    f"Similarité entre 'python' (programmation) et 'python' (animal) :
    ↪  {similarity:.4f}"
)
```

Code 6 : Résultat de l'exemple. python

```
Similarité entre 'python' (programmation) et 'python' (animal) : 0.5223
```

3 Les transformers

3.1 Le mécanisme d'attention

Le mécanisme d'attention est le mécanisme le plus important des transformers. Ce mécanisme permet de capturer les connexions et les corrélations entre les tokens, de manière mathématique, permettant ensuite de prédire le token suivant. Ce mécanisme a été développé et rendu célèbre par la publication « Attention Is All You Need » réalisée par des chercheurs de Google [5]. Nous présentons ici la théorie derrière ce modèle.

Concrètement, le mécanisme repose sur trois ensembles de vecteurs permettant d'attribuer des poids aux différents tokens de la séquence d'entrée selon leur importance dans la tâche en cours. On a les requêtes Q, les clés K et les valeurs V. Les requêtes vont représenter mathématiquement ce que les tokens cherchent à obtenir des autres, les clés décrivent les informations que le token peut offrir et la valeur est une représentation (embedding) contextualisée d'un token. Il est commode de regrouper chaque ensemble dans une même matrice qu'on notera encore respectivement Q, K et V. Chacune de ces matrices est associée à une matrice de poids notée W_Q, W_K et W_V respectivement. Ce sont ces matrices qui seront entraînées et qui permettront à Q, K et V de réaliser leur rôle. Plus formellement, pour une entrée $X \in \mathbb{R}^{n \times d_{\text{modèle}}}$, où n est le nombre de tokens, et $d_{\text{modèle}}$ la dimension de représentation des tokens, on a les relations suivantes.

$$Q = XW_Q, \quad K = XW_K \quad \text{et} \quad V = XW_V,$$

où $W_Q, W_K \in \mathbb{R}^{n \times d_k}$ et $W_V \in \mathbb{R}^{n \times d_v}$ avec d_k et d_v les dimensions des vecteurs requêtes et clés, et des vecteurs valeurs respectivement.

Ainsi, chaque ligne de Q, K et V correspond à la projection d'un token dans l'espace des requêtes, clés ou valeurs.

Pour chaque token, l'idée est de mesurer la similarité entre sa requête et les clés des autres tokens de la séquence d'entrée. On a la relation :

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

Plus en détails, le terme $QK^\top$ est un score qui mesure la similarité entre la requête d'un token et la clé d'un autre grâce au produit matriciel. La division par $\sqrt{d_k}$ est une simple normalisation pour éviter que ces scores ne deviennent trop grands. L'application de softmax permet de convertir ces valeurs en probabilités, en ramenant les résultats dans l'intervalle $[0, 1]$. Enfin, le produit par V permet d'obtenir la nouvelle représentation de chaque token de l'entrée. Lorsque l'attention est calculée sur l'ensemble de l'entrée, on parle plus précisément de self-attention.

On voit que ce calcul d'attention est directement lié aux matrices de poids, puisque ces dernières donne les valeurs des matrices Q, K et V. Mais en réalité, utiliser une seule initialisation des matrices de poids W_Q, W_K et W_V ne permet pas forcément de correctement capturer la richesse du langage de l'entrée pour chaque token, comme la sémantique, la grammaire, etc. C'est pourquoi plusieurs initialisations sont effectuées en

parallèle dans les modèles utilisés en pratique. Plus précisément, on parle de « têtes » (*heads*, en anglais). Formellement, on note :

$$\text{head}_i = \text{Attention}\left(QW_Q^{(i)}, KW_K^{(i)}, VW_V^{(i)}\right),$$

où chaque $\left(W_\eta^{(i)}\right)_{\eta\in\{Q,K,V\}}$ représente une initialisation de matrices de poids différentes, spécifiques à la tête i de dimensions d_k et d_v généralement choisies de telles sorte que $d_{\text{modèle}} = h \cdot d_k$, si h est le nombre de têtes.

Une fois calculées, les sorties de toutes les têtes sont concaténées :

$$\text{MultiHead}\,(Q, K, V) = \text{concat}\,(\text{head}_1, \ldots, \text{head}_h)\, W_O,$$

où $W_O \in \mathbb{R}^{(h\cdot d_v)\times d_{\text{modèle}}}$ est une nouvelle matrice de poids (elle aussi entraînée) qui rassemble l'information des différentes têtes. L'ensemble du processus décrit jusqu'à maintenant est résumé en figure 1.

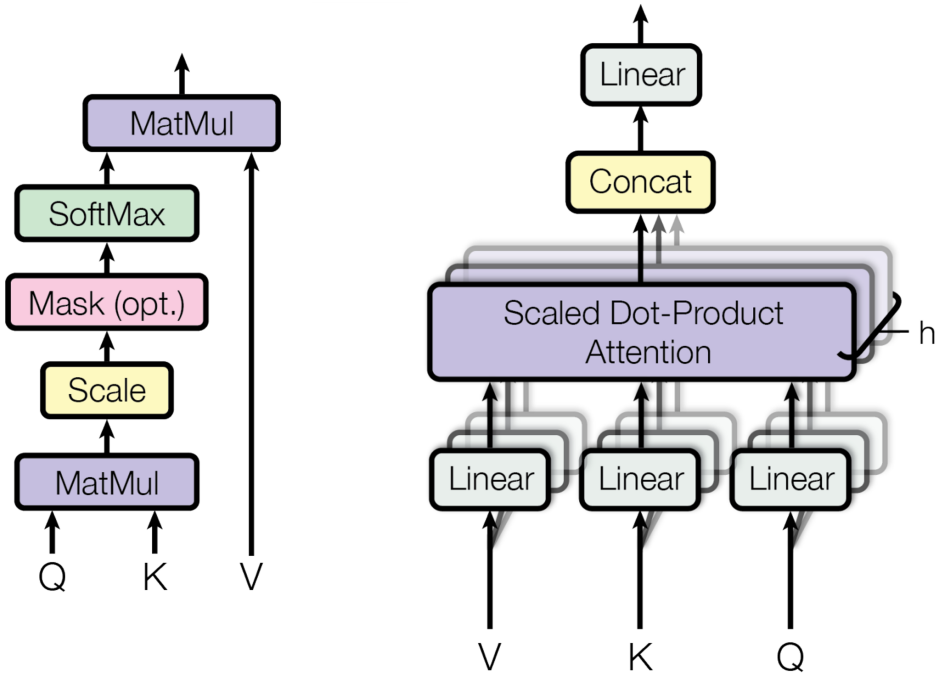

FIGURE 1 – À gauche : calcul du self-attention. À droite : calcul du multi-head attention, composé de plusieurs couches de self-attention fonctionnant en parallèle [5].

3.2 Architecture globale du transformer

Maintenant que le mécanisme au cœur du transformer a été présenté, il est possible de comprendre le fonctionnement complet d'un transformer. Dans sa version originale, un transformer est composé de deux parties principales : un encodeur et un décodeur. Mais avant de présenter ces deux éléments, un point théorique doit être traité, celui de la position des tokens.

3.2.1 Prise en compte de la position

Les informations de position (ou « positional encodings ») permettent d'incorporer l'ordre des tokens dans la séquence, car le mécanisme d'attention du transformer, qui traite tous les tokens en parallèle, est intrinsèquement invariant à l'ordre. Pour y remédier, on calcule un vecteur de positional encoding pour chaque position dans la séquence, puis on l'ajoute élément par élément à l'embedding du token correspondant pour conserver cette information au sein de l'embedding.

Dans la version originale du transformer, ces vecteurs sont déterministes et définis à l'aide de fonctions sinusoïdales. Pour une position κ et pour la dimension i (en commençant à $i = 0$), les formules sont :

$$\text{PositionalEncoding}(\kappa, 2i) = \sin\left(\frac{\kappa}{10\,000^{\frac{2i}{d_{\text{modèle}}}}}\right),$$

$$\text{PositionalEncoding}(\kappa, 2i+1) = \cos\left(\frac{\kappa}{10\,000^{\frac{2i}{d_{\text{modèle}}}}}\right).$$

Ces formules génèrent un vecteur de dimension $d_{\text{modèle}}$ pour chaque position, garantissant que chaque token reçoit une information de position unique et que les relations entre positions (notamment les distances relatives) sont encodées de manière continue.

L'intégration se fait simplement par addition :

$$\text{EmbeddingFinal} = \text{Embedding} + \text{PositionalEncoding},$$

où l'opération d'addition est effectuée terme à terme. Cette fusion additive permet au modèle de conserver la dimension $d_{\text{modèle}}$ tout en combinant l'information sémantique issue de l'embedding et l'information positionnelle apportée par le positional encoding. Cet EmbeddingFinal constituera notamment l'entrée de l'encodeur du transformer.

En pratique, cette somme permet aux couches subséquentes du transformer d'apprendre à exploiter conjointement ces deux types d'information, compensant ainsi l'absence de traitement séquentiel propre qu'on retrouve dans les RNN.

3.2.2 Encodeur

L'encodeur va chercher à capturer les relations contextuelles syntaxiques et sémantiques de l'ensemble de la séquence d'entrée. Pour cela, il est constitué de N_e couches identiques, comportant chacune deux sous-couches.

La première sous-couche applique un mécanisme de multi-head attention présenté précédemment.

La seconde sous-couche est un feed-forward positionnelle. Son rôle est d'introduire une transformation non-linéaire et de permettre au modèle de capturer des interactions complexes au sein de chaque représentation de token, indépendamment des autres positions.

Concrètement, pour chaque token, la transformation est définie par :

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

où x est la représentation d'un token issue de la sous-couche d'attention. Cette représentation est ensuite projetée dans un espace de dimension supérieure par la multiplication par la matrice W_1 (de dimensions $d_{\text{modèle}} \times d_{\text{ff}}$), où d_{ff} est généralement beaucoup plus grand que $d_{\text{modèle}}$ (par exemple, $d_{\text{ff}} = 2048$ quand $d_{\text{modèle}} = 512$ dans l'architecture originale du transformer). Un biais b_1 est ajouté pour permettre une plus grande flexibilité.

Ensuite, la fonction d'activation non-linéaire, ici la ReLU ($\text{ReLU}(x) = \max(0, x)$), est appliquée. Cette non-linéarité permet de modéliser des interactions complexes et de rendre le réseau plus expressif.

La sortie de la ReLU est ensuite projetée de nouveau par la matrice W_2 (de dimensions $d_{\text{ff}} \times d_{\text{modèle}}$) et un biais b_2 est ajouté pour ramener la dimension du vecteur à $d_{\text{modèle}}$, afin qu'il soit compatible avec la dimension de l'entrée du transformer.

Enfin, pour faciliter l'apprentissage et stabiliser l'entraînement, une connexion résiduelle est appliquée. Cela signifie que l'entrée originale x est additionnée à la sortie du réseau feed-forward, puis le résultat est normalisé via une normalisation de couche (notée LayerNorm). La formule complète devient alors :

$$\text{Output}_{\text{encodeur}} = \text{LayerNorm}\left(x + \text{FFN}(x)\right).$$

3.2.3 Décodeur

Le décodeur, quant à lui, est chargé de générer la séquence de sortie de manière auto-régressive. Il se compose de N_d couches identiques, chacune intégrant trois sous-couches successives. Dans un premier temps, une sous-couche de self-attention masquée permet à chaque token généré d'interagir uniquement avec les tokens précédents (au sens large), afin de respecter l'ordre chronologique de la génération (d'où le terme « masqué »). Pour ce faire, la self-attention masquée est calculée comme suit :

$$\text{Attention}_{\text{mask}}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right) V,$$

où M est un masque triangulaire tel que :

$$M_{ij} = \begin{cases} 0 & \text{si } j \leq i \\ -\infty & \text{si } j > i \end{cases},$$

empêchant ainsi l'accès aux informations futures.

Ensuite, une sous-couche d'attention encodeur-décodeur permet au décodeur d'intégrer les informations contextuelles extraites par l'encodeur. Dans cette étape, la requête Q' est dérivée de la sortie de la self-attention masquée du décodeur, tandis que les clés K' et les valeurs V' proviennent de la représentation Z produite par l'encodeur, via les transformations :

$$K' = ZW'_K, \quad V' = ZW'_V.$$

L'attention encodeur-décodeur est alors calculée par :

$$\text{Attention}(Q', K', V') = \text{softmax}\left(\frac{Q'\left(K'\right)^\top}{\sqrt{d_k}}\right) V',$$

ce qui permet d'extraire les informations pertinentes de la séquence d'entrée pour guider la génération du token suivant.

Enfin, chaque couche du décodeur inclut également une sous-couche feed-forward identique à celle de l'encodeur, appliquée indépendamment à chaque position. La formule de la sortie est la même que pour l'encodeur :

$$\text{Output}_{\text{décodeur}} = \text{LayerNorm}\left(x + \text{FFN}(x)\right).$$

Lors de la dernière génération de la séquence, c'est le vecteur de sortie correspondant à la dernière position (ou au dernier token généré) qui est utilisé pour prédire le token suivant, et le cycle se répète. L'ensemble du processus est résumé par le schéma présenté en figure 2.

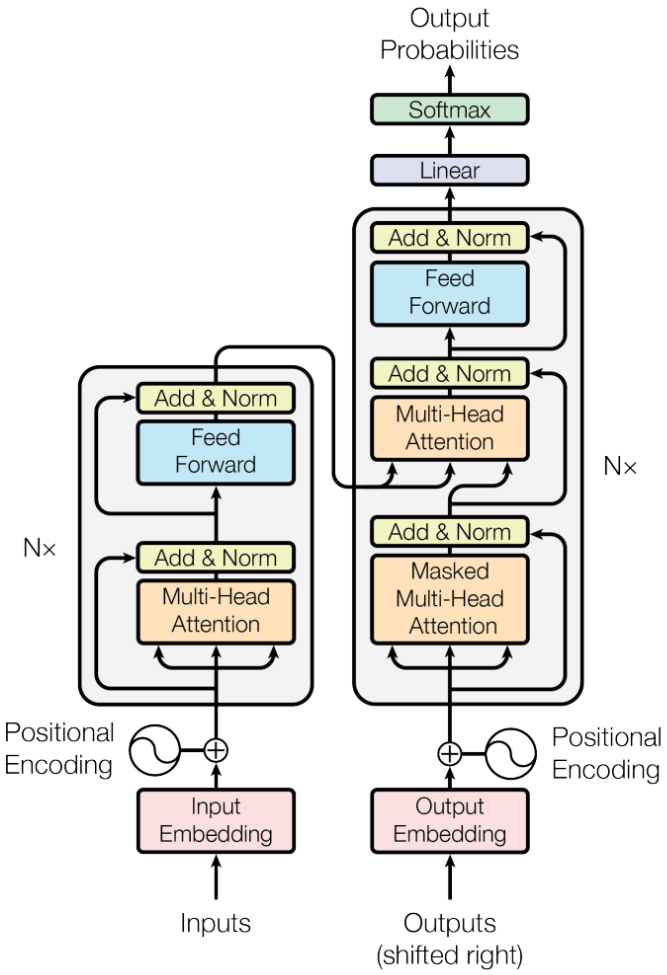

FIGURE 2 – Architecture du modèle original de transformer [5].

3.3 Différences avec Mamba

On traite dans cette partie des différences entre le transformer et d'un autre modèle célèbre, Mamba, qu'on présente maintenant.

3.3.1 Mamba : Une alternative prometteuse aux transformers en NLP

Les modèles de traitement du langage naturel ont été dominés par les architectures transformer malgré leur complexité quadratique ($\mathcal{O}(N^2)$) due au mécanisme d'attention. L'article fondateur « Mamba : Linear-Time Sequence Modeling with Selective State Spaces » [13] introduit une approche radicalement différente basée sur les State Space Models (SSM), offrant une complexité linéaire ($\mathcal{O}(N)$) tout en maintenant, voire surpassant, les performances des transformers sur de nombreuses tâches. En effet, les SSM sont une approche issue des systèmes dynamiques continus, largement utilisés en traitement du signal et en systèmes de contrôle.

3.3.2 Principes mathématiques des SSM

Les SSM sont une classe de modèles inspirés des systèmes dynamiques continus dont l'idée principale est de représenter une séquence $x(t)$ comme un système dynamique latent $h(t)$, qui est ensuite discrétisé pour une implémentation numérique efficace. Leur structure générale est montrée en figure 3.

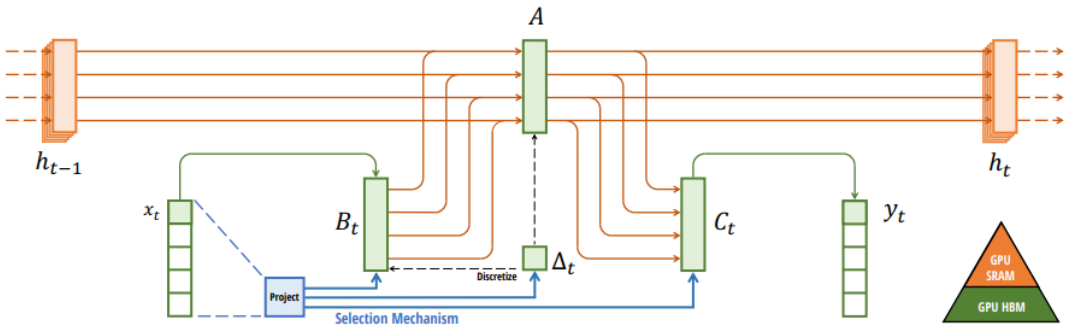

FIGURE 3 – Structure d'un modèle SSM standard [13].

Formulation continue des SSM

Le modèle SSM continu est défini par les équations différentielles suivantes :

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}t}h(t) &= Ah(t) + Bx(t) \\ y(t) &= Ch(t) \end{aligned}$$

où :

- $h(t) \in \mathbb{R}^N$ est l'état caché (vecteur de dimension N).

- $x(t)$ est l'entrée du système (par exemple, un signal temporel).
- $y(t)$ est la sortie du système.
- A, B, C sont des matrices paramétriques définissant la dynamique du système :
  - $A \in \mathbb{R}^{N \times N}$ (matrice d'état).
  - $B \in \mathbb{R}^{N \times 1}$ (matrice d'entrée).
  - $C \in \mathbb{R}^{1 \times N}$ (matrice de sortie).

Discrétisation du SSM

Pour un traitement numérique, on doit convertir cette équation continue en une version discrète avec un pas de temps Δ. La méthode utilisée dans l'article est la discrétisation par maintien d'ordre zéro (*zero-order hold* ou ZOH) [13], qui donne les équations suivantes :

$$\begin{aligned} \boldsymbol{A}_d &= e^{\Delta \boldsymbol{A}} \\ \boldsymbol{B}_d &= \boldsymbol{A}^{-1}(e^{\Delta \boldsymbol{A}} - \boldsymbol{I})\boldsymbol{B}\Delta \end{aligned}$$

Ces nouvelles matrices discrètes ($\boldsymbol{A}_d$, $\boldsymbol{B}_d$) permettent d'écrire la dynamique en version discrète :

$$\begin{aligned} h_t &= \boldsymbol{A}_d h_{t-1} + \boldsymbol{B}_d x_t \\ y_t &= \boldsymbol{C} h_t \end{aligned}$$

3.3.3 Architecture de Mamba

Mamba améliore les SSM en introduisant une dynamique sélective et en optimisant l'implémentation pour une exécution efficace sur GPU.

Sélection dynamique des états

Les modèles SSM classiques utilisent une évolution d'état fixe, mais Mamba introduit un mécanisme de sélection qui permet aux paramètres du modèle d'évoluer en fonction des entrées. Cela se traduit par les équations modifiées suivantes :

$$\begin{aligned} \boldsymbol{h}_t &= \boldsymbol{A}(\boldsymbol{h}_{t-1})\boldsymbol{h}_{t-1} + \boldsymbol{B}(\boldsymbol{h}_{t-1})\boldsymbol{x}_t \\ \boldsymbol{y}_t &= \boldsymbol{C}\boldsymbol{h}_t \end{aligned}$$

Ici, les matrices $\boldsymbol{A}$ et $\boldsymbol{B}$ ne sont plus constantes mais deviennent des fonctions dépendantes de l'état précédent $\boldsymbol{h}_{t-1}$, ce qui rend le modèle plus flexible pour capturer des relations complexes dans les données.

3.3.4 Synthèse

Mamba traite les séquences (entrées) comme un flux séquentiel : pour chaque token (par exemple, un mot), il met à jour dynamiquement son état caché (mémoire contextuelle) via un SSM sélectif qui filtre les informations utiles. La sortie est générée en combinant cet état avec une projection linéaire ($C \cdot h_t$), permettant des prédictions précises sans calculer toutes les interactions entre tokens. Une table synthétique résumant les différences entre Mamba et les transformers est donnée en table 1.

Critère	**Mamba**	**Transformers**
Mécanisme de base	State Space Models sélectifs	Multi-head attention
Complexité	$\mathcal{O}(n)$ [14]	$\mathcal{O}(n^2)$
Traitement	Récurrent	Parallèle
Mémoire	Constante (optimisée)	Élevée (stocke toutes les interactions)
Séquences longues	Excellente scalabilité [14]	Limité par mémoire GPU
Sélection	Contextuelle dynamique	Uniforme
Usage typique	Génomes, ECG, livres	Traduction, texte court

TABLE 1 – Comparaison entre Mamba et les transformers.

Ainsi, Mamba et les transformers représentent deux approches complémentaires pour le traitement des séquences. Mamba excelle par son efficacité linéaire, idéale pour les longues séquences avec une optimisation matérielle inégalée. Les transformers, grâce à leur mécanisme d'attention globale, restent incontournables pour les tâches nécessitant une compréhension contextuelle profonde.

4 Mini-projet : Détecter le ton d'un avis de film

Pour illustrer les notions abordées précédemment, on cherche à créer un modèle capable de déterminer si un avis de film adopte un ton plutôt négatif ou positif. On cherchera donc à effectuer une classification binaire.

On utilisera pour cela une base de données fournie par Hugging Face [15]. Cette base de données regroupe des avis de film issus du site allocine.fr. On entraînera donc un modèle en français. Elle contient 160 000 avis pour l'entraînement, 20 000 pour la validation et 20 000 pour le test. La quantité de données est donc suffisante pour entraîner correctement un modèle qu'on définit maintenant.

Le modèle utilisé est le modèle `CamemBERT`. Il s'agit d'un modèle pré-entraîné pour le français. Plus précisément, on utilisera le modèle `CamembertForSequenceClassification`, spécialisé dans la classification. Il est basé sur le modèle de transformer, il utilise donc les mécanismes de multi-head attention développés précédemment.

Il est accompagné de son propre tokenizer, `CamembertTokenizer`, permettant de transformer les entrées en données numériques adaptées au modèle en utilisant le `Byte-Pair Encoding`.

Les définitions et paramètres du modèle sont donnés en Code 7.

Code 7 : Définition et paramètres du modèle utilisé. python

```python
# Chargement des données
data_files = {
    "train": "databases/train.jsonl",
    "validation": "databases/val.jsonl",
    "test": "databases/test.jsonl",
}
dataset = load_dataset("json", data_files=data_files)

# Chargement du tokenizer
tokenizer = CamembertTokenizer.from_pretrained("camembert-base")


def tokenize_function(example):
    return tokenizer(
        example["review"], truncation=True, padding="max_length",
        ↪  max_length=128
    )


tokenized_datasets = dataset.map(tokenize_function, batched=True)

model = CamembertForSequenceClassification.from_pretrained("./results",
↪  num_labels=2)

training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
    fp16=True,
    no_cuda=False,
)

trainer = Trainer(
```

```
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    tokenizer=tokenizer,
)
```

Après entraînement, on obtient un modèle avec une précision de 95,91 % de précision dans la classification binaire, ce qui démontre son efficacité. La matrice de confusion du modèle est donnée en figure 4.

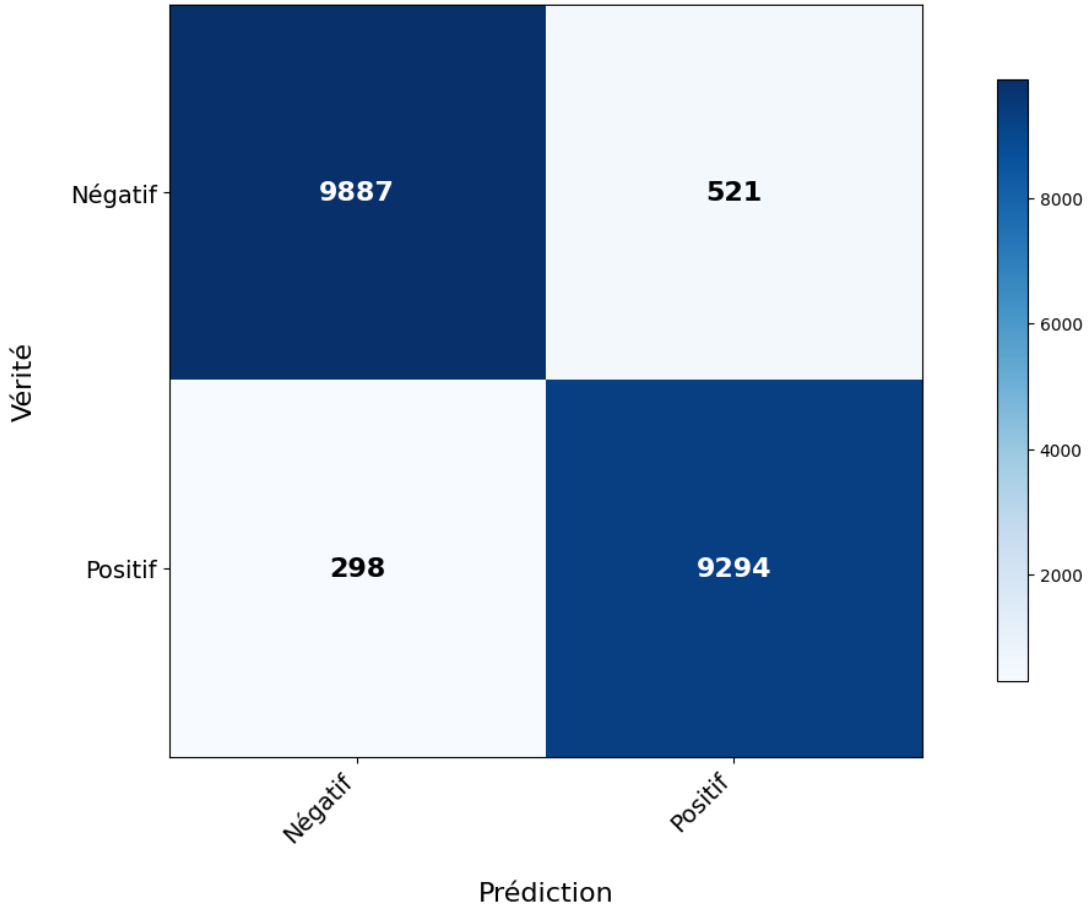

FIGURE 4 – Matrice de confusion de notre modèle sur les données de test.

Le code complet est donné dans l'archive fournie. Un GPU a été utilisé pour l'entraînement (en utilisant `CUDA`, notamment). La référence de la carte graphique utilisée est la suivante : `NVIDIA GeForce RTX 4060 Laptop GPU`.

5 Conclusion

Les transformers ont révolutionné le domaine de l'intelligence artificielle en introduisant un nouveau paradigme pour le traitement des données séquentielles. Le mécanisme d'attention a surpassé les limites des architectures traditionnelles en modélisant efficacement les dépendances à longue distance. L'évolution des techniques d'embedding, du simple `Word2Vec` aux représentations contextuelles de `BERT`, illustre la progression vers des modèles capables de saisir les subtilités sémantiques du langage. Notre mini-projet sur la classification du ton des avis de films, atteignant une précision de 95,91%, démontre l'efficacité pratique de ces architectures.

Cependant, la complexité quadratique des transformers reste un défi pour le traitement de très longues séquences, laissant place à des innovations comme Mamba avec sa complexité linéaire. À l'avenir, nous pouvons anticiper que ces deux approches évolueront en parallèle, chacune trouvant sa niche selon les contraintes spécifiques des tâches à accomplir : les transformers pour une compréhension profonde du contexte, et les modèles comme Mamba pour l'efficacité sur de longues séquences. Cette complémentarité promet de continuer à repousser les frontières de l'intelligence artificielle dans les années à venir.

CENTRALE LYON

6 Bibliographie

[1] Sepp Hochreiter et Jürgen Schmidhuber. « Long short-term memory ». In : *Neural Computation* 9.8 (1997).

[2] Yann LeCun et al. « Gradient-based learning applied to document recognition ». In : *Proceedings of the IEEE*. T. 86. 11. 1998.

[3] Kaiming He et al. « Deep Residual Learning for Image Recognition ». In : *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[4] Dzmitry Bahdanau, Kyunghyun Cho et Yoshua Bengio. « Neural Machine Translation by Jointly Learning to Align and Translate ». In : *International Conference on Learning Representations (ICLR)*. 2015.

[5] Ashish Vaswani et al. « Attention is All You Need ». In : *Advances in Neural Information Processing Systems*. T. 30. 2017, p. 5998-6008.

[6] Ian Goodfellow, Yoshua Bengio et Aaron Courville. *Deep Learning*. MIT Press, 2016.

[7] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[8] Kurt Hornik, Maxwell Stinchcombe et Halbert White. « Multilayer feedforward networks are universal approximators ». In : *Neural Networks* 2.5 (1989), p. 359-366.

[9] Jacob Devlin et al. « BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding ». In : *Proceedings of NAACL-HLT* (2019).

[10] Tom B. Brown et al. « Language models are few-shot learners ». In : *Advances in Neural Information Processing Systems* 33 (2020).

[11] Alexey Dosovitskiy et al. « An image is worth 16x16 words : Transformers for image recognition at scale ». In : *International Conference on Learning Representations (ICLR)*. 2020.

[12] J. R. Firth. *Studies in Linguistic Analysis*. Wiley-Blackwell, 1957, p. 11.

[13] Albert Gu et Tri Dao. « Mamba : Linear-Time Sequence Modeling with Selective State Spaces ». In : *arXiv preprint arXiv :2312.00752* (2023). Consulté le 27 mars 2025. URL : `https://minjiazhang.github.io/courses/fall24-resource/Mamba.pdf`.

[14] Jinhao Li et al. « MARCA : Mamba Accelerator with ReConfigurable Architecture ». In : *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. New York, NY, USA, oct. 2024, 1 et 6.

[15] Théophile Blard. *Allociné : un jeu de données de critiques de films*. `https://huggingface.co/datasets/tblard/allocine`. Consulté le 26 mars 2025. 2021.