
CONSTRUIRE
UN AGENT LLM
UN LIVRE DE
PATRICE HUETZ

AgentLLM Narratif

Maîtriser les agents LLM en Python

Patrice Huetz

patrice-huetz.fr

patrice-huetz.fr · contact@patrice-huetz.fr

CHAPITRE 1

Comprendre les Large Language Models

Un mardi soir, dans un café près du campus universitaire...

Lina fixait son écran, perplexe. Elle venait de passer trois heures à interagir avec ChatGPT, lui demandant d'expliquer du code, de générer des tests, de suggérer des refactorisations. Les résultats étaient tantôt brillants, tantôt absurdes. À un moment, le modèle avait produit une solution élégante à un problème de concurrence qu'elle n'arrivait pas à résoudre depuis des jours. L'instant d'après, il affirmait avec une assurance déconcertante qu'une bibliothèque inexistante était « la meilleure solution pour ce cas d'usage ».

— « Comment ça peut être si intelligent et si stupide à la fois ? » murmura-t-elle en repoussant son ordinateur.

Son ami Marcus, doctorant en machine learning, s'assit à côté d'elle avec son café. Il avait entendu cette question des dizaines de fois — de la part d'étudiants, de collègues, même de professeurs chevronnés. C'était LA question que tout le monde se posait face aux LLMs.

— « Tu sais comment ça fonctionne, un LLM ? » demanda-t-il.

Lina haussa les épaules avec une moue frustrée.

— « Vaguement. Des réseaux de neurones, beaucoup de données, quelque chose avec l'attention... Mais honnêtement, ça ressemble à de la magie noire. Une magie noire qui ment parfois avec beaucoup d'aplomb. »

Marcus sourit. Il connaissait ce sentiment d'émerveillement mêlé de méfiance. Pendant des mois, il avait lui aussi traité ces modèles comme des boîtes noires, acceptant leurs réponses sans vraiment comprendre d'où elles venaient. Puis il avait plongé dans les articles de recherche, les implémentations open-source, les visualisations d'attention. Et tout avait changé.

— « C'est un bon début. Mais si tu veux vraiment construire des outils qui utilisent ces modèles — pas juste les subir, mais les maîtriser — tu dois comprendre ce qu'ils sont vraiment. Pas la version marketing. La vraie mécanique. Les forces. Les faiblesses. Les raisons profondes de leurs comportements. »

Il sortit un carnet et un stylo, dessina rapidement un schéma.

— « Laisse-moi te raconter une histoire. Elle commence en 2017, dans les bureaux de Google Brain, avec un article qui allait bouleverser tout le domaine de l'intelligence artificielle... »

Table des Matières

1.1 Une Brève Histoire des Modèles de Langage

Pour comprendre pourquoi les LLMs actuels sont si puissants — et pourquoi ils ont des limitations spécifiques — il faut d'abord comprendre ce qui existait avant eux. L'histoire des modèles de langage est une histoire de compromis : entre expressivité et efficacité, entre mémoire et calcul, entre généralité et spécialisation. Chaque génération de modèles a résolu certains problèmes tout en en créant d'autres, jusqu'à ce qu'une innovation fondamentale — le Transformer — change les règles du jeu.

1.1.1 L'Ère Statistique : Les Modèles N-grammes

Pendant des décennies, le traitement automatique du langage naturel (NLP) reposait sur des approches purement statistiques. L'idée fondamentale était simple : si nous pouvons compter combien de fois certaines séquences de mots apparaissent ensemble dans un grand corpus de texte, nous pouvons prédire quel mot viendra probablement après une séquence donnée.

Les **modèles n-grammes** incarnaient cette philosophie. Un modèle bigramme (n=2) prédisait le mot suivant uniquement en fonction du mot précédent. Un modèle trigramme (n=3) utilisait les deux mots précédents. Et ainsi de suite.

Prenons un exemple concret. Supposons que nous ayons entraîné un modèle 5-grammes sur un corpus de textes français. Face à la séquence "le chat dort sur le", le modèle consulterait ses tables de fréquences :

- "le chat dort sur le **canapé**" : vu 1,247 fois dans le corpus
- "le chat dort sur le **tapis**" : vu 892 fois
- "le chat dort sur le **lit**" : vu 756 fois
- "le chat dort sur le **toit**" : vu 23 fois

Le modèle prédirait donc "canapé" avec une probabilité proportionnelle à ces fréquences. Simple, efficace... et profondément limité.

Le problème fondamental des n-grammes tient en un mot : **contexte**. Ces modèles ne peuvent "voir" qu'une fenêtre fixe de mots — typiquement 3 à 5. Or, le langage humain regorge de dépendances à longue distance. Considérez cette phrase :

> *"Le développeur qui avait passé trois ans à travailler sur ce projet, malgré les difficultés rencontrées avec l'équipe de management et les contraintes budgétaires imposées par la direction,* ***était*** *finalement satisfait du résultat."*

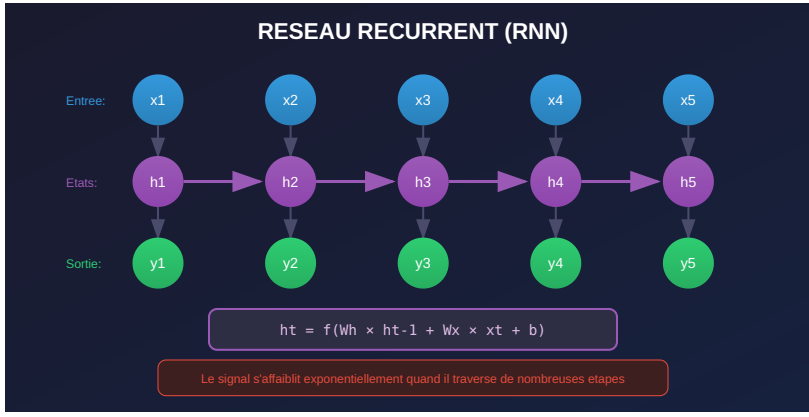
Le verbe “était” doit s’accorder avec “Le développeur” — un mot situé à plus de trente tokens de distance ! Aucun modèle n-gramme pratique ne pouvait capturer cette relation. C’était comme essayer de comprendre un roman en ne lisant que des phrases isolées, sans jamais voir les connexions entre les personnages et les événements.

Aspect	Modèles N-grammes	Limitation
Mémoire	Fenêtre fixe (3-5 mots)	Perte du contexte lointain
Taille	Croissance exponentielle	V^n entrées pour vocabulaire V
Généralisation	Aucune	Ne reconnaît que ce qu’il a vu exactement
Données rares	Problématique	“smoothing” nécessaire mais imparfait

1.1.2 Les Réseaux Récurrents : Une Promesse Partiellement Tenue

Dans les années 2010, une nouvelle approche émergea : les réseaux de neurones récurrents (RNN). L’idée était élégante et biologiquement inspirée. Au lieu de regarder une fenêtre fixe de mots, le réseau maintiendrait un **état caché** — une sorte de “mémoire de travail” — qui se propagerait d’un mot au suivant.

Imaginez un lecteur humain parcourant un texte. À chaque mot, il ne repart pas de zéro : il accumule une compréhension du contexte, des personnages, du ton. Les RNN tentaient de reproduire ce mécanisme. L’état caché à l’étape t dépendait de l’entrée actuelle ET de l’état caché à l’étape t-1, créant une chaîne théoriquement capable de transporter l’information sur des distances arbitraires.



Architecture RNN

Les variantes comme LSTM (Long Short-Term Memory) et GRU (Gated Recurrent Unit) ajoutèrent des mécanismes de “portes” pour mieux contrôler le flux d’information. Ces architectures connurent un succès considérable et dominèrent le NLP pendant plusieurs années.

Cependant, deux problèmes fondamentaux persistaient :

Le gradient évanescent : Lors de l’entraînement, les signaux d’erreur doivent se propager à travers la chaîne de récurrence. À chaque étape, ils sont multipliés par des poids, et si ces poids sont inférieurs à 1 (ce qui est souvent le cas), le signal diminue exponentiellement. Après 50 ou 100 étapes, il devient pratiquement imperceptible. Le réseau « oublie » donc ce qu’il a vu au début de la séquence.

La séquentialité imposée : Par construction, un RNN doit traiter les mots un par un, dans l’ordre. Il est impossible de calculer h_3 avant d’avoir calculé h_2, qui lui-même dépend de h_1. Cette dépendance séquentielle empêche toute parallélisation efficace. Sur les GPU modernes, conçus pour exécuter des milliers d’opérations simultanément, cette limitation était catastrophique pour les temps d’entraînement.

Critère	N-grammes	RNN/LSTM	Impact pratique
Contexte	~5 mots	~100-500 mots (théorique)	LSTM meilleur mais imparfait
Parallélisation	Excellente	Impossible	Entraînement 10-100x plus lent
Mémoire GPU	Faible	Modérée	LSTM plus gourmand
Dépendances longues	Aucune	Difficiles	Gradient vanishing persiste

1.1.3 Juin 2017 : « Attention Is All You Need »

Le 12 juin 2017, une équipe de huit chercheurs chez Google publia un article au titre provocateur : « **Attention Is All You Need** ». Parmi eux, des noms qui allaient devenir légendaires dans le domaine : Ashish Vaswani, Noam Shazeer, Niki Parmar, et Jakob Uszkoreit.

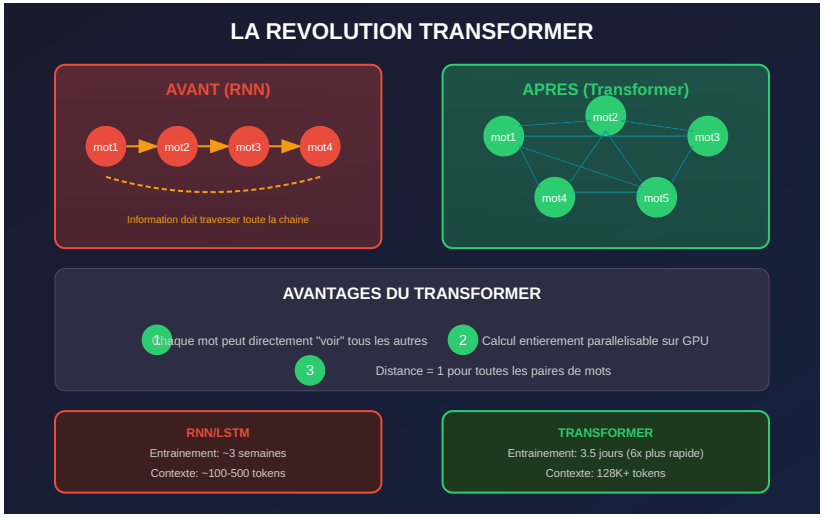
L'article proposait une architecture radicalement différente appelée **Transformer**. L'idée centrale tenait en une question audacieuse : et si on abandonnait complètement la récurrence ? Et si, au lieu de traiter les mots séquentiellement, on les traitait **tous en parallèle**, en utilisant uniquement des mécanismes d'attention pour capturer les relations entre eux ?

Architecture Transformer

L'intuition derrière cette approche était profonde. Dans un RNN, l'information doit "voyager" à travers de nombreuses étapes pour connecter des mots éloignés. Chaque étape introduit du bruit et de l'atténuation.

Mais que se passerait-il si chaque mot pouvait directement “regarder” tous les autres mots, sans intermédiaire ?

C’est exactement ce que fait le mécanisme d’attention : il permet à chaque position dans la séquence de calculer une connexion directe avec chaque autre position. La distance entre deux mots n’a plus d’importance — ils sont tous à “un saut d’attention” l’un de l’autre.



La Révolution Transformer

Les résultats furent spectaculaires. Sur la tâche de traduction anglais-allemand du benchmark WMT 2014, le Transformer atteignit un score BLEU de 28.4, surpassant tous les modèles précédents de plus de 2 points — une marge énorme dans ce domaine. Plus impressionnant encore : l’entraînement ne prenait que 3.5 jours sur 8 GPUs, contre des semaines pour les meilleurs modèles RNN.

Métrique	LSTM (meilleur)	Transformer	Amélioration
BLEU (EN→DE)	25.8	28.4	+10%
BLEU (EN→FR)	41.0	41.8	+2%
Temps d’entraînement	~3 semaines	3.5 jours	**~6x plus rapide**
Paramètres	~200M	65M	3x moins

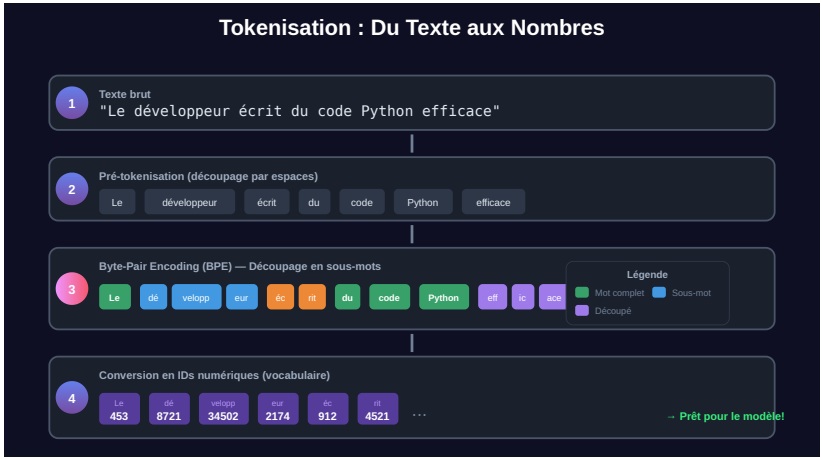
Un an plus tard, Google dévoilait **BERT** (Bidirectional Encoder Representations from Transformers) et OpenAI présentait **GPT** (Generative Pre-trained Transformer). L’ère des Large Language Models venait de commencer, et rien ne serait plus jamais pareil.

1.2 L’Anatomie d’un Transformer

Maintenant que nous comprenons le contexte historique, plongeons dans les détails techniques. Un Transformer est composé de plusieurs éléments interconnectés, chacun jouant un rôle précis dans la transformation du texte brut en représentations riches de sens.

1.2.1 La Tokenisation : Découper le Langage

Avant même d’entrer dans le réseau de neurones, le texte doit être converti en nombres. Cette étape, appelée **tokenisation**, est plus subtile et plus importante qu’il n’y paraît. Les choix faits à ce niveau ont des répercussions profondes sur les performances, les coûts, et même les biais du modèle.



Processus de Tokenisation

Le problème du vocabulaire

Une approche naïve consisterait à attribuer un identifiant unique à chaque mot du dictionnaire. Mais cette stratégie se heurte à plusieurs obstacles :

1. **La taille du vocabulaire** : Le français compte environ 100,000 mots courants, l'anglais environ 170,000. Mais avec les noms propres, les termes techniques, le jargon internet, les nouvelles créations... le vocabulaire effectif est pratiquement infini.
2. **Les mots rares** : Même avec un vocabulaire de 100,000 entrées, de nombreux mots ne seront vus qu'une ou deux fois pendant l'entraînement. Le modèle n'aura pas assez d'exemples pour apprendre leur signification.
3. **Les langues agglutinantes** : En allemand, finnois ou turc, les mots peuvent être composés de nombreux morphèmes. "Donaudampfschifffahrtsgesellschaftskapitän" (capitaine de la compagnie de navigation à vapeur du Danube) est un mot allemand parfaitement valide.

La solution : Byte-Pair Encoding (BPE)

La solution moderne est le **Byte-Pair Encoding**, un algorithme de compression adapté à la tokenisation. L'idée est de construire un vocabulaire de "sous-mots" — des fragments qui peuvent être combinés pour former n'importe quel mot.

L'algorithme fonctionne ainsi : 1. Commencer avec un vocabulaire contenant uniquement les caractères individuels 2. Compter toutes les paires de tokens adjacents dans le corpus 3. Fusionner la paire la plus fréquente en un nouveau token 4. Répéter jusqu'à atteindre la taille de vocabulaire désirée

Après entraînement sur un grand corpus, le vocabulaire contient : - Des caractères individuels (pour gérer n'importe quelle entrée) - Des morphèmes communs ("ing", "tion", "pré", "anti") - Des mots fréquents entiers ("the", "is", "de", "le") - Des fragments de mots moins courants

Tokenisation BPE en Action

Implications pratiques pour les développeurs

Cette mécanique de tokenisation a des conséquences directes sur l'utilisation des LLMs :

Impact	Description	Conseil pratique
Coût	Les API facturent par token	Noms de variables courts = moins cher
Limite de contexte	128K tokens ≠ 128K caractères	Un fichier de 10KB peut consommer 3-5K tokens
Langues	Non-anglais = plus de tokens	Budget 30-50% de tokens en plus pour le français
Code	Syntaxe verbale = plus de tokens	`calculateTotalAmountWithTax` = ~8 tokens
Comptage	LLMs comptent mal les caractères	"Combien de 'r' dans strawberry ?" → souvent faux

Ce dernier point mérite une explication. Quand vous demandez à un LLM de compter les lettres dans un mot, il ne "voit" pas les caractères individuels — il voit des tokens. Le mot "strawberry" pourrait être tokenisé en ["straw", "berry"] ou même ["str", "aw", "berry"]. Le modèle n'a pas accès direct aux caractères 'r' et doit inférer leur nombre à partir de sa connaissance statistique des mots — une tâche où il échoue souvent.

1.2.2 Les Embeddings : Transformer les Symboles en Vecteurs de Sens

Une fois le texte tokenisé, chaque identifiant numérique doit être converti en une représentation que le réseau de neurones peut manipuler. Cette représentation prend la forme d'un **embedding** : un vecteur dense de nombres réels, typiquement de dimension 768 à 12,288 selon la taille du modèle.