# *Large Language Model*

Cours ≪ Recherche d'Information et Graphe de Données ≫

Nicolas Delestre

INSA
ROUEN NORMANDIE

## Aides de mes amis imaginaires

- Ce cours a été rédigé en LaTeX sous Emacs avec le mode Copilot
- J'ai très souvent échangé avec ChatGPT (version 5.2) et Gemini (version 3) pour présenter les concepts de ce cours plus clairement

INSA ROUEN NORMANDIE

# Plan


1. Introduction à l'IA Générative
2. Architecture des LLM
3. Utilisation des LLM
   - En local
   - Depuis un serveur
4. Chatbot
5. Conclusion


INSA ROUEN NORMANDIE

# Définition et Concepts Clés

## Qu'est-ce que l'IA Générative ?

L'IA générative est une IA capable de produire du contenu (texte, image, son, code) à partir de données d'entraînement

## Exemples d'applications

- Chatbots et assistants virtuels
- Génération automatique de code
- Création de contenu multimédia

INSA ROUEN NORMANDIE

# Historique et Évolution

## Des premiers modèles à l'essor des LLM

- **Chaînes de Markov** (années 1950) : modélisation de la probabilité d'apparition d'un mot en fonction du mot précédent
- **Réseaux de Neurones Récurrents (RNN)** (années 1990) : introduction de la mémoire courte, permettant de traiter des séquences de texte
- **Long Short-Term Memory (LSTM)** (1997) : amélioration des RNN avec des portes de mémoire pour capturer des dépendances à long terme
- **Transformers** (2017) : introduction du mécanisme d'attention permettant de traiter des phrases entières en parallèle
- **GPT-3, GPT-4** (2020-2023) : exploitation des *Transformers* à grande échelle pour une génération de texte fluide et contextuelle

# Tokenisation

## Texte → Tokens

- Le texte est transformé en une séquence de tokens (mots ou sous-mots)
- Utilisation de techniques comme BPE (Byte Pair Encoding) pour découper les mots en unités plus petites
- Permet au modèle de comprendre des mots inconnus en analysant leurs sous-unités

## Exemple de tokenisation

- Mot original : `"transformers"`
- Tokenisation BPE : `"trans"`, `"##form"`, `"##ers"`
- Réduction du vocabulaire de GPT-3 à environ 50 000 tokens

INSA ROUEN NORMANDIE

# Décodeur auto regressif 1 / 4

## Principe

- Décodeur : génère du texte à partir d'une séquence d'entrée
- Auto-régressif : prédit chaque token en fonction des tokens précédents

## Couches d'attention multi-têtes masqué

- Reprend les couches d'attention multi-têtes vues dans le cours sur les embeddings de mots dynamiques (BERT), mais additionne un masque $M$ de façon à ne prendre en compte que les tokens précédents

$$Attention(Q, K, V) = softmax(\frac{K^T Q}{\sqrt{d_k}} + M)V$$

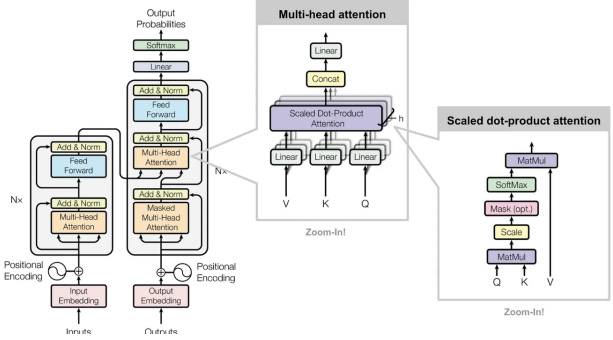
avec :

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

# Décodeur auto regressif 2 / 4

## Dernière couches

- Puis utilisation d'une combinaison linéaire (MLP à une seule couche avec *id* comme fonction d'activation)
- Enfin utilisation softmax pour obtenir une distribution de probabilité sur le prochain token



# Décodeur auto regressif 3 / 4

## En résumé

Liste de mots

↓

Liste de tokens

↓

Embeddings + position

↓

Couches d'attention multi-têtes masqués

↓

Combinaison linéaire

↓

Softmax

↓

Distribution de probabilité sur le prochain token

# Décodeur auto regressif 4 / 4

## Quelques modèles

| Modèle | Fenêtre de contexte | Têtes d'attention / couche | Nb de couches |
|---|---|---|---|
| GPT-2 (Small) | 1 024 | 12 | 12 |
| GPT-3 (175B) | 2 048 | 96 | 96 |
| LLaMA 2 (7B) | 4 096 | 32 | 32 |
| LLaMA 2 (70B) | 4 096 | 64 | 80 |
| Mistral 7B | 8 192 | 32 | 32 |
| Mixtral 8x7B | 32 768 | 32 | 32 |
| Claude 2 | 100 000 | $\sim$ n.d. | $\sim$ n.d. |

- Aujourd'hui les LLM les plus puissants acceptent plus d'un million de tokens en entrée
- Les entreprises ne communiquent plus les détails techniques de leurs modèles fermés

INSA ROUEN NORMANDIE

# Lois d'échelle : pourquoi « plus gros » marche souvent

## Observation empirique

Sur beaucoup de tâches, quand on augmente :

- le nombre de paramètres,
- la taille des données d'entraînement,
- la quantité de calcul,

alors la **loss** diminue de façon régulière

## Intuition

- Plus de paramètres → plus de capacité de représentation
- Plus de données → meilleure couverture linguistique / factuelle
- Plus de calcul → optimisation plus efficace

# Taille, données, compute : le triangle de l'entraînement

## Compromis classique

À un budget donné, on doit équilibrer :

- **Paramètres** (taille du modèle)
- **Données** (nombre de tokens)
- **Calcul** (temps GPU / TPU)

## Conséquences concrètes

- Un modèle **trop petit** sous-apprend même avec beaucoup de données
- Un modèle **trop gros** sur-apprend si on manque de données
- Le coût d'entraînement et le coût d'inférence explosent si on met à l'échellle sans stratégie

# Coûts : entraînement vs inférence

## Entraînement

- Très coûteux (beaucoup de GPU, sur plusieurs semaines)
- Rarement fait par les utilisateurs finaux
- Investissement initial important

## Inférence

- Coût **par requête** (latence, VRAM, énergie)
- Dépend fortement de :
  - la longueur du contexte (#tokens en entrée),
  - la longueur générée (#tokens en sortie),
  - la taille du modèle

## Objectif industriel

Réduire le coût d'inférence : **quantification**, distillation (petit modèle qui imite un grand modèle), **MoE**

# Quantification

## Principe

- Technique qui permet de réduire la taille d'un modèle en inférence
- Remplace les poids des réseaux de neurones (float en 16 ou 32 bits) par des entiers (int8, int4, etc.)

## Avantage

- Permet de faire tourner des modèles plus grands sur du matériel plus modeste

## Inconvénients

- Peut entraîner des pertes de précision, surtout pour les modèles plus petits
- Nécessite des bibliothèques spécialisées (comme `bitsandbytes`)

# *Mixture of Experts*

## Principe

- Un modèle MoE contient plusieurs « experts » (sous-réseaux).
- À chaque token, un **routeur** sélectionne seulement quelques experts à activer

## Avantages

- **Capacité** élevée (beaucoup de paramètres au total)
- **Calcul** modéré à l'inférence (on n'active pas tout)
- Bon compromis qualité / coût

## Inconvénients

- Complexité d'entraînement et de déploiement
- Équilibrage des experts (éviter que certains soient sur-utilisés)

# Pré-entraînement et Fine-Tuning 1 / 3

## Phases d'apprentissage

L'entraînement des LLM repose sur plusieurs étapes :

- **le pré-entraînement**, qui permet au modèle d'apprendre les structures du langage à partir d'un immense corpus de textes
- **les fine-tuning**, qui ajustent le modèle

# Pré-entraînement et Fine-Tuning 2 / 3

## Pré-entraînement

- Apprentissage non supervisé sur de vastes ensembles de données textuelles (livres, articles, pages web)
- Objectif : prédire le prochain token dans une phrase
- Modèles entraînés sur plusieurs téraoctets de données

## Exemple de pré-entraînement

- GPT-3 a été entraîné sur un corpus estimé à plus de **500 milliards de tokens**
- Coût énergétique : plusieurs milliers de GPU pendant plusieurs semaines

INSA ROUEN NORMANDIE

# Pré-entraînement et Fine-Tuning 3 / 3

## Les étapes de Fine-Tuning

1. Fine-tuning supervisé :
   - Objectif : transformer les LLM en chatbot
   - Méthode : présentés des exemples de conversation
2. Fine-tuning par les préférences :
   - Objectif : améliorer la qualité des réponses et vérifié qu'elles sont légales et pas dangereuses
   - Méthode : des humains choisissent la meilleur réponse par mi celles proposées
3. Fine-tuning par le raisonnement
   - Objectif : améliorer la capacité du LLM à effectuer des tâches de raisonnement
   - Méthode : sur des problèmes vérifiables (comme les mathématiques ou le code informatique) on applique des méthodes d'apprentissage par renforcement (automatique)

Les 4 étapes pour entrainer les LLM (ScienceEtonnante)
https://www.youtube.com/watch?v=YcIbZGTRMjI

# Hugging Face

- Hugging Face est une plateforme (historiquement française) qui propose une large collection de modèles (LLM, Embeddings, etc.), ainsi que des outils pour les utiliser facilement
- Il permet de télécharger des modèles open source et de les utiliser localement ou via une API

`https://huggingface.co/`

## Contenu d'un modèle de LLM téléchargé (`~/.cache/huggingface`)

- Un fichier de configuration (`config.json`)
- Les poids des réseaux de neurones (fichiers en `.bin` ou `.safetensors`)
- Le tokenizer et son vocabulaire (`tokenizer.json`, `tokenizer.model`, `vocab.json`, etc.)
- Éventuellement des fichiers de quantification

# Utiliser un LLM en local avec Hugging Face

## Composants nécessaires

- `transformers`
- `torch`
- (optionnel) `accelerate`
- (optionnel) `bitsandbytes` pour la quantification

## Commande d'installation

```
pip install transformers torch accelerate
pip install bitsandbytes # optionnel
```

INSA ROUEN NORMANDIE

# Chargement d'un LLM

## API principale

- **Tokenizer** : `AutoTokenizer`
- **Modèle auto-régressif (GPT-like)** : `AutoModelForCausalLM`
- **Modèle encodeur seul (BERT-like)** : `AutoModel`
- Modèle Seq2Seq (T5, BART) : `AutoModelForSeq2SeqLM`
- Classification de texte : `AutoModelForSequenceClassification`
- Token classification (NER) : `AutoModelForTokenClassification`
- Question Answering extractif : `AutoModelForQuestionAnswering`

## Chargement du tokenizer et du modèle

```
from transformers import AutoTokenizer
from transformers import AutoModelForCausalLM
model_name = "mistralai/Mistral-7B-v0.1"
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

# Génération de texte

## Exemple de génération

```
prompt = "Les modèles de langage sont"

inputs = tokenizer(prompt, return_tensors="pt")

outputs = model.generate(
    **inputs,
    max_new_tokens=50
)

print(tokenizer.decode(outputs[0],
      skip_special_tokens=True))
```

# Utilisation du GPU

## Accélération matérielle

Le paramètre `device_map="auto"` permet de répartir automatiquement le modèle sur le GPU

## Chargement sur GPU

```
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype="auto",
    device_map="auto"
)
```

# Quantification 4-bit

## Chargement en 4-bit

```
from transformers import BitsAndBytesConfig

quant_config = BitsAndBytesConfig(
    load_in_4bit=True
)

model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=quant_config,
    device_map="auto"
)
```

# Serveur LLM compatible OpenAI

## Idée générale

- Centraliser les modèles et les ressources
- Accéder aux modèles à l'aide du protocole HTTP et du JSON pour représenter les requêtes et les réponses
- Autoriser ou pas l'accès à ces services (Bearer)
- Exemple de serveur compatible OpenAI : vLLM (Meta)

## *Endpoints* typiquement disponibles

- `GET /v1/models` : obtenir la liste de modèles disponibles
- `POST /v1/completions` : générer une réponse à partir d'un prompt simple
- `POST /v1/chat/completions` : générer une réponse à partir d'un contexte de conversation

# Démarrer un serveur vLLM (mode OpenAI)

## Principe

On lance un serveur HTTP local qui sert un modèle Hugging Face et expose une API `/v1/*`

## Lancement du serveur

```
python -m vllm.entrypoints.openai.api_server \
  --model mistralai/Mistral-7B-v0.1 \
  --host 0.0.0.0 \
  --port 8000
```

## Accès

- Base URL : `http://localhost:8000/v1`
- *Endpoint* chat : `http://localhost:8000/v1/chat/completions`

# Appel HTTP : /v1/chat/completions

## Requête HTTP

- Méthode : POST
- URL : /v1/chat/completions
- Corps : JSON avec model, messages, paramètres de génération

## Exemple avec curl

```
curl http://.../v1/chat/completions \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $CLE_API" \
  -d '{
    "model": "mistralai/Mistral-7B-v0.1",
    "messages": [
      {"role": "user", "content": "Explique l attention causale."}
    ],
    "temperature": 0.7,
    "max_tokens": 128
```

# Client Python bas niveau : `requests`

## Avantages

- Contrôle explicite de l'URL, headers, timeout, parsing JSON
- Utile pour déboguer et comprendre le protocole

## Exemple `requests`

```python
import requests
cle_api = "..."
url = "http://.../v1/chat/completions"
headers = {
  "Content-Type": "application/json",
  "Authorization": f"Bearer {cle_api}"
}
payload = {
  "model": "mistralai/Mistral-7B-v0.1",
  "messages": [{"role": "user", "content": "Donne 3 usages de RoPE."}],
  "temperature": 0.2,
  "max_tokens": 128
}
r = requests.post(url, json=payload, headers=headers, timeout=60)
r.raise_for_status()
data = r.json()
print(data["choices"][0]["message"]["content"])
```

# Client Python haut niveau : SDK OpenAI (base_url)

## Principe

On peut réutiliser le SDK OpenAI en changeant uniquement :

- `base_url` → serveur local (`http://localhost:8000/v1`)
- `api_key` → valeur factice (ex. `EMPTY`)

## Exemple avec `openai` (chat completions)

```python
from openai import OpenAI

cle_api = "..."
client = OpenAI(
  base_url="http://.../v1",
  api_key="EMPTY"
)

resp = client.chat.completions.create(
  model="mistralai/Mistral-7B-v0.1",
  messages=[{"role": "user", "content": "Resume l auto-regression."}],
  temperature=0.3,
  max_tokens=120
)

print(resp.choices[0].message.content)
```

# Paramètres de génération courants

## Contrôle de la génération

- `temperature` : distribution de probabilité (paramètre de softmax $\frac{e^{z_i/T}}{\sum_j e^{z_j/T}}$)
- `top_p` : le nombre de tokens à considérer (somme des probabilités les plus probables) pour la génération
- `top_k` : garde les k tokens les plus probables
- `max_tokens` : longueur maximale générée
- `stop` : séquences d'arrêt

## Bonnes pratiques

- Fixer un `timeout` côté client
- Journaliser les requêtes/réponses (debug)
- Adapter `max_tokens` pour éviter les réponses trop longues

# API OpenAI (2025) : points clés côté HTTP (évolution vers Responses)

- **Nouveau endpoint unifié** : `POST /v1/responses` (en remplacement progressif de `/v1/chat/completions`)
- **Changement de payload** : `messages` → `input` (format plus générique, prêt pour texte + outils + multimodal)
- **Réponse HTTP structurée et typée** : `choices[]` → `output[]` avec des blocs de contenu (ex. `output_text`) au lieu d'un seul champ texte
- **Outils (function calling) généralisés** : `functions` → `tools` (mécanisme plus extensible, multi-outils natif)
- **Gestion d'état optionnelle côté API** : possibilité de chaîner via un identifiant (ex. `previous_response_id`) au lieu de renvoyer tout l'historique
- **Headers globalement similaires** : `Authorization: Bearer ...`, `Content-Type: application/json`

INSA ROUEN NORMANDIE

# Chatbot

## Caractéristiques

- Interface conversationnelle qui utilise un LLM
- Comportement configuré à l'aide :
  - d'un prompt système
  - de paramètres internes, par exemple température ou top_p
  - d'un historique de la conversation (contexte), contenu dans le champ `messages` de la requête, avec les rôles :
    - `system` : prompt système
    - `user` : messages de l'utilisateur
    - `assistant` : messages générés par le chatbot

- Par défaut ne cherche pas d'information, ne fait pas de calcul, ne peut pas faire d'action

# Hallucinations : génération $\neq$ vérification

## Définition

- Une hallucination est une information **plausible mais fausse** générée par le modèle.
- Rappel : le LLM optimise la **probabilité du prochain token**, pas la vérité

## Quand cela arrive plus souvent ?

- Domaine très spécialisé ou hors distribution
- Manque de contexte / consignes ambiguës
- Paramètres de génération plus « créatifs » (température élevée)

## Bonnes pratiques

- Demander des sources, des hypothèses, des étapes
- Valider par une référence externe (moteur de recherche, documentation, RAG)

# Biais et sécurité : enjeux en pratique

## Biais

- Les données d'entraînement contiennent des biais (sociaux, culturels, linguistiques)
- Filtrer et aligner aide, mais ne garantit pas l'absence de biais
- Importance de l'évaluation (tests, red teaming, métriques)

## Sécurité (exemples)

- *Prompt injection* : l'utilisateur tente de contourner les consignes
- Fuite d'informations : données sensibles dans le contexte / logs
- Usages non souhaités : génération de contenu illégal ou dangereux

## Implication

- Un LLM en production nécessite : filtrage, monitoring, garde-fous, politiques de données

# Conclusion

## Nous avons vu dans ce cours

- Comment fonctionnent les LLM (tokenisation, attention, auto-régression)
- Comment les entraîner (pré-entraînement, fine-tuning, RLHF)
- Comment les utiliser en local (Hugging Face) ou via un serveur compatible OpenAI
- Ce qu'est un chatbot et comment il utilise un LLM

## Important

Les LLM ne font pas de recherche d'information, ils produisent une succession de tokens qui sont statiquement probables

INSA ROUEN NORMANDIE