# Guide Technique Détaillé : Créer une IA Locale de A à Z

## Introduction à l’IA Locale

Dans un monde où l’intelligence artificielle prend une place prépondérante, la capacité à maîtriser et à déployer des solutions d’[IA](#ia-locale) en local devient un avantage stratégique majeur. Ce guide technique est conçu pour vous accompagner pas à pas dans la création de votre propre système d’[IA](#ia-locale) fonctionnant entièrement sur votre infrastructure, sans dépendance aux services cloud externes. Que ce soit pour des raisons de confidentialité des données, de maîtrise des coûts, de souveraineté technologique ou de personnalisation avancée, l’[IA locale](#ia-locale) offre une flexibilité et un contrôle inégalés.

Nous explorerons ensemble les concepts fondamentaux, les pré-requis matériels et logiciels, les algorithmes clés (comme le [RAG](#rag-retrieval-augmented-generation) et le [Fine-tuning](#fine-tuning)), les outils essentiels et les meilleures pratiques pour mettre en œuvre une IA performante et sécurisée. Ce document s’adresse aux développeurs, ingénieurs et techniciens souhaitant approfondir leurs connaissances et concrétiser des projets d’IA en environnement local.

## Glossaire

* **Accelerate**: Bibliothèque de Hugging Face qui simplifie l’entraînement de modèles sur des configurations distribuées (multi-GPU, TPU).
* **API (Application Programming Interface)**: Interface permettant à différents logiciels de communiquer entre eux.
* **AWQ (Activation-aware Weight Quantization)**: Méthode de quantification des modèles pour réduire leur taille et accélérer l’inférence.
* **bitsandbytes**: Bibliothèque permettant la quantification des modèles (ex: en 8-bit ou 4-bit) pour réduire leur empreinte mémoire.
* **CatBoost**: Algorithme de boosting d’arbres de décision, performant sur les données tabulaires.
* **Chunking**: Processus de segmentation de documents textuels en morceaux plus petits (chunks) pour l’indexation et le traitement par des modèles de langage.
* **Chroma**: Base de données vectorielle locale, simple d’utilisation.
* **CNN (Convolutional Neural Network)**: Type de réseau de neurones particulièrement efficace pour le traitement d’images.
* **Cross-Encoder**: Type de modèle qui évalue la pertinence d’une paire de textes (ex: question et document) en les traitant simultanément. Plus lent mais plus précis que les bi-encodeurs, il est souvent utilisé pour le re-ranking.
* **CUDA**: Architecture de calcul parallèle et plateforme de programmation développée par NVIDIA pour ses GPU. Essentielle pour l’accélération matérielle du deep learning.
* **cuDNN**: Bibliothèque de primitives accélérées par GPU pour les réseaux de neurones profonds, complémentaire à CUDA.
* **Docker**: Plateforme permettant d’encapsuler des applications et leurs dépendances dans des conteneurs isolés pour une meilleure reproductibilité.
* **DVC (Data Version Control)**: Système de gestion de versions pour les données et les modèles de Machine Learning.
* **Embeddings**: Représentations vectorielles (listes de nombres) de données (mots, phrases, documents) dans un espace de grande dimension. La proximité entre vecteurs indique une similarité sémantique.
* **FAISS (Facebook AI Similarity Search)**: Bibliothèque développée par Facebook AI pour la recherche efficace de similarité parmi des milliards de vecteurs.
* **Fine-tuning**: Processus de spécialisation d’un modèle de langage pré-entraîné sur un jeu de données spécifique à une tâche ou un domaine, en ajustant ses poids.
* **GGUF (GPT-Generated Unified Format)**: Format de fichier conçu pour exécuter efficacement les modèles de langage sur CPU et GPU, popularisé par des outils comme llama.cpp.
* **GPTQ**: Méthode de quantification des modèles (souvent en 4 bits) pour réduire leur taille et accélérer l’inférence.
* **HNSW (Hierarchical Navigable Small World)**: Algorithme de recherche de plus proches voisins approximatif, performant et équilibré, utilisé dans des bases de données vectorielles comme FAISS.
* **Inference**: Phase d’utilisation d’un modèle entraîné pour faire des prédictions sur de nouvelles données.
* **InfoNCE (Info Noise-Contrastive Estimation)**: Fonction de perte utilisée pour entraîner des modèles d’embeddings, notamment dans les approches contrastives.
* **IA Locale**: Intelligence artificielle qui fonctionne entièrement sur l’ordinateur de l’utilisateur, sans nécessiter de connexion Internet ou de services cloud.
* **IVF (Inverted File Index)**: Méthode d’indexation pour la recherche de vecteurs qui partitionne l’espace vectoriel en cellules pour accélérer la recherche.
* **kNN (k-Nearest Neighbors)**: Algorithme de recherche des k éléments les plus proches d’un point donné dans un espace de données.
* **KV-caching**: Technique d’optimisation pour les LLM qui stocke les clés et valeurs (Key-Value) des couches d’attention précédentes pour accélérer la génération de texte.
* **LangChain**: Framework d’orchestration pour construire des applications basées sur les LLM, notamment des systèmes RAG.
* **LLM (Large Language Model)**: Modèle de langage de grande échelle, comme GPT-3 ou Llama, entraîné sur de vastes corpus de texte.
* **LM Studio**: Interface utilisateur graphique conviviale pour télécharger, exécuter et interagir avec des LLM locaux.
* **LoRA (Low-Rank Adaptation)**: Technique de fine-tuning efficace qui consiste à n’entraîner qu’un petit nombre de paramètres (adaptateurs) ajoutés au modèle, réduisant ainsi considérablement les besoins en calcul et en mémoire.
* **LlamaIndex**: Alternative à LangChain, également axée sur la construction d’applications LLM et de pipelines RAG.
* **LSTM (Long Short-Term Memory)**: Type de réseau de neurones récurrents, efficace pour les séquences de données comme les séries temporelles.
* **MMR (Maximal Marginal Relevance)**: Stratégie de diversification utilisée lors de la recherche de documents pour éviter la redondance et augmenter la couverture de l’information.
* **MLOps**: Ensemble de pratiques pour déployer et maintenir des modèles de Machine Learning en production.
* **MVP (Minimum Viable Product)**: Version d’un produit avec juste assez de fonctionnalités pour être utilisable par les premiers clients.
* **MTEB (Massive Text Embedding Benchmark)**: Benchmark complet pour évaluer la qualité des modèles d’embeddings sur un large éventail de tâches.
* **NF4 (NormalFloat 4-bit)**: Format de quantification en 4 bits utilisé dans QLoRA pour les poids des modèles.
* **OCR (Optical Character Recognition)**: Technique permettant de convertir des images de texte en texte éditable.
* **Ollama**: Outil convivial qui simplifie le téléchargement, l’exécution et la gestion de modèles de langage locaux via une interface en ligne de commande ou une API.
* **PEFT (Parameter-Efficient Fine-Tuning)**: Ensemble de techniques (dont LoRA) visant à adapter les grands modèles pré-entraînés à des tâches en aval de manière efficace, en ne modifiant qu’une petite fraction de leurs paramètres.
* **PII (Personally Identifiable Information)**: Informations personnelles identifiables (nom, adresse, etc.) qu’il est crucial de protéger.
* **Playbook**: Guide ou ensemble d’instructions détaillées pour réaliser une tâche ou un processus.
* **Prompt Engineering**: Art de concevoir des instructions (prompts) claires et efficaces pour guider un modèle de langage vers la sortie souhaitée.
* **PyTorch**: Framework de deep learning open-source populaire, connu pour sa flexibilité et son écosystème riche.
* **Qdrant**: Base de données vectorielle open-source, optimisée pour la recherche de similarité.
* **QLoRA (Quantized Low-Rank Adaptation)**: Variante de LoRA qui combine le fine-tuning par adaptateurs avec la quantification du modèle de base (généralement en 4 bits), permettant d’entraîner des modèles très volumineux sur des GPU avec une VRAM limitée.
* **Quantization**: Processus de réduction de la précision numérique des poids d’un modèle (ex: de 32-bit à 8-bit ou 4-bit) pour diminuer sa taille, son empreinte mémoire et accélérer l’inférence, souvent avec une perte de qualité minime.
* **RAG (Retrieval-Augmented Generation)**: Architecture où un modèle de langage ne s’appuie pas uniquement sur sa mémoire interne, mais “récupère” (retrieve) des informations pertinentes d’une base de connaissances externe (comme une base de données vectorielle) avant de générer une réponse.
* **RandomForest**: Algorithme d’apprentissage automatique basé sur la construction de multiples arbres de décision.
* **ROCm**: Plateforme de calcul open-source de AMD pour le calcul sur GPU, alternative à CUDA.
* **Telemetry**: Collecte et transmission automatique de données à distance pour surveiller le fonctionnement d’un système.
* **TGI (Text Generation Inference)**: Serveur d’inférence de Hugging Face optimisé pour la génération de texte à haut débit.
* **TPU (Tensor Processing Unit)**: Processeur spécialisé développé par Google pour accélérer les charges de travail de Machine Learning.
* **Transformers**: Bibliothèque de Hugging Face qui donne accès à des milliers de modèles pré-entraînés et à des outils pour les entraîner et les utiliser.
* **Transformer temporel (TFT)**: Modèle de type Transformer adapté aux prévisions de séries temporelles.
* **Vector Database**: Base de données spécialisée dans le stockage et la recherche ultra-rapide de représentations vectorielles (embeddings). Exemples : FAISS, Chroma, Qdrant, Milvus.
* **ViT (Vision Transformer)**: Modèle de type Transformer appliqué aux tâches de vision par ordinateur.
* **vLLM**: Serveur d’inférence pour LLM très performant, optimisé pour un débit élevé.
* **VRAM (Video RAM)**: Mémoire vive dédiée aux cartes graphiques (GPU), essentielle pour les calculs d’IA.
* **Whisper**: Modèle de reconnaissance vocale automatique (ASR) développé par OpenAI.
* **XGBoost**: Algorithme de boosting d’arbres de décision très efficace et populaire.
* **YOLO (You Only Look Once)**: Algorithme de détection d’objets en temps réel.

## Table des Matières

1. [Introduction à l’IA Locale](#introduction-à-lia-locale)
2. [Pré-requis & Environnement](#pré-requis--environnement)
3. [Définir le Problème et la Stratégie](#définir-le-problème-et-la-stratégie)
4. [Préparation des Données](#préparation-des-données)
5. [Les Algorithmes Clés Expliqués](#les-algorithmes-clés-expliqués)
   * [RAG (Retrieval-Augmented Generation)](#rag-retrieval-augmented-generation)
   * [Fine-tuning (LoRA / QLoRA)](#fine-tuning-lora--qlora)
   * [Autres algorithmes (Classification, Vision, etc.)](#Xa9df6617b340172333620dbcd7ee5f259a5e6b2)
6. [Outils et Stacks Locales](#outils-et-stacks-locales)
7. [Mise en Pratique : Pas-à-Pas](#mise-en-pratique--pas-à-pas)
   * [Étape A : Préparer l’environnement](#étape-a--préparer-lenvironnement)
   * [Étape B : Construire la base de connaissances (RAG)](#Xf5e275a4923c6a79aab0180f9e2963be34e62d8)
   * [Étape C : Inférence locale avec un LLM](#étape-c--inférence-locale-avec-un-llm)
   * [Étape D : Fine-tuning léger (QLoRA)](#étape-d--fine-tuning-léger-qlora)
   * [Étape E : Évaluation](#étape-e--évaluation)
   * [Étape F : Déploiement local via une API](#étape-f--déploiement-local-via-une-api)
8. [Cas d’Usage : Exploiter son Archive YouTube](#Xbb0af6689c4975937a41bfeb71ddaf939031659)
9. [Sécurité, Confidentialité & Licences](#sécurité-confidentialité--licences)
10. [Annexes : Exemples de Code](#annexes--exemples-de-code)
11. [Modèles & tailles conseillés (local)](#modèles--tailles-conseillés-local)
12. [Bonnes pratiques de projet](#bonnes-pratiques-de-projet)
13. [Checklist finale](#checklist-finale)

## 0) Pré-requis & environnement

* **OS** : Windows, macOS, Linux (Ubuntu recommandé pour le GPU).
* **Matériel** :
  + CPU moderne ; 16–32 Go RAM conseillés.
  + [**GPU**](#gpu) recommandé pour l’entraînement / l’inférence rapide :
    - NVIDIA ([CUDA](#cuda), >= 8–12 Go [VRAM](#vram) pour petits modèles ; 24–48 Go+ pour fine-tuning LLM plus grands)
    - AMD ([ROCm](#rocm)) ou Apple Silicon (Metal)
* **Gestion d’environnements** : conda ou pyenv + venv.
* **Pilotes & Toolkits** :
  + NVIDIA : pilotes + [**CUDA**](#cuda) + [**cuDNN**](#cudnn)
  + AMD : **ROCm** (Linux)
  + Apple : **Xcode Command Line Tools** (Metal)
* **Paquets clés** : Python 3.10/3.11, [pytorch](#pytorch), [transformers](#transformers), [accelerate](#accelerate), [bitsandbytes](#bitsandbytes), datasets, scikit-learn, sentence-transformers, [faiss](#faiss)/[qdrant-client](#qdrant), [langchain](#langchain)/[llama-index](#llamaindex) (au choix).

Astuce : si tu veux un démarrage ultra-simple sans config complexe, commence avec [**Ollama**](#ollama) ou [**LM Studio**](#lm-studio) pour l’inférence locale, puis ajoute l’entraînement léger ([LoRA](#lora)/[QLoRA](#qlora)) plus tard.

## 1) Définir le **problème** (cartographie vers les algorithmes)

1. **Type de tâche** :
   * **Texte → Texte** (assistant, résumé, génération) ⇒ [**LLM**](#llm-large-language-model) (GPT-like) + éventuellement [**RAG**](#rag-retrieval-augmented-generation).
   * **Texte → Classe** (étiquette) ⇒ **Classif.** (LLM + classifier ou scikit-learn/transformers).
   * **Recherche sémantique / FAQ** ⇒ [**Embeddings**](#embeddings) + **base vectorielle** + (option) **reranking**.
   * **Image** (classification, détection, OCR) ⇒ [**CNN**](#cnn-convolutional-neural-network)/[**ViT**](#vit-vision-transformer), timm, ultralytics ([YOLO](#yolo-you-only-look-once)), mmocr.
   * **Audio** (transcription) ⇒ [**Whisper**](#whisper).
   * **Tabulaires** (prédiction) ⇒ [**XGBoost**](#xgboost), [RandomForest](#randomforest), [CatBoost](#catboost).
   * **Séries temporelles** ⇒ **Prophet**, [LSTM](#lstm-long-short-term-memory)/[Transformer temporel (TFT)](#transformer-temporel-tft).
2. **Contraintes** : latence, mémoire, confidentialité, interprétabilité, données disponibles.
3. **Stratégie** :
   * Peu de données privées ? ⇒ [**RAG**](#rag-retrieval-augmented-generation) + modèle pré-entraîné **quantifié** ([GGUF](#gguf-gpt-generated-unified-format), [AWQ](#awq-activation-aware-weight-quantization), [GPTQ](#gptq)).
   * Beaucoup de données spécifiques ? ⇒ [**Fine-tuning**](#fine-tuning) partiel ([**LoRA**](#lora-low-rank-adaptation)/[**QLoRA**](#qlora-quantized-low-rank-adaptation)) ou complet (coûteux).

## 2) Préparer les **données** (générales & personnelles)

### 2.1 Données générales

* Open datasets (Hugging Face Hub, Kaggle), documents internes (PDF, pages web), logs, tickets.
* Nettoyage : encodage UTF-8, déduplication, normalisation (casse, espaces), suppression [PII](#pii-personally-identifiable-information) si besoin.

### 2.2 Données personnelles depuis **Google Takeout** (YouTube)

* Fichiers utiles : watch-history.json, playlists.json, comments.csv, subscriptions.csv, dossier captions/ (sous-titres), etc.
* Utilisation :
  + Construire une **bibliographie** de contenus (titres, URLs) sur l’IA.
  + Extraire les **thèmes** (NLP, [RAG](#rag-retrieval-augmented-generation), [fine-tuning](#fine-tuning), vector DB, etc.).
  + Générer un **corpus** pour [RAG](#rag-retrieval-augmented-generation) (notes, transcriptions associées, résumés).

### 2.3 Pipeline de préparation (texte)

* **Segmentation/**[**Chunking**](#chunking) :
  + *Heuristique* (par paragraphes, titres) ou **RecursiveCharacterTextSplitter** (ex. [LangChain](#langchain))
  + Par **tokens** (compteur tokenizer) pour respecter les fenêtres contextuelles.
* **Nettoyage** : supprimer HTML, normaliser espaces, corriger [OCR](#ocr-optical-character-recognition).
* **Enrichissement** : métadonnées (source, date, auteur, URL), tags, langue.

## 3) Choisir et **expliquer** les algorithmes clés

### 3.1 [RAG](#rag-retrieval-augmented-generation) (Retrieval-Augmented Generation)

* **Idée** : au lieu d’entraîner le modèle sur tout, on **indexe** nos documents. À l’inférence, on *retrouve* ([kNN](#knn-k-nearest-neighbors)) les passages pertinents (via [**embeddings**](#embeddings)) puis le [LLM](#llm-large-language-model) **génère** une réponse en s’appuyant dessus.
* **Composants** :
  1. [Embeddings](#embeddings) (encodeurs bi-encodeurs) → vecteurs :
     + Populaires : bge-small/bge-base, e5-base, all-MiniLM-L6-v2, nomic-embed, gte-small.
  2. Index vectoriel : [FAISS](#faiss-facebook-ai-similarity-search) (local, rapide), [Qdrant](#qdrant)/Milvus (serveur), [Chroma](#chroma) (simple, local).
  3. [Reranking](#reranking) (option) : [Cross-Encoder](#cross-encoder) (ex. cross-encoder/ms-marco-MiniLM-L-6-v2) pour reclasser les passages.
  4. [LLM](#llm-large-language-model) local : via llama.cpp / [Ollama](#ollama) / [vLLM](#vllm) / [LM Studio](#lm-studio) (quantifié en [GGUF](#gguf-gpt-generated-unified-format) pour CPU/[GPU](#gpu) modestes).
* Quand choisir [RAG](#rag-retrieval-augmented-generation) ?
  + Quand les données évoluent souvent, que la confidentialité compte, et que tu veux éviter le coût du [fine-tuning](#fine-tuning).

### 3.2 [Fine-tuning](#fine-tuning) [**LoRA**](#lora-low-rank-adaptation) / [**QLoRA**](#qlora-quantized-low-rank-adaptation) ([LLM](#llm-large-language-model))

* **Idée** : geler la plupart des poids et n’entraîner que des **adaptateurs bas-rang** ([LoRA](#lora-low-rank-adaptation)). [**QLoRA**](#qlora-quantized-low-rank-adaptation) quantifie les poids en 4 bits (sauvegardant la [VRAM](#vram)) tout en apprenant en 16 bits effectifs sur les adaptateurs.
* **Avantages** : rapide, peu coûteux, personnalisable, réversible (on applique les deltas).
* **Quand ?**
  + Quand la **tonalité** ou le **raisonnement** doivent refléter fortement ton domaine et que tu as des exemples d’instructions/réponses.

### 3.3 Classifieurs & régressions (tabulaires)

* [**XGBoost**](#xgboost) / LightGBM / [CatBoost](#catboost) : excellents sur tabulaires, rapides, interprétables via SHAP.
* **Quand ?**
  + Prédictions structurées (score, churn, fraude), faible besoin de texte long.

### 3.4 Vision & Audio

* **Vision** : timm ([ViT](#vit-vision-transformer), ConvNeXt), [**YOLOv8**](#yolo-you-only-look-once) (détection), Segment Anything (segmentation).
* **Audio** : [**Whisper**](#whisper) (ASR), modèles TTS locaux (ex. Piper, VITS) si besoin.

## 4) **Outils** et stacks locales

* **Gestion de modèles** :
  + [**Ollama**](#ollama) (macOS/Windows/Linux) : ollama pull llama3:8b puis ollama run ...
  + [**LM Studio**](#lm-studio) : UI conviviale pour charger des [GGUF](#gguf-gpt-generated-unified-format) et chatter en local.
  + llama.cpp : binaire C++ performant CPU/[GPU](#gpu) (fichiers [**GGUF**](#gguf-gpt-generated-unified-format)).
  + [**vLLM**](#vllm) : serveur haut débit pour cartes [GPU](#gpu) plus puissantes).
* **Entraînement** : [PyTorch](#pytorch) + [**Hugging Face Transformers**](#transformers), [**PEFT**](#peft-parameter-efficient-fine-tuning) ([LoRA](#lora-low-rank-adaptation)), [Accelerate](#accelerate), [bitsandbytes](#bitsandbytes) (4/8 bits).
* [**RAG**](#rag-retrieval-augmented-generation) : [FAISS](#faiss-facebook-ai-similarity-search)/[Qdrant](#qdrant)/[Chroma](#chroma), [LangChain](#langchain) ou [LlamaIndex](#llamaindex) (chaînes, retrievers, splitters).
* **Évaluation** : lm-eval-harness ([LLM](#llm-large-language-model)), [**MTEB**](#mteb-massive-text-embedding-benchmark) ([embeddings](#embeddings)), scikit-learn (F1, ROC-AUC), BLEU/ROUGE.
* [**MLOps**](#mlops) local : [DVC](#dvc-data-version-control) (données), Weights & Biases/MLflow (suivi), [Docker](#docker) (reproductibilité).

## 5) Parcours **pas-à-pas** (scénario NLP local)

### Étape A — Préparer l’environnement

# Exemple (Linux, NVIDIA)  
conda create -n ia-local python=3.11 -y  
conda activate ia-local  
pip install torch --index-url https://download.pytorch.org/whl/cu124  
pip install transformers accelerate bitsandbytes datasets sentence-transformers faiss-cpu langchain chromadb peft

### Étape B — Construire la base de connaissances ([RAG](#rag-retrieval-augmented-generation))

1. **Collecte** : PDF, notes, articles, **transcriptions YouTube** (depuis Takeout ou [API](#api-application-programming-interface)).
2. [**Chunking**](#chunking) : 500–1 000 tokens, overlap 50–100.
3. [**Embeddings**](#embeddings) : sentence-transformers (ex. bge-small pour CPU rapide).
4. **Index** : [FAISS](#faiss-facebook-ai-similarity-search) ou [Chroma](#chroma) (local).
5. **Chaîne** : Retriever → Prompt → [LLM](#llm-large-language-model) local.

**Exemple minimal (Python)**

from sentence\_transformers import SentenceTransformer  
from chromadb import Client  
from chromadb.config import Settings  
  
texts = ["Qu'est-ce que le RAG ?", "Le RAG combine retrieval et génération."]  
metadatas = [{"source":"note1"},{"source":"note2"}]  
olds = ["1","2"]  
  
emb = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")  
vecs = emb.encode(texts).tolist()  
  
chroma = Client(Settings(anonymized\_telemetry=False,persist\_directory="./chroma"))  
col = chroma.create\_collection(name="docs")  
col.add(embeddings=vecs, documents=texts, metadatas=metadatas, ids=ids)  
  
q = "Décris le RAG"  
qv = emb.encode([q]).tolist()  
res = col.query(query\_embeddings=qv, n\_results=2)  
print(res["documents"])

Ensuite, on injecte ces passages dans le prompt envoyé à un **LLM local** (Ollama/llama.cpp) pour générer la réponse ancrée sur tes données.

### Étape C — Inférence locale ([LLM](#llm-large-language-model))

* [**Ollama**](#ollama)

# Installer Ollama puis :  
ollama pull llama3.1:8b  
ollama run llama3.1:8b

* llama.cpp ([GGUF](#gguf-gpt-generated-unified-format))

# Convertir ou télécharger un modèle GGUF puis :  
./main -m models/llama.gguf -p "Bonjour" -n 256

### Étape D — [Fine-tuning](#fine-tuning) léger ([QLoRA](#qlora-quantized-low-rank-adaptation)) sur tes exemples

from datasets import load\_dataset  
from [transformers](#transformers) import AutoModelForCausalLM, AutoTokenizer, TrainingArguments  
from [peft](#peft-parameter-efficient-fine-tuning) import LoraConfig, get\_peft\_model  
from trl import SFTTrainer  
  
base = "meta-llama/Llama-3.1-8B-Instruct" # exemple public (remplacer si besoin)  
model = AutoModelForCausalLM.from\_pretrained(base, load\_in\_4bit=True, device\_map="auto")  
tok = AutoTokenizer.from\_pretrained(base)  
  
lora = LoraConfig(r=8, lora\_alpha=16, lora\_dropout=0.05, task\_type="CAUSAL\_LM")  
model = get\_peft\_model(model, lora)  
  
# Dataset d'instructions (prompt, response)  
ds = load\_dataset("json", data\_files="instructions.jsonl")  
  
args = TrainingArguments(  
 output\_dir="out",  
 per\_device\_train\_batch\_size=2,  
 gradient\_accumulation\_steps=8,  
 learning\_rate=2e-4,  
 num\_train\_epochs=3,  
 fp16=True,  
 logging\_steps=10,  
 save\_steps=200,  
)  
trainer = SFTTrainer(model=model, tokenizer=tok, train\_dataset=ds["train"], args=args)  
trainer.train()  
model.save\_pretrained("out-lora")

### Étape E — Évaluation rapide

* [**LLM**](#llm-large-language-model) : perplexité (proxy), tests d’instructions (jeu de prompts) ; métriques de factualité (simple : exact match sur Q/A connues).
* [**Embeddings**](#embeddings) : [MTEB](#mteb-massive-text-embedding-benchmark) (ou tests maison : recherche de n voisin le plus proche et précision@k).
* **Classif** : F1, précision, rappel, ROC-AUC.

### Étape F — Déploiement **local** ([API](#api-application-programming-interface))

* [**Ollama**](#ollama) :

ollama serve # lance une API locale (http://localhost:11434)

* [**vLLM**](#vllm) :

python -m vllm.entrypoints.openai.api\_server --model /chemin/vers/ton-modele

* **Intégration** [**RAG**](#rag-retrieval-augmented-generation) : expose un endpoint /ask qui : (1) vectorise la question, (2) retrouve k passages, (3) les met dans le prompt, (4) appelle le [LLM](#llm-large-language-model) local, (5) renvoie la réponse + sources.

## 6) Comment **choisir** l’algorithme ? (arbre de décision simple)

1. Ai-je surtout **des documents** (PDF, notes, transcriptions) ?
   * Oui ⇒ [**RAG**](#rag-retrieval-augmented-generation) + [LLM](#llm-large-language-model) quantifié.
2. Ai-je **beaucoup d’exemples d’instructions** (prompt → réponse) propres à mon domaine ?
   * Oui ⇒ [**QLoRA**](#qlora-quantized-low-rank-adaptation) ([fine-tuning](#fine-tuning) léger) + éventuellement [RAG](#rag-retrieval-augmented-generation).
3. Ai-je des **variables tabulaires** et un **objectif** numérique/catégoriel ?
   * Oui ⇒ [**XGBoost**](#xgboost)/LightGBM.
4. Ai-je besoin de **vision**/**audio** ?
   * Oui ⇒ [ViT](#vit-vision-transformer)/[YOLO](#yolo-you-only-look-once) ; [Whisper](#whisper).

Règle pratique : commence par [**RAG**](#rag-retrieval-augmented-generation) (rapide, robuste), puis ajoute du [fine-tuning](#fine-tuning) seulement si les réponses manquent de style ou de raisonnement spécifique.

## 7) Détails d’algorithmes (en bref)

* [**Embeddings**](#embeddings) **bi-encodeurs** : deux encodeurs identiques qui projettent texte requête et texte document dans le **même espace vectoriel**. Similarité (cos, dot). Entraînés avec pertes **contrastives** (ex. [InfoNCE](#Xb79b30e3e7eefe264686b0e7644e365ef330c3f)) sur paires positives/négatives.
* **Rerankers cross-encodeurs** : concatènent requête+document et score le pair avec un encodeur bi-entrée. Plus lents mais plus précis. Utiles après un premier filtrage [kNN](#knn-k-nearest-neighbors).
* [**LLM**](#llm-large-language-model) (transformer décoder) : auto-régressif, *self-attention* causale. Taille de contexte, quantification, [KV-caching](#kv-caching) influencent la latence.
* [**LoRA**](#lora-low-rank-adaptation) : décompose une matrice W en W + A·B (rang bas), seuls A/B sont appris.
* [**QLoRA**](#qlora-quantized-low-rank-adaptation) : quantifie W en 4 bits ([NF4](#nf4-normalfloat-4-bit)) + adapte via [LoRA](#lora-low-rank-adaptation).
* [**FAISS**](#faiss-facebook-ai-similarity-search) : index [kNN](#knn-k-nearest-neighbors) (Flat, [IVFFlat](#ivf-inverted-file-index), [HNSW](#hnsw-hierarchical-navigable-small-world)) ; compromis précision/latence/mémoire.
* [**XGBoost**](#xgboost) : boosting d’arbres de décision, gère non-linéarités et interactions, régularisation.

## 8) Sécurité, confidentialité & licences

* **Local-first** : garde les documents sensibles hors du cloud.
* **Masquage** [**PII**](#pii-personally-identifiable-information) dans le corpus (regex e-mail, téléphone, IBAN, etc.).
* **Licences** : vérifie la licence des modèles (usage commercial ?), des datasets et des poids.
* **Traçabilité** : journalise sources et versions ([DVC](#dvc-data-version-control)/MLflow).

## 9) Exploiter **ton archive YouTube** pour alimenter le guide & le système

### 9.1 Extraction de ressources depuis Takeout

* Parcours watch-history.json et playlists.json ⇒ filtre par mots-clés (“IA”, “machine learning”, “deep learning”, “RAG”, “NLP”, etc.).
* Établis une **liste de lecture** (titres, URLs, chaînes, date) + **thèmes**.
* Option : mapper vers des **transcriptions** (dossier captions/ si présentes) et créer des **notes résumées** (via [LLM](#llm-large-language-model) local).

**Pseudo-code**

import json, csv, re, pathlib  
from datetime import datetime  
  
KEYWORDS = r'''\b([IA](#ia-locale)|intelligence artificielle|machine learning|deep learning|nlp|[RAG](#rag-retrieval-augmented-generation)|[embedding](#embeddings)|[transformer](#transformers)|[pytorch](#pytorch)|[hugging face](#transformers)|[vector](#vector-database)|[faiss](#faiss-facebook-ai-similarity-search)|[qdrant](#qdrant)|[lora](#lora-low-rank-adaptation)|[quantization](#quantization))\b'''  
  
root = pathlib.Path("/chemin/vers/Takeout/YouTube et YouTube Music/")  
wh = json.loads((root/"history/watch-history.json").read\_text(encoding="utf-8"))  
rows = []  
for item in wh:  
 title = item.get("title", "")  
 url = item.get("titleUrl")  
 ch = item.get("subtitles", [{}])[0].get("name") if item.get("subtitles") else None  
 dt = item.get("time")  
 if re.search(KEYWORDS, (title or "").lower()):  
 rows.append({"title":title, "url":url, "channel":ch, "time":dt})  
  
with open("ressources\_youtube\_ia.csv","w",newline='',encoding="utf-8") as f:  
 w=csv.DictWriter(f,fieldnames=["title","url","channel","time"])  
 w.writeheader(); w.writerows(rows)

### 9.2 Générer un **corpus** [**RAG**](#rag-retrieval-augmented-generation) à partir des sous-titres

* Convertis .vtt/.srt en texte, segmente, ajoute **métadonnées** (vidéo, timecodes, URL).
* Indexe avec [**FAISS**](#faiss-facebook-ai-similarity-search)**/**[**Chroma**](#chroma) pour du QA local sur **tes contenus favoris**.

## 10) Modèles & tailles conseillés (local)

* [**LLM**](#llm-large-language-model) **généraux** : 3–8B paramètres pour CPU/[GPU](#gpu) modestes (quantifiés) ; 13–14B si 24–32 Go [VRAM](#vram) ; 70B+ si multi-[GPU](#gpu).
* [**Embeddings**](#embeddings) : bge-small/gte-small (rapides), e5-base (équilibre), nomic-embed (gros corpus).
* **Reranking** : MiniLM cross-encoder léger.
* **Vision** : [ViT](#vit-vision-transformer)-Base / ConvNeXt-Tiny pour commencer.
* **Audio** : [Whisper](#whisper)-small/base.

## 11) Bonnes pratiques de **projet**

* Commencer par un [**MVP**](#mvp-minimum-viable-product) : [RAG](#rag-retrieval-augmented-generation) + [LLM](#llm-large-language-model) 3–8B.
* Ajouter la **télémetry locale** (latence, tokens, RAM/[VRAM](#vram)) pour dimensionner.
* Itérer : meilleure segmentation, [embeddings](#embeddings) plus solides, ajout d’un reranker, [prompt engineering](#prompt-engineering), [QLoRA](#qlora-quantized-low-rank-adaptation) si besoin.
* **Tests** end-to-end avec un *playbook de prompts* réels.

## 12) Annexes – Exemples de code utiles

### 12.1 [Embeddings](#embeddings) + [FAISS](#faiss-facebook-ai-similarity-search) (pur Python)

import faiss, numpy as np  
from sentence\_transformers import SentenceTransformer  
  
texts = ["doc1 sur le RAG", "doc2 sur LoRA"]  
model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")  
X = model.encode(texts)  
index = faiss.IndexFlatIP(X.shape[1])  
faiss.normalize\_L2(X)  
index.add(X)  
q = model.encode(["explication RAG"])  
faiss.normalize\_L2(q)  
D,I = index.search(q, k=2)  
print(I, D)

### 12.2 Pipeline [QLoRA](#qlora-quantized-low-rank-adaptation) ([Hugging Face Transformers](#transformers), résumé)

# voir Étape D pour une version complète ; ici focus hyperparams  
peft\_config = LoraConfig(r=16, lora\_alpha=32, lora\_dropout=0.05, target\_modules=["q\_proj","v\_proj"])  
# learning\_rate: 2e-4 à 1e-5 ; epochs: 1-3 ; cutoff\_len: 2k-4k tokens

### 12.3 Serveur local [RAG](#rag-retrieval-augmented-generation) (Fast[API](#api-application-programming-interface), schéma minimal)

from fastapi import FastAPI  
from sentence\_transformers import SentenceTransformer  
from chromadb import Client; from chromadb.config import Settings  
import requests  
  
app = FastAPI(); emb = SentenceTransformer("all-MiniLM-L6-v2")  
chroma = Client(Settings(persist\_directory="./chroma"))  
col = chroma.get\_or\_create\_collection("docs")  
  
@app.get("/ask")  
def ask(q: str):  
 qv = emb.encode([q]).tolist()  
 res = col.query(query\_embeddings=qv, n\_results=4)  
 context = "\n".join(sum(res["documents"], []))  
 prompt = f'''Réponds précisément en citant les sources.\nContexte:\n{context}\nQuestion:{q}'''  
 # Exemple : appeler [Ollama](#ollama) local  
 r = requests.post("http://localhost:11434/api/generate", json={"model":"llama3.1:8b","prompt":prompt})  
 ans = r.json().get("response","(pas de réponse)")  
 return {"answer": ans, "sources": res["metadatas"]}

## 13) Checklist finale

* Objectif défini (tâche, contraintes)
* Données prêtes (corpus/Takeout nettoyé, chunké, métadonné)
* Index vectoriel construit ([FAISS](#faiss-facebook-ai-similarity-search)/[Chroma](#chroma)) + (option) reranker
* Modèle local opérationnel ([Ollama](#ollama)/llama.cpp/[vLLM](#vllm))
* Évaluation de base (playbook de questions, précision@k)
* Itérations ([prompting](#prompt-engineering), [embeddings](#embeddings), [QLoRA](#qlora-quantized-low-rank-adaptation) si besoin)
* Packaging ([API](#api-application-programming-interface) locale, scripts)

### Besoin d’un **script sur-mesure** pour parser ton Takeout et générer automatiquement :

* une bibliographie [IA](#ia-locale) (CSV),
* un corpus [RAG](#rag-retrieval-augmented-generation) (JSONL chunké + métadonnées),
* et un playbook de prompts basé sur tes vidéos préférées ?

Dis-moi le chemin de ton dossier Takeout et je fournis les scripts adaptés (Windows/Linux/macOS).