AYI NEDJIMI Consultants

# AWQ et GPTQ — Quantization de LLM pour Déploiement On-Premise

Catégorie : Intelligence Artificielle  Lecture : 17 min  Publié le : 12/04/2026  Auteur : Ayi NEDJIMI

*Maîtrisez AWQ et GPTQ pour quantizer vos LLM en INT4. Comparaison GGUF et bitsandbytes, déploiement vLLM, TGI, Ollama. Benchmarks Llama 3.1 70B inclus.*

Déployer un LLM de 70 milliards de paramètres en production on-premise nécessite, en précision native FP16, environ 140 Go de VRAM — soit deux GPU NVIDIA A100 80 Go ou quatre RTX 4090 24 Go. La quantization réduit cette empreinte mémoire d'un facteur 2 à 4 en convertissant les poids du modèle depuis le format flottant 16 bits vers des représentations entières 8 bits ou 4 bits, avec une dégradation de qualité souvent imperceptible pour les cas d'usage professionnels. AWQ (Activation-aware Weight Quantization) et GPTQ (Generative Pre-Trained Transformer Quantization) sont les deux méthodes de quantization post-entraînement qui dominent le déploiement on-premise en 2026. Chacune repose sur des principes mathématiques différents et offre des compromis distincts entre vitesse d'inférence, qualité de sortie et compatibilité runtime. Ce guide détaille le fonctionnement interne de chaque méthode, les compare face à GGUF et bitsandbytes, et fournit des recettes de déploiement pratiques avec AutoAWQ, AutoGPTQ, vLLM, Text Generation Inference (TGI) et Ollama. Les benchmarks présentés sont reproductibles et couvrent les modèles Llama 3.1, Mistral, Qwen 2.5 et Command R+ sur du matériel accessible (RTX 4090, A100). Les recommandations s'appuient sur notre expérience de déploiement LLM chez des clients soumis à des contraintes de souveraineté des données.

## Principes de la quantization

Un modèle de langage est essentiellement un ensemble de matrices de poids (weights) qui transforment les vecteurs d'entrée à travers des couches d'attention et de feed-forward. En FP16 (float16), chaque poids occupe 2 octets. La quantization consiste à représenter ces poids avec moins de bits — typiquement 8 bits (INT8) ou 4 bits (INT4) — tout en préservant au maximum la distribution statistique des activations du modèle.

### De FP16 à INT4 : ce que l'on gagne et ce que l'on perd

| Format | Bits/ poids | VRAM pour 70B | Perte qualité typique | Cas d'usage |
|---|---|---|---|---|
| FP32 | 32 | 280 Go | Référence | Entraînement uniquement |
| FP16 / BF16 | 16 | 140 Go | Négligeable | Inférence haute qualité |
| INT8 | 8 | 70 Go | < 1% perplexité | Production généraliste |
| INT4 (GPTQ/AWQ) | 4 | 35 Go | 1-3% perplexité | Production on-premise |
| INT4 (GGUF Q4_K_M) | ~4.8 | 38 Go | 1-2% perplexité | CPU + GPU offload |
| INT3/INT2 | 2-3 | 18-26 Go | 5-15% perplexité | Edge / expérimental |

## AWQ — Activation-aware Weight Quantization

AWQ, publié par le MIT (Song Han et al., 2023), part d'un constat simple : dans un réseau de neurones, tous les poids n'ont pas la même importance. Certains canaux (channels) produisent des activations de grande magnitude qui sont critiques pour la qualité de sortie. Quantizer uniformément tous les poids détruit l'information portée par ces canaux critiques. AWQ identifie les canaux importants en analysant les activations sur un petit jeu de calibration, puis applique un facteur d'échelle (scaling) avant la quantization pour protéger ces canaux. Pour comprendre les architectures d'inférence associées, consultez notre guide sur le speculative decoding.

### Fonctionnement interne

```
# Pseudo-code AWQ simplifié
# 1. Collecter les activations sur un jeu de calibration
activations = collect_activations(model, calibration_data)

# 2. Identifier les canaux importants (top 1% par magnitude)
channel_importance = activations.abs().mean(dim=0)
important_channels = channel_importance > threshold

# 3. Calculer les facteurs d'échelle optimaux
# s* = argmin || Q(W * diag(s)) * diag(s)^-1 * X - W * X ||
scales = optimize_scales(weights, activations)

# 4. Appliquer les scales et quantizer
scaled_weights = weights * scales
quantized_weights = quantize_to_int4(scaled_weights)

# L'inférence inverse les scales : output = dequant(W_q) / scales * input
```

### Quantization avec AutoAWQ

```
# Installation
pip install autoawq torch transformers

# Quantization d'un modèle
from awq import AutoAWQForCausalLM
from transformers import AutoTokenizer

model_path = "meta-llama/Llama-3.1-70B-Instruct"
quant_path = "llama-3.1-70b-instruct-awq-int4"

# Charger le modèle en FP16
model = AutoAWQForCausalLM.from_pretrained(model_path, device_map="auto")
tokenizer = AutoTokenizer.from_pretrained(model_path)

# Configuration de quantization
quant_config = {
    "zero_point": True,
    "q_group_size": 128,
    "w_bit": 4,
    "version": "GEMM"  # ou "GEMV" pour batch_size=1
}

# Quantizer (nécessite ~280 Go RAM ou swap pour 70B)
model.quantize(tokenizer, quant_config=quant_config)

# Sauvegarder
model.save_quantized(quant_path)
tokenizer.save_pretrained(quant_path)
```

## GPTQ — Post-Training Quantization

GPTQ (Frantar et al., 2022) utilise une approche différente basée sur la méthode OBQ (Optimal Brain Quantization). Au lieu de protéger certains canaux, GPTQ quantize les poids un par un en compensant l'erreur introduite par chaque poids quantizé sur les poids restants. Cette compensation itérative minimise l'erreur de reconstruction couche par couche. GPTQ traite chaque matrice de poids de manière indépendante, ce qui le rend parallélisable et relativement rapide. Pour approfondir les choix de modèles, consultez notre analyse des small language models.

## Quantization avec AutoGPTQ

```
# Installation
pip install auto-gptq torch transformers optimum

# Quantization GPTQ
from transformers import AutoModelForCausalLM, AutoTokenizer, GPTQConfig

model_id = "meta-llama/Llama-3.1-70B-Instruct"
quant_path = "llama-3.1-70b-instruct-gptq-int4"

tokenizer = AutoTokenizer.from_pretrained(model_id)

# Jeu de calibration (512-1024 exemples suffisent)
from datasets import load_dataset
dataset = load_dataset("wikitext", "wikitext-2-raw-v1", split="train")
calibration_data = [tokenizer(t, return_tensors="pt") for t in dataset["text"][:512] if
len(t) > 100]

# Configuration GPTQ
gptq_config = GPTQConfig(
    bits=4,
    group_size=128,
    dataset=calibration_data,
    desc_act=True,  # Activer l'activation ordering (meilleure qualité, plus lent)
    sym=False,       # Asymmetric quantization (meilleure qualité)
    damp_percent=0.01
)

# Quantizer
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=gptq_config,
    device_map="auto"
)

# Sauvegarder
model.save_pretrained(quant_path)
tokenizer.save_pretrained(quant_path)
```

## Comparaison : AWQ vs GPTQ vs GGUF vs bitsandbytes

| Critère | AWQ | GPTQ | GGUF (llama.cpp) | bitsandbytes (NF4) |
|---|---|---|---|---|
| Approche | Channel-aware scaling | Layer-wise OBQ | Block-wise k-quant | NormalFloat4 |
| Vitesse inférence | Rapide (kernels GEMM) | Rapide (Exllama/ Marlin) | Bonne (CPU+GPU) | Moyenne |
| Qualité (4 bits) | Excellente | Très bonne | Très bonne (Q4_K_M) | Bonne |
| Temps de quantization | 1-4h (70B) | 2-8h (70B) | 30min-2h (70B) | À la volée |
| Support vLLM | Natif | Natif (Marlin) | Non | Non |
| Support TGI | Natif | Natif | Non | Non |
| Support Ollama | Non | Non | Natif | Non |
| CPU offload | Non | Non | Oui (natif) | Non |
| Meilleur pour | vLLM / TGI prod | vLLM / TGI prod | Ollama / edge | Fine-tuning QLoRA |

**Choix de méthode : arbre de décision**
**Déploiement vLLM/TGI en production** → AWQ (meilleur compromis vitesse/qualité avec les kernels GEMM optimisés). **Déploiement Ollama ou llama.cpp** → GGUF Q4_K_M (seul format supporté, excellent en qualité). **Fine-tuning avec quantization** → bitsandbytes NF4 + QLoRA. **Budget GPU limité, besoin de CPU offload** → GGUF (llama.cpp gère nativement le split CPU/GPU). **Comparaison et évaluation** → quantizez en AWQ et GPTQ, benchmarkez sur votre dataset de test, gardez le meilleur.

## Déploiement pratique

### vLLM avec modèle AWQ

```
# Déploiement vLLM avec un modèle AWQ pré-quantizé
pip install vllm

# Serveur d'inférence
vllm serve TheBloke/Llama-3.1-70B-Instruct-AWQ \
  --quantization awq \
  --tensor-parallel-size 2 \
  --gpu-memory-utilization 0.90 \
  --max-model-len 16384 \
  --port 8000

# Test
curl http://localhost:8000/v1/completions \
  -H "Content-Type: application/json" \
  -d '{"model": "TheBloke/Llama-3.1-70B-Instruct-AWQ", "prompt": "Explique la quantization
AWQ en une phrase :", "max_tokens": 100}'
```

### TGI avec modèle GPTQ

```
# Déploiement TGI (Hugging Face Text Generation Inference) avec GPTQ
docker run --gpus all -p 8080:80 \
  -v /data/models:/data \
  ghcr.io/huggingface/text-generation-inference:latest \
  --model-id TheBloke/Llama-3.1-70B-Instruct-GPTQ \
  --quantize gptq \
  --num-shard 2 \
  --max-input-tokens 4096 \
  --max-total-tokens 8192
```

### Ollama avec modèle GGUF

```
# Ollama utilise exclusivement le format GGUF
# Créer un Modelfile personnalisé
cat > Modelfile << 'EOF'
FROM ./llama-3.1-70b-instruct-q4_K_M.gguf
PARAMETER temperature 0.7
PARAMETER num_ctx 8192
PARAMETER num_gpu 99
SYSTEM "Tu es un assistant technique expert en cybersécurité."
EOF

# Créer le modèle Ollama
ollama create llama31-70b-cyber -f Modelfile

# Tester
ollama run llama31-70b-cyber "Explique le principe de la quantization AWQ"
```

## Benchmarks : vitesse vs qualité vs mémoire

Les benchmarks suivants sont réalisés sur une configuration double RTX 4090 (48 Go VRAM total) avec Llama 3.1 70B Instruct. Le dataset d'évaluation est un mélange de 500 prompts techniques en français et en anglais, couvrant la génération de code, le résumé, le Q&A et l'analyse. Pour des stratégies d'optimisation des coûts, consultez notre guide d'optimisation d'inférence.

| Configuration | VRAM | tok/s (prompt) | tok/s (génération) | Perplexité (wiki) | Score MMLU |
|---|---|---|---|---|---|
| FP16 (référence, 4x A100) | 140 Go | 2800 | 42 | 3.12 | 79.8% |
| AWQ INT4 (2x 4090) | 36 Go | 3200 | 38 | 3.21 | 79.1% |
| GPTQ INT4 Marlin (2x 4090) | 36 Go | 3100 | 36 | 3.24 | 78.9% |
| GGUF Q4_K_M (2x 4090) | 38 Go | 2400 | 28 | 3.19 | 79.2% |
| GGUF Q4_K_M (CPU 64 cores) | 0 (RAM) | 180 | 8 | 3.19 | 79.2% |
| bitsandbytes NF4 (2x 4090) | 37 Go | 1800 | 22 | 3.28 | 78.5% |

## Cas d'usage : LLM on-premise souverain

De nombreuses organisations françaises, notamment dans les secteurs défense, santé et finance, déploient des LLM on-premise pour des raisons de souveraineté des données. La quantization rend possible l'hébergement de modèles performants (70B paramètres) sur du matériel accessible, réduisant le coût d'entrée d'un facteur 4 par rapport à un déploiement FP16. Pour les considérations de sécurité des embeddings, consultez notre guide sur la confidentialité des embeddings.

## FAQ — Questions fréquentes

### La quantization 4 bits dégrade-t-elle significativement la qualité des réponses ?

Pour les modèles de 13B paramètres et plus, la dégradation est rarement perceptible par un utilisateur humain. Les benchmarks montrent une perte de 0.5 à 3 points de perplexité et 0.5 à 1.5% sur MMLU — des différences qui se traduisent par des reformulations légèrement moins élégantes ou des erreurs factuelles marginalement plus fréquentes. En pratique, sur des tâches de production (résumé, Q&A, génération de code), les modèles quantizés INT4 sont indiscernables du FP16 dans 95% des cas. La dégradation devient notable uniquement sur les modèles petits (7B et moins) et les tâches nécessitant un raisonnement mathématique complexe.

### Faut-il refaire la quantization à chaque mise à jour du modèle ?

Oui. La quantization est spécifique aux poids d'un modèle donné. Quand Meta publie Llama 3.2, il faut re-quantizer — les poids quantizés de Llama 3.1 ne sont pas réutilisables. En pratique, la communauté (TheBloke, Hugging Face) publie les versions quantizées des modèles populaires dans les heures suivant leur sortie. Pour les modèles internes (fine-tunés), automatisez la quantization dans votre pipeline CI/CD de ML.

### AWQ ou GPTQ pour un déploiement vLLM en production ?

AWQ, pour trois raisons. Premièrement, les kernels AWQ GEMM dans vLLM sont légèrement plus rapides que les kernels GPTQ Marlin pour le continuous batching (scénario production avec requêtes concurrentes). Deuxièmement, la qualité AWQ est marginalement supérieure sur les modèles récents (Llama 3.x, Qwen 2.5) grâce à la préservation des canaux d'activation critiques. Troisièmement, AutoAWQ est plus rapide à exécuter (1-2h vs 4-8h pour GPTQ sur un modèle 70B), ce qui accélère les cycles de mise à jour. GPTQ reste pertinent si vous avez besoin de la fonctionnalité desc_act (activation ordering) pour maximiser la qualité au détriment de la vitesse, ou si votre runtime ne supporte que GPTQ.

---

**Ayi NEDJIMI Consultants** — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr