Research Part

Abdelrahman Osheba

February 14, 2026

Abstract

Large Language Models (LLMs) such as BERT and LLaMA can contain billions of parameters, making them expensive to store, deploy, and run. A core deployment bottleneck is **memory** (model weights + runtime cache) and **compute** (matrix multiplications). This report explains **quantization** as a practical solution to reduce model size and often improve inference efficiency. We cover the main quantization ideas (INT8/INT4, post-training vs. quantization-aware training), give real code examples (PyTorch + Hugging Face), and include a simple size comparison plot.

1 Why Large Models Are a Problem

1.1 Storage and Deployment

If a model has P parameters and each parameter is stored using b bits, the raw weight storage (ignoring metadata) is:

$$\text{Bytes} \approx \frac{P \cdot b}{8}. \tag{1}$$

For example, a **7B** parameter model:

$$\text{FP16 (16-bit)} \approx 7 \times 10^9 \cdot 2 \approx 14 \times 10^9 \text{ bytes} \approx 14 \text{ GB (decimal)}.$$

That is only the *weights*. In inference, LLMs also allocate memory for the **KV cache** (key/value tensors used by attention), which grows with sequence length and batch size.

1.2 Latency and Hardware Constraints

Even if you can store the model, real-time inference is limited by:

- **VRAM/RAM limits**: consumer GPUs often have 8–24GB VRAM.
- **Bandwidth**: reading weights from memory can dominate runtime.
- **Compute**: large matrix multiplications are heavy.

2 Quantization: The Main Idea

2.1 Definition

Quantization represents floating-point values (weights and/or activations) using fewer bits, e.g. INT8 or INT4, to reduce memory and sometimes speed up inference.

A common form is **affine (uniform) quantization**:

$$q = \text{clip}\Big(\text{round}\big(\tfrac{x}{s}\big) + z,\ q_{\min}, q_{\max}\Big), \tag{2}$$

$$\hat{x} = s \cdot (q - z), \tag{3}$$

where:

- x is the original floating value,
- q is the quantized integer,
- s is the scale,
- z is the zero-point (offset),
- $\hat{x}$ is the reconstructed (dequantized) value.

2.2 What Gets Quantized?

- **Weights-only quantization**: common for transformers; reduces model size a lot.
- **Weights + activations**: can further speed up matmuls and reduce runtime memory.
- **KV-cache quantization**: reduces attention cache memory in long-context inference (engine-dependent).

2.3 PTQ vs. QAT

- **Post-Training Quantization (PTQ)**: quantize after training; fast and widely used for deployment.
- **Quantization-Aware Training (QAT)**: train while simulating quantization effects; usually better accuracy, more effort.

3 How Much Size Do We Save?

Table 1 shows ideal weight-only size (no overhead). In practice, formats store extra scaling/metadata, so real files may be slightly larger.

Table 1: Ideal weight storage for a 7B model (approx.).

Precision	Bits/param	Bytes/param	Size (decimal GB)
FP16	16	2.0	≈ 14
INT8	8	1.0	≈ 7
INT4	4	0.5	≈ 3.5

3.1 Plot: Size vs. Precision

4 Quantization Methods Used in Practice (LLMs)

4.1 INT8 Quantization (Practical and Robust)

Many deployment stacks use 8-bit quantization because accuracy drop is often small, and hardware/software support is strong. Some methods treat *outlier* channels differently (e.g. keeping some operations in FP16) to preserve quality.

4.2 4-bit Quantization (Maximum Compression)

4-bit formats (e.g. NF4/FP4) can dramatically reduce memory. A popular strategy is:

- Quantize the base model to 4-bit for memory efficiency.
- Fine-tune using small trainable adapters (LoRA), known as **QLoRA**.

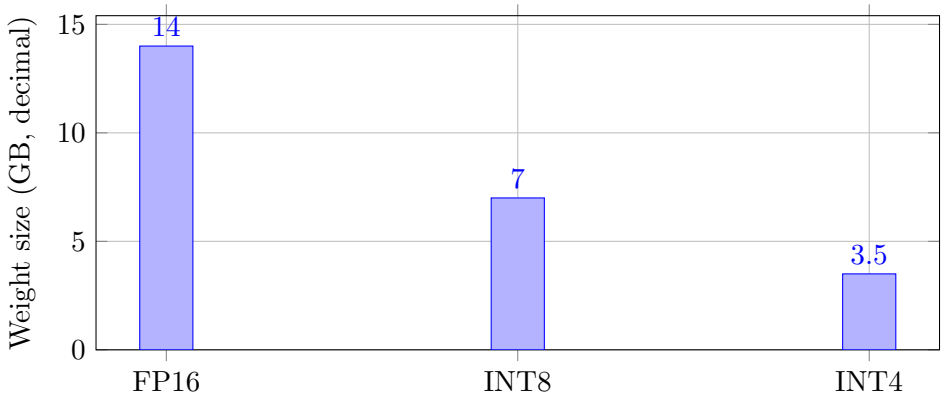

Figure 1: Ideal weight-only storage for a 7B model under different precisions.

4.3 Common Deployment Formats

- **Hugging Face + bitsandbytes**: easy INT8/INT4 loading in Python.
- **GPTQ/AWQ**: weight-only quantization schemes optimized for inference quality/speed.
- **GGUF (llama.cpp)**: a portable quantized format widely used for local inference.

5 Code Examples

5.1 Example 1: PyTorch Dynamic Quantization (CPU-Friendly)

Dynamic quantization is a quick win for transformer-like models (especially Linear layers) on CPU.

Listing 1: PyTorch dynamic quantization for Linear layers (weights-only INT8).

```
import torch
import torch.nn as nn
from torch.ao.quantization import quantize_dynamic  # torch >= 1.13+

# A tiny toy model (Linear layers similar to transformer MLP blocks)
class TinyMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1024, 4096),
            nn.ReLU(),
            nn.Linear(4096, 1024),
        )

    def forward(self, x):
        return self.net(x)

model_fp32 = TinyMLP().eval()

# Quantize only Linear layers to int8 (dynamic/weights-only)
model_int8 = quantize_dynamic(model_fp32, {nn.Linear}, dtype=torch.
    qint8)

# Quick sanity check
x = torch.randn(2, 1024)
```

```
with torch.no_grad():
    y_fp32 = model_fp32(x)
    y_int8 = model_int8(x)

print("FP32 output mean:", y_fp32.mean().item())
print("INT8 output mean:", y_int8.mean().item())
```

Tip: To compare sizes, you can save both models with `state_dict()` and compare file sizes on disk.

5.2 Example 2: Hugging Face Transformers INT8 Loading (bitsandbytes)

Listing 2: Load a model in 8-bit with Hugging Face Transformers.

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM,
    BitsAndBytesConfig

model_id = "meta-llama/Llama-2-7b-hf"    # example id (requires access)
bnb_config = BitsAndBytesConfig(load_in_8bit=True)

tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=bnb_config,
    device_map="auto"
)

prompt = "Explain quantization in one paragraph."
inputs = tokenizer(prompt, return_tensors="pt").to(model.device)
with torch.no_grad():
    out = model.generate(**inputs, max_new_tokens=80)
print(tokenizer.decode(out[0], skip_special_tokens=True))
```

5.3 Example 3: Hugging Face Transformers 4-bit (NF4) + QLoRA Setup

4-bit is typically used with LoRA adapters to fine-tune cheaply.

Listing 3: 4-bit NF4 loading config (common in QLoRA pipelines).

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM,
    BitsAndBytesConfig

model_id = "meta-llama/Llama-2-7b-hf"  # example id
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",          # normalized float 4
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=True
)

tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(
    model_id,
    quantization_config=bnb_config,
    device_map="auto"
)
```

Where LoRA comes in: you keep the 4-bit base model frozen, and train only low-rank adapter matrices (small number of parameters). This is the key idea behind QLoRA.

5.4 Example 4: GGUF Quantization for Local Inference (llama.cpp)

Many local inference workflows export to GGUF and quantize with llama.cpp tools.

Listing 4: Illustrative llama.cpp-style quantization command (GGUF).

```
# Example workflow (high-level):
# 1) Convert a HF model to GGUF (tooling depends on llama.cpp version)
# 2) Quantize to a chosen format (e.g., Q4_K_M, Q8_0)

# Quantize an existing GGUF file:
./llama-quantize model-f16.gguf model-q4_k_m.gguf Q4_K_M
```

6 Accuracy vs. Speed: Practical Notes

- INT8 usually preserves quality well and is a safe deployment default.
- INT4 gives the biggest memory savings, but may reduce accuracy more.
- Some layers (e.g. LayerNorm) are often kept in FP16/FP32 for stability.
- Real speedups depend on hardware support and kernel implementation (GPU/CPU, Tensor Cores, optimized GEMM).
- Quantizing weights reduces model memory, but long-context inference can still be dominated by KV-cache memory unless the engine also optimizes/quantizes it.

7 Conclusion

Quantization is one of the most practical techniques to deploy large models under limited memory and compute budgets. By moving from FP16 to INT8 or INT4, we can reduce the weight footprint by approximately 2× to 4×, enabling larger models on smaller GPUs/CPUs. Modern toolchains (PyTorch quantization and Hugging Face + bitsandbytes) make quantization accessible with only a few lines of code, while advanced formats (GPTQ/AWQ/GGUF) further optimize real-world inference.