Reducing Size of Large Language Models: Quantization Techniques, Theory and Practical Examples

Prepared by: Doaa Said

Abstract

Large pre-trained models (BERT family, GPT-family, LLaMA-like models) achieve state-of-the-art results but are extremely large and expensive to run. This document explains why these models are large, surveys quantization techniques to reduce storage and inference compute cost, and provides hands-on code examples and practical tips for applying quantization to real-world models. Bonus: LaTeX, math, code, and plots included.

Contents

1 Introduction: Why are models like BERT / LLaMA so large?

Modern transformer models grow in size for three core reasons:

1. **Model depth and width:** Transformers stack many layers (12–70+), each with multi-head attention and MLP blocks; parameter count scales roughly as $O(d^2)$ where d is hidden dimension.
2. **Embedding matrices:** Vocabulary embeddings and output (LM) heads are large (vocab size $\times$ embedding dim).
3. **Pretraining objectives and scale:** Training on massive corpora drives architecture scaling to improve performance (Kaplan et al., scaling laws).

Typical sizes:

- DistilBERT: tens of millions parameters (e.g. $\sim 66M$).
- BERT large: $\sim 340M$.
- LLaMA/GPT-style: hundreds of millions to tens of billions of parameters (7B, 13B, 70B, ...).

Memory examples: A 7B-parameter model stored in 16-bit floats needs $7 \times 10^9 \times 2$ bytes ≈ 14 GB just for weights. Running inference also needs activations, optimizer state (for training), etc.

2 Addressing the size problem: overview

Common approaches:

- **Smaller architectures:** Distillation, pruning, architecture search.
- **Parameter-efficient fine-tuning:** LoRA, adapters (reduce trainable parameters).
- **Model compression / quantization:** Reduce numerical precision of weights and/or activations.
- **Offloading / sharding:** Split model across multiple devices or use CPU + GPU offloading.

This document focuses on **quantization**.

3 Quantization: concept and basic math

Quantization reduces numerical precision of weights (and sometimes activations). If weights are stored in b bits instead of 32-bit floats, memory reduces roughly by factor $32/b$ ignoring bookkeeping.

3.1 Scalar uniform quantization (simple)

For a tensor w with min/max $w_{\min}, w_{\max}$, uniform affine quantization maps:

$$\hat{w} = \text{round}\left(\frac{w - w_{\min}}{\Delta}\right), \quad \Delta = \frac{w_{\max} - w_{\min}}{2^b - 1}.$$

Dequantization:

$$w \approx \hat{w} \cdot \Delta + w_{\min}.$$

3.2 Per-channel quantization

For convolutional/linear layers, using per-output-channel min/max or scale reduces error vs global quantization.

3.3 Non-uniform / learned quantization

Learned quantizers (k-means, product quantization) minimize quantization error but complicate inference.

3.4 Quantization error and propagation

Quantization introduces noise. For linear layers the expected squared error scales with Δ^2; deep networks can be robust to small quantization noise but sensitive if quantization is too aggressive or not matched to distribution of activations.

4 Quantization types and workflows

4.1 Post-training quantization (PTQ)

- **Dynamic quantization (aka run-time or post-training dynamic):** Weights quantized to int8, activations kept in float and quantized on the fly (no calibration).
- **Static (calibrated) PTQ:** Quantize weights and activations using calibration dataset to compute quantization ranges. More accurate than dynamic.
- **Advanced PTQ:** Techniques like GPTQ (Greedy Post-Training Quantization) for 4-bit quantization of language models with low accuracy loss.

4.2 Quantization-aware training (QAT)

Insert fake-quantization nodes during training so the model learns to be robust to quantized weights/activations. Best accuracy but requires additional training.

4.3 Mixed precision and low-bit tricks

- 8-bit (int8) quantization usually works well for transformer weights.
- 4-bit quantization with advanced methods (e.g., NF4, double quantization) is now practical with GPTQ / BitsAndBytes.
- Extreme quantization (2-bit, 1-bit) needs specialized algorithms (binary networks, XNOR nets) or heavy retraining.

5 Practical libraries and tools

- **PyTorch native quantization**: `torch.quantization` (PTQ and QAT for smaller models).
- **ONNX + ONNX Runtime**: convert model to ONNX and use `onnxruntime.quantization` for PTQ.
- **Hugging Face + bitsandbytes**: `bitsandbytes` supports 8-bit optimizers and 4-bit loading (bnb 4-bit) plus integration with transformers ‘from$_pretrained(..., load_in_8bit = True)‘or‘BitsAndBytesConfig$‘.
- **GPTQ / AWQ / QLoRA**: community methods for 4-bit/3-bit quantization of LLMs with minimal accuracy loss. QLoRA = Q + LoRA (quantized base + LoRA fine-tune).
- **Intel Neural Compressor / OpenVINO**: enterprise PTQ and QAT support.

6 Code examples

Below are practical examples you can run. These examples assume you have a Python environment with `transformers`, `torch`, `bitsandbytes`, `onnxruntime` installed as needed.

6.1 Example A: Dynamic (post-training) quantization with PyTorch (small classifier)

This is useful for BERT-like encoder models used for classification (not very large LLMs).

```python
from transformers import DistilBertForSequenceClassification,
    DistilBertTokenizerFast
import torch

model_name = "distilbert-base-uncased"
tokenizer = DistilBertTokenizerFast.from_pretrained(model_name)
model = DistilBertForSequenceClassification.from_pretrained(
    model_name, num_labels=9)
model.eval()

# Dynamic quantization: quantize linear layers to int8
quantized = torch.quantization.quantize_dynamic(
    model, {torch.nn.Linear}, dtype=torch.qint8
)

# Save quantized model
torch.save(quantized.state_dict(), "distilbert_qdyn.pth")
# Example inference
text = "This is an example text"
inputs = tokenizer(text, return_tensors="pt", truncation=True,
    padding=True)
with torch.no_grad():
    out = quantized(**inputs)
print(out.logits.argmax(-1))
```

Listing 1: Dynamic quantization (PyTorch) for a DistilBERT classifier.

Notes: PyTorch dynamic quantization works well for CPU inference and is straightforward. It does not quantize activations and often yields 2–4x speedups on CPU with small accuracy drop.

6.2 Example B: Static PTQ via ONNX Runtime

Convert to ONNX, then quantize.

```python
# 1) Export model to ONNX (use transformers.onnx.convert or manual
    export)
from transformers import AutoTokenizer,
    AutoModelForSequenceClassification
import torch

model = AutoModelForSequenceClassification.from_pretrained("
    distilbert-base-uncased", num_labels=9)
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")

# Build a dummy input
dummy = tokenizer("Hello world", return_tensors="pt", truncation=
    True, padding=True)
torch.onnx.export(model, (dummy["input_ids"], dummy["attention_mask"
    ]),
                 "model.onnx", input_names=["input_ids","
                     attention_mask"], output_names=["logits"],
                 opset_version=13)

# 2) Use onnxruntime to quantize
from onnxruntime.quantization import quantize_dynamic, QuantType
quantize_dynamic("model.onnx", "model_q.onnx", weight_type=QuantType.
    QInt8)
```

Listing 2: Export to ONNX and quantize with onnxruntime.quantization

Notes: ONNX Runtime supports dynamic and static (calibration-based) quantization. Static requires representative calibration data.

6.3 Example C: Hugging Face + bitsandbytes — 8-bit or 4-bit inference

For large LLMs, the easiest route for big models is bitsandbytes + transformers support.

```
# Requires: pip install bitsandbytes accelerate transformers
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

MODEL_NAME = "facebook/opt-6.7b" # example
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)

# 8-bit load
model = AutoModelForCausalLM.from_pretrained(
    MODEL_NAME,
    load_in_8bit=True, # bitsandbytes 8-bit
    device_map="auto" # automatic device mapping (accelerate)
)

input_ids = tokenizer("Write a short poem about data compression:",
    return_tensors="pt").input_ids.cuda()
outputs = model.generate(input_ids, max_new_tokens=64)
print(tokenizer.decode(outputs[0], skip_special_tokens=True))
```

Listing 3: Load model in 8-bit using bitsandbytes

Notes: ‘load$_{i}n_{8}bit = True‘usesbitsandbytesunder - the - hood.For4 - bitloading, newerAPIuses‘bnb_{c}onfig‘/‘BitsAndBytesConfig‘(seeadvancedsection).$

6.4 Example D: QLoRA style: quantize to 4-bit then fine-tune LoRA adapters

QLoRA (as used in recent research) loads model in 4-bit (bnb), freeze base weights, and fine-tunes LoRA adapters. Below is a **conceptual** snippet — check your installed bitsandbytes/transformers docs for exact arguments.

```python
from transformers import AutoTokenizer, AutoModelForCausalLM
from peft import LoraConfig, get_peft_model
from transformers import BitsAndBytesConfig

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4"
)

tokenizer = AutoTokenizer.from_pretrained("decapoda-research/llama-7
    b-hf")
model = AutoModelForCausalLM.from_pretrained(
    "decapoda-research/llama-7b-hf",
    quantization_config=bnb_config,
    device_map="auto"
)

lora_config = LoraConfig(r=8, lora_alpha=16, target_modules=["q_proj
    ","v_proj"], task_type="CAUSAL_LM")
model = get_peft_model(model, lora_config)
# Now fine-tune model normally using Trainer/accelerate; only LoRA
    params train.
```

Listing 4: QLoRA-style: load small model in 4-bit and apply LoRA (conceptual)

Caveat: the exact function names and parameter names evolve rapidly. Before running, check the versions of ‘transformers‘, ‘bitsandbytes‘, and ‘peft‘ you have.

7 Empirical graph: model size vs bit-width

We show a theoretical plot of compression factor vs bits per weight. Save this document and compile; the plot is produced by `pgfplots`.

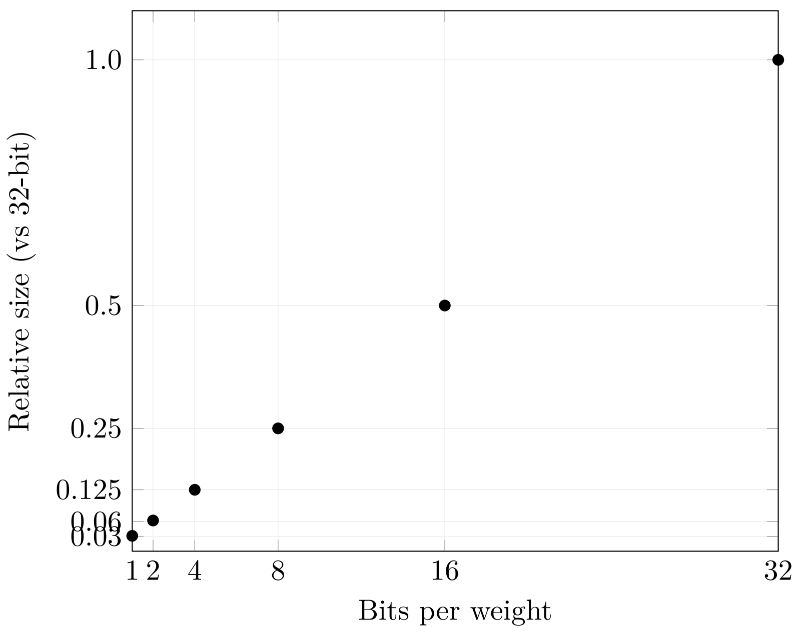

Figure 1: Relative model size versus bits per weight. Example: 4-bit storage reduces size by 8x vs 32-bit.

8 Practical tips & trade-offs

- **8-bit inference** is often the lowest-hassle: good compression, wide tool support (bitsandbytes, ONNX).
- **4-bit** gains more compression (8x vs 32-bit) but needs careful algorithms (GPTQ, NF4) or QAT for small accuracy loss.
- **QAT** delivers best accuracy for extreme quantization but requires training compute.
- **Calibration data** matters for PTQ — pick a representative dataset for activation ranges.
- **Per-channel scales** reduce quantization error for linear layers.
- **Evaluation is crucial**: measure downstream metrics (accuracy / F1) after quantization—different tasks react differently.

9 Advanced: GPTQ, AWQ, NF4 and community methods

- **GPTQ (Greedy PTQ)**: greedy search to minimize layer-wise quantization error in LLMs; yields very good 3/4-bit performance in many cases.

- **AWQ** and other new methods: further refine quantization and address per-tensor distribution mismatch.
- **NF4**: a special 4-bit format optimized for LLM weight distributions (used in bitsandbytes / QLoRA workflows).

10 Example workflow: from a fine-tuned DistilBERT classifier to a quantized deployable artifact

1. Fine-tune model (or use pretrained) with standard float32 training.
2. Evaluate baseline on validation set and save checkpoint.
3. Try dynamic quantization (PyTorch) for CPU deployment.
4. If GPU is target: try bitsandbytes 8-bit load for large models or ONNX-RT quantization for smaller models.
5. For aggressive compression (4-bit): use GPTQ or QLoRA approach with representative calibration / LoRA fine-tuning.
6. Validate on held-out test set and compare metrics.

11 Suggested references and further reading

- Jacob et al., Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference (Google).
- Micikevicius et al., Mixed Precision Training.
- Hugging Face docs: `https://huggingface.co/docs/transformers`
- bitsandbytes repo and docs: `https://github.com/facebookresearch/bitsandbytes`
- ONNX Runtime quantization docs.
- Papers on GPTQ, QLoRA (search for "GPTQ quantization LLM", "QLoRA").

12 Appendix: runnable end-to-end example (DistilBERT quantization + inference)

12.1 Train small classifier (sketch)

```
# 1) Fine-tune DistilBERT normally using Trainer (sketch)
from transformers import DistilBertTokenizerFast,
    DistilBertForSequenceClassification, Trainer, TrainingArguments
# prepare dataset tokenized...
training_args = TrainingArguments(output_dir="./out",
    num_train_epochs=3, per_device_train_batch_size=16)
trainer = Trainer(model=model, args=training_args, train_dataset=
    train_dataset, eval_dataset=val_dataset)
trainer.train()
trainer.save_model("./out/model")
```

12.2 Dynamic quantization + inference

```
# 2) Dynamic quantization
import torch
from transformers import AutoModelForSequenceClassification,
    AutoTokenizer

model = AutoModelForSequenceClassification.from_pretrained("./out/
    model")
tokenizer = AutoTokenizer.from_pretrained("./out/model")
model.eval()
qmodel = torch.quantization.quantize_dynamic(model, {torch.nn.Linear
    }, dtype=torch.qint8)
# Save
torch.save(qmodel.state_dict(), "distilbert_dyn_q.pth")
```

12.3 bitsandbytes 8-bit inference (LLM)

```
# 3) bitsandbytes 8-bit example
from transformers import AutoTokenizer, AutoModelForCausalLM
tokenizer = AutoTokenizer.from_pretrained("facebook/opt-6.7b")
model = AutoModelForCausalLM.from_pretrained("facebook/opt-6.7b",
    load_in_8bit=True, device_map="auto")
print(tokenizer.decode(model.generate(tokenizer("Hello",
    return_tensors="pt").input_ids.cuda())[0]))
```

13 Conclusion

Quantization is a practical and powerful tool to reduce model size and inference cost. Choose strategies based on:

- target hardware (CPU vs GPU),
- acceptable accuracy trade-off,
- timeframe (PTQ is fast; QAT and GPTQ need more compute).

Notes: Tools and APIs evolve rapidly. Before running 4-bit or QLoRA-style code, check the versions of `transformers`, `bitsandbytes`, and `peft` in your environment; parameter names occasionally change across releases.