

Comprehensive Evaluation of Quantization Methods for Edge LLM Deployment

Zahraa Selim, Menna Hamed, Wesam Ahmed, Sohyla Said, Sara Basheer, and Rami Zewail

Computer Science and Engineering Department

Egypt-Japan University of Science and Technology (E-JUST)

Alexandria, Egypt

{zahraa.selim, menna.hamed, rami.zewail}@ejust.edu.eg

I. METHODOLOGY

A. Base Model and Hardware

This study utilizes **Mistral-7B-v0.1** [1] as the baseline model, selected for its parameter efficiency and strong performance profile. All experiments are conducted on a single **NVIDIA Tesla T4 GPU (16GB VRAM)** to simulate realistic edge deployment scenarios. The software environment includes PyTorch 2.x, Transformers 4.x, and quantization-specific libraries (bitsandbytes, AutoGPTQ, AutoAWQ, HQQ).

B. Compression Techniques

1) GPTQ

GPTQ (Generative Pre-trained Transformer Quantization) is a post-training quantization method that quantizes weights to 2-8 bits while maintaining model quality through layer-wise optimization [2].

Method Overview:

GPTQ addresses the computational challenges of optimal quantization by reformulating the problem as a series of layer-wise optimizations. For each layer's weight matrix $W \in \mathbb{R}^{d_{out} \times d_{in}}$, GPTQ solves:

$$\min_{\hat{W}} \|WX - \hat{W}X\|_2^2 \quad (1)$$

where W is the original weight matrix, \hat{W} is the quantized weight matrix, and $X \in \mathbb{R}^{d_{in} \times n}$ represents calibration inputs with n samples.

The key innovation is the use of approximate second-order information through the Hessian matrix:

$$H = \frac{2}{n} XX^T \quad (2)$$

GPTQ employs a greedy coordinate descent approach, quantizing weights column-by-column while adjusting subsequent weights to compensate for quantization error. For each weight w_q being quantized:

$$\hat{w}_q = \text{quant}(w_q), \quad \delta_q = w_q - \hat{w}_q \quad (3)$$

The quantization error δ_q is then propagated to remaining weights:

$$w_{q'} \leftarrow w_{q'} - \delta_q \cdot \frac{H_{qq'}}{H_{qq}} \quad \forall q' > q \quad (4)$$

This Optimal Brain Quantization (OBQ)-inspired approach ensures that quantization errors are minimized across the entire weight matrix [3].

Group-wise Quantization:

To balance precision and compression, GPTQ applies quantization in groups. Each group of g weights (e.g., $g = 128$) shares quantization parameters (scale s and zero-point z):

$$\hat{w}_i = \text{round}\left(\frac{w_i}{s}\right) - z, \quad s = \frac{\max(w_g) - \min(w_g)}{2^b - 1} \quad (5)$$

where b is the number of bits (2, 3, or 4). Smaller group sizes increase accuracy at the cost of additional metadata storage.

Activation Ordering:

The optional `desc_act` parameter reorders columns by activation magnitude to group similar-scale weights, potentially reducing quantization error:

$$\pi = \text{argsort}(\text{diag}(H)), \quad W \leftarrow W[:, \pi] \quad (6)$$

Quantization Process:

We quantized Mistral-7B-v0.1 at three precision levels (4-bit, 3-bit, and 2-bit) using auto-gptq on a P100 GPU, with the following configuration:

- Base model: mistralai/Mistral-7B-v0.1 (7.24B parameters)
- Bits: 2, 3, 4
- Group size: 128 (default), 64 (quality experiment)
- Calibration dataset: WikiText-2 (128-512 samples, max length 512)

This work was conducted at Egypt-Japan University of Science and Technology (E-JUST).

- Damping factor: 0.01 (regularization for Hessian stability)
- desc_act: False (basic), True (quality experiment)
- Memory optimization: cache_examples_on_gpu=False
- Environment: PYTORCH_CUDA_ALLOC_CONF="expandable_segments:True"

Implementation:

```

1 import torch
2 from auto_gptq import AutoGPTQForCausalLM,
   BaseQuantizeConfig
3 from datasets import load_dataset
4
5 # Prepare calibration data
6 def prepare_calibration_data(tokenizer, n_samples=128,
7                             max_length=512):
8     dataset = load_dataset("wikitext", "wikitext-2-raw
9                             -v1",
10                            split="train")
11     dataset = dataset.filter(lambda x: len(x["text"])
12                             > 200)
13
14     examples = []
15     for i in range(min(n_samples, len(dataset))):
16         text = dataset[i]["text"]
17         tokenized = tokenizer(
18             text,
19             return_tensors="pt",
20             max_length=max_length,
21             truncation=True,
22             padding=False
23         )
24         examples.append({
25             "input_ids": tokenized["input_ids"],
26             "attention_mask": tokenized["
27                 attention_mask"]
28         })
29     return examples
30
31 calibration_data = prepare_calibration_data(tokenizer)
32
33 # Quantize model
34 model = AutoGPTQForCausalLM.from_pretrained(
35     "mistralai/Mistral-7B-v0.1",
36     quantize_config=BaseQuantizeConfig(
37         bits=4,
38         group_size=128,
39         desc_act=False,
40         damp_percent=0.01
41     ),
42     max_memory={0: "10GiB", "cpu": "50GiB"}
43 )
44
45 # Layer-wise quantization with error compensation
46 model.quantize(
47     calibration_data,
48     use_triton=False,
49     batch_size=1,
50     cache_examples_on_gpu=False # Critical for P100
51 )
52
53 # Save quantized weights
54 model.save_quantized("./mistral-7b-gptq-4bit",
55                      use_safetensors=True)

```

The quantization process takes approximately 22 minutes per model on a P100 GPU, with memory usage optimized through CPU offloading and disabling GPU caching of calibration data.

Optimized Inference:

For evaluation, we employed ExLlamaV2, an optimized inference engine for GPTQ models that provides 2-3x faster inference through specialized CUDA kernels and efficient

memory management:

```

1 from exllamav2 import (ExLlamaV2, ExLlamaV2Config,
2                         ExLlamaV2Cache,
3                         ExLlamaV2Tokenizer)
4 from exllamav2.generator import (
5     ExLlamaV2StreamingGenerator,
6     ExLlamaV2Sampler)
7
8 # Load with ExLlamaV2
9 config = ExLlamaV2Config()
10 config.model_dir = "./mistral-7b-gptq-4bit"
11 config.prepare()
12 config.max_seq_len = 4096
13
14 model = ExLlamaV2(config)
15 cache = ExLlamaV2Cache(model, lazy=True)
16 model.load_autosplit(cache)
17
18 tokenizer = ExLlamaV2Tokenizer(config)
19
20 # Generate with optimized kernels
21 generator = ExLlamaV2StreamingGenerator(model, cache,
22                                         tokenizer)
23
24 settings = ExLlamaV2Sampler.Settings()
25 settings.temperature = 0.7
26 settings.top_p = 0.9
27
28 input_ids = tokenizer.encode(prompt)
29 generator.begin_stream(input_ids, settings)

```

ExLlamaV2 optimizations include:

- Custom CUDA kernels for 2/3/4-bit matrix multiplication
- Fused attention and MLP operations
- Lazy KV cache loading to reduce memory overhead
- Efficient memory layout for quantized weights

Quality Enhancement Experiments:

To investigate quality-efficiency trade-offs, we conducted additional experiments beyond baseline configurations:

- 1) **Activation ordering** (desc_act=True): Reorders weight columns by activation magnitude, grouping similar-scale weights to reduce intra-group variance:

```

1 quantize_config=BaseQuantizeConfig(
2     bits=3,
3     group_size=128,
4     desc_act=True # Enable activation ordering
5 )

```

- 2) **Smaller group size** (group_size=64): Reduces quantization granularity, allowing finer-grained scale/zero-point adaptation at the cost of 2x metadata storage:

```

1 quantize_config=BaseQuantizeConfig(
2     bits=3,
3     group_size=64 # Halve group size
4 )

```

- 3) **Increased calibration** (512 samples): For 2-bit quantization, we quadrupled calibration data to provide more representative Hessian approximation:

```

1 calibration_data = prepare_calibration_data(
2     tokenizer,

```

```

3     n_samples=512 # 4x increase
4 )

```

These configurations were systematically evaluated to characterize the quality-compression frontier across 2-4 bit precision levels, with results presented in Section ??.

Theoretical Compression:

For Mistral-7B with 7.24B parameters, theoretical model sizes are:

- FP16 baseline: $7.24 \times 2 = 14.48$ GB
- 4-bit GPTQ: $7.24 \times 0.5 + \text{overhead} \approx 3.62$ GB (4.0x compression)
- 3-bit GPTQ: $7.24 \times 0.375 + \text{overhead} \approx 2.72$ GB (5.3x compression)
- 2-bit GPTQ: $7.24 \times 0.25 + \text{overhead} \approx 1.81$ GB (8.0x compression)

where overhead includes quantization metadata (scales, zero-points) and unquantized components (embeddings, layer norms).

2) NF4 (NormalFloat 4-bit)

NF4 utilizes the quantile-based NormalFloat data type, which is information-theoretically optimal for zero-centered normal distributions. Double quantization is employed to further minimize memory footprint by quantizing the quantization constants themselves [?].

Configuration:

- Quantization type: NF4 (NormalFloat4)
- Double quantization: Enabled
- Compute dtype: float16
- Group size: Per-tensor

Implementation:

```

1 from transformers import BitsAndBytesConfig
2 import torch
3
4 nf4_config = BitsAndBytesConfig(
5     load_in_4bit=True,
6     bnb_4bit_quant_type="nf4",
7     bnb_4bit_use_double_quant=True,
8     bnb_4bit_compute_dtype=torch.float16
9 )
10
11 model = AutoModelForCausalLM.from_pretrained(
12     "mistralai/Mistral-7B-Instruct-v0.1",
13     quantization_config=nf4_config,
14     device_map="auto"
15 )

```

3) AWQ (Activation-aware Weight Quantization)

AWQ protects salient weights based on activation magnitudes, preserving the top 1% most impactful weights with per-channel scaling [4].

Configuration:

- Bits: 4
- Group size: 128
- Zero point: True
- Version: GEMM

Implementation:

```

1 from awq import AutoAWQForCausalLM
2
3 model = AutoAWQForCausalLM.from_quantized(
4     "TheBloke/Mistral-7B-Instruct-v0.1-AWQ",
5     fuse_layers=True,
6     safetensors=True
7 )

```

4) HQQ (Half-Quadratic Quantization)

HQQ performs fast, calibration-free quantization optimizing a custom loss function with on-the-fly parameter optimization [?].

Configuration:

- Bits: 4
- Group size: 64
- Axis: 1 (row-wise)
- Compute dtype: float16

Implementation:

```

1 from hqq.models.hf.base import AutoHQQHFModel
2 from hqq.core.quantize import BaseQuantizeConfig
3
4 quant_config = BaseQuantizeConfig(
5     nbits=4, group_size=64, axis=1
6 )
7
8 model = AutoHQQHFModel.from_pretrained(
9     "mistralai/Mistral-7B-Instruct-v0.1",
10    torch_dtype=torch.float16
11 )
12 model.quantize_model(
13    quant_config=quant_config, device='cuda'
14 )

```

5) AutoRound

AutoRound employs adaptive quantization with iterative optimization for per-layer parameter tuning.

Configuration:

- Bits: 4
- Group size: 128
- Samples: 64
- Iterations: 50
- Batch size: 4
- Learning rate: 5e-3

Implementation:

```

1 from auto_round import AutoRound
2

```

```

3  ar = AutoRound(
4      model=MODEL_PATH,
5      scheme="W4A16",
6      bits=4,
7      group_size=128,
8      nsamples=64,
9      iters=50,
10     lr=5e-3,
11     seqlen=1024,
12     batch_size=4,
13     amp_dtype=torch.float16
14 )
15
16 ar.quantize_and_save(
17     output_dir=OUTPUT_DIR,
18     format="auto_round"
19 )

```

C. Evaluation Metrics

We assess compressed models across two primary dimensions: computational efficiency and task performance.

1) Efficiency Metrics

a) Latency Measurements

- **Average Latency:** Mean time per generated token (ms)
- **Time-to-First-Token (TTFT):** Initial response latency
- **Prefill vs. Decode:** Separate measurement of prompt processing vs. autoregressive generation

b) Throughput and Memory

- **Throughput:** Tokens per second (tok/s)
- **Peak Memory:** Maximum GPU memory (MB)
- **Model Size:** Disk storage requirements (GB)

c) Computational Efficiency

- **Model FLOPs Utilization (MFU):** Percentage of theoretical peak hardware FLOPs achieved:

$$\text{MFU} = \frac{\text{Achieved FLOPs/s}}{\text{Peak Hardware FLOPs/s}} \times 100\% \quad (7)$$

- **Energy Consumption:** Estimated energy per token (mJ):

$$E = (P_{\text{TDP}} - P_{\text{idle}}) \times t_{\text{inference}} \quad (8)$$

$$\text{where } P_{\text{idle}} \approx 0.3 \times P_{\text{TDP}}$$

2) Performance Benchmarks

We evaluate models using established language modeling benchmarks:

a) Language Modeling

Perplexity: Measured on WikiText-2 test set using sliding window evaluation (stride 512):

$$\text{PPL}(W) = \exp \left(-\frac{1}{N} \sum_{i=1}^N \ln P(w_i | w_{<i}) \right) \quad (9)$$

b) Core Task Benchmarks

- **HellaSwag** (0-shot): Commonsense reasoning via sentence completion
- **ARC-Easy** (0-shot): Grade-school science questions
- **ARC-Challenge** (0-shot): Challenge-level scientific reasoning
- **GSM8K** (8-shot): Grade-school math word problems
- **MMLU** (5-shot): Multi-domain knowledge across 57 subjects
- **HumanEval** (0-shot): Python code generation (pass@1)

All evaluations use the Language Model Evaluation Harness [5] with consistent hyperparameters for reproducibility.

II. EXPERIMENTAL RESULTS

A. Computational Efficiency Analysis

Table I presents comprehensive efficiency metrics on Tesla T4 hardware.

TABLE I
EFFICIENCY METRICS ON NVIDIA TESLA T4

Method	Disk (GB)	Peak Mem (MB)	TTFT (ms)	Decode (ms/T)	TPut (T/s)	Energy (mJ/T)
FP16	13.49	7013	68.84	61.36	15.84	3086
NF4	3.74	1859	177.16	79.09	12.30	3975
AWQ	3.87	1869	79.25	77.22	12.79	3855
HQQ	3.74	4721	664.73	649.84	1.51	32421
GPTQ	3.87	4415	1113.00	1117.76	0.88	55703

Key Findings:

- 1) **Hardware Compatibility Crisis:** GPTQ and HQQ exhibit severe performance degradation on Tesla T4, with throughputs below 2 tok/s and energy consumption exceeding 30,000 mJ/token. This represents a **10-18× slowdown** compared to NF4/AWQ, rendering them impractical for this hardware.
- 2) **NF4 vs. AWQ Trade-off:** NF4 achieves 12.30 tok/s with 3975 mJ/token, while AWQ achieves 12.79 tok/s with 3855 mJ/token. AWQ provides marginal efficiency gains (3% better energy), but as shown in Section IV-B, this comes at the cost of task performance.
- 3) **Memory Efficiency:** All 4-bit methods achieve approximately 3.6× compression (13.49 GB \rightarrow 3.7-3.9 GB), validating theoretical compression ratios. However, peak memory usage varies significantly, with HQQ and GPTQ requiring 2.5× more runtime memory due to kernel overhead.

B. General Task Performance

Table II presents performance across reasoning benchmarks.

TABLE II
PERFORMANCE COMPARISON ACROSS BENCHMARKS

Method	Perplexity (↓)	HellaSwag (0-shot)	ARC-Easy (0-shot)	ARC-Challenge (0-shot)	GSM8K (8-shot)	MMLU (5-shot)	HumanEval (0-shot)
FP16	12.79	0.72	0.76	0.58	0.36	1.00	0.05
NF4	13.02	0.70	0.75	0.58	0.27	0.55	0.05
GPTQ	12.85	0.68	0.75	0.60	—	—	0.05
AWQ	13.47	—	—	—	—	—	—
HQQ	13.50	0.69	0.72	0.50	—	—	—

TABLE III
AVERAGE ACCURACY AND DEGRADATION

Method	Avg Acc	PPL Inc	Acc Drop	Tasks
FP16	0.57	—	—	6
NF4	0.52	+1.80%	-0.05	6
GPTQ	0.52	+0.47%	-0.05	6
AWQ	—	+5.32%	—	6
HQQ	—	—	—	6

Performance Summary:

Task-Specific Analysis:

- 1) **Knowledge Retrieval Robustness:** ARC-Challenge and ARC-Easy maintain near-FP16 performance (0.58 vs 0.58 for NF4), indicating that factual knowledge retrieval is preserved under quantization.
- 2) **Mathematical Reasoning Degradation:** GSM8K experiences 25% accuracy drop (0.36 \rightarrow 0.27), suggesting multi-step reasoning is more sensitive to quantization than pattern-matching tasks.
- 3) **Code Generation Stability:** HumanEval maintains perfect parity (0.05 across all methods), likely due to the task’s low baseline difficulty for 7B models.
- 4) **Perplexity Mismatch:** Despite similar perplexity scores, methods show different task performance profiles, confirming that perplexity alone is insufficient for evaluating practical model quality.

III. DISCUSSION

A. Hardware-Algorithm Compatibility as Primary Factor

Our most critical finding is that **hardware compatibility dominates compression effectiveness**. While theoretical compression ratios are identical (3.6 \times for all 4-bit methods), practical performance varies by 18 \times on Tesla T4.

Hypothesis: GPTQ and HQQ rely on INT4 tensor cores and optimized GEMM kernels introduced in Ampere architecture. On Turing GPUs (Tesla T4), these operations fall back to inefficient FP16 emulation, causing severe overhead.

Implication: For edge deployment on consumer/previous-generation GPUs, **distribution-based quantization (NF4) is the only viable option**. Kernel-dependent methods should be reserved for latest-generation hardware.

B. Task-Dependent Sensitivity

Quantization impacts tasks differentially:

- **Preserved:** Knowledge retrieval (ARC), commonsense reasoning (HellaSwag)
- **Degraded:** Mathematical reasoning (GSM8K -25%)
- **Stable:** Code generation (HumanEval)

This suggests that quantization affects **deep reasoning chains** more than **pattern matching**. For applications requiring complex multi-step logic, either fine-tuning or hybrid precision strategies may be necessary.

C. The NF4 vs. AWQ Trade-off

AWQ offers marginal efficiency gains (3% lower energy) but shows higher perplexity degradation (+5.32% vs +1.80% for NF4). For general-purpose deployment, **NF4 provides better performance preservation** despite slightly higher energy consumption.

However, AWQ may be preferable for:

- Latency-critical applications where 3% speedup matters
- Tasks where perplexity degradation is acceptable
- Scenarios prioritizing energy efficiency over quality

D. Deployment Guidelines

Based on our comprehensive analysis, we provide prescriptive recommendations:

IV. CONCLUSION

This study provides the first comprehensive hardware-aware benchmarking of modern quantization methods on consumer-grade edge hardware. Our key contributions include:

- 1) **Hardware Compatibility Analysis:** Demonstrating that algorithm-hardware compatibility is the primary determinant of practical performance, with kernel-dependent

TABLE IV
QUANTIZATION METHOD SELECTION GUIDE

Scenario	Recommended Method
Turing GPUs (T4, RTX 20-series)	NF4 (only viable option)
Ampere+ GPUs (A100, RTX 30/40)	AWQ or GPTQ (test both)
Math-heavy applications	NF4 + targeted fine-tuning
Knowledge retrieval	Any 4-bit method
Maximum energy efficiency	AWQ (if compatible)
General-purpose deployment	NF4 (best balance)

methods experiencing 10-18× slowdown on Turing GPUs.

- 2) **Task-Specific Evaluation:** Establishing that quantization impacts tasks differentially, with mathematical reasoning showing 25% degradation while knowledge retrieval remains robust.
- 3) **Prescriptive Guidelines:** Identifying NF4 as the optimal method for Tesla T4 and similar hardware, achieving 3.6× compression with minimal performance loss and superior hardware compatibility.

For Turing-generation hardware, NF4 represents the **only viable quantization method**, achieving stable 12.30 tok/s throughput with reasonable energy consumption (3975 mJ/-token). While AWQ offers marginal efficiency gains, NF4’s superior task performance and hardware stability make it the recommended choice for general-purpose edge deployment.

Future Work: Investigating hybrid precision strategies where critical layers (attention heads for reasoning tasks) retain higher precision while feedforward networks are aggressively quantized may resolve the mathematical reasoning degradation observed in our study.

ACKNOWLEDGMENTS

This research was conducted at Egypt-Japan University of Science and Technology (E-JUST). We thank the Anthropic team for access to computational resources.

REFERENCES

- [1] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de Las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, “Mistral 7b,” *arXiv preprint arXiv:2310.06825*, 2023.
- [2] E. Frantar, S. Ashkboos, T. Hoefer, and D. Alistarh, “Gptq: Accurate post-training quantization for generative pre-trained transformers,” in *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 2023, pp. 10017–10030.
- [3] E. Frantar and D. Alistarh, “Optimal brain compression: A framework for accurate post-training quantization and pruning,” in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 4475–4488.
- [4] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, “Awq: Activation-aware weight quantization for llm compression and acceleration,” *arXiv preprint arXiv:2306.00978*, 2023, related quantization method.
- [5] L. Gao, J. Tow, B. Abbasi, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, A. Le Noac’h *et al.*, “A framework for few-shot language model evaluation,” *Version v0.4.0*, 2023, lm-eval-harness for task evaluation. [Online]. Available: <https://github.com/EleutherAI/lm-evaluation-harness>