Compression Techniques for Mistral-7B: A Comprehensive Evaluation of Efficiency and Performance

Zahraa Selim Menna Hamed Wesam Ahmed Sohyla Said Sara Basheer

Under the supervision of
Prof. Rami Zewail

Department of Computer Science and Engineering,
Egypt-Japan University of Science and Technology (E-JUST)

1 Experimental Setup

1.1 Overview

This study conducts a comprehensive comparative evaluation of multiple compression techniques applied to the Mistral-7B model. Our experimental protocol follows a two-phase approach: (1) compress the base model and evaluate compression impact, and (2) fine-tune the compressed models on downstream tasks and re-evaluate to measure performance recovery. All experiments are conducted under resource-constrained environments to reflect realistic deployment scenarios.

1.2 Environment and Resources

All experiments were conducted on cloud-based platforms with the following specifications:

- **Platforms:** Kaggle and Google Colab free-tier instances
- **Hardware:** NVIDIA Tesla T4 GPU (16GB VRAM)
- **Software:** PyTorch 2.x, Transformers 4.x, bitsandbytes, AutoGPTQ, AutoAWQ
- **Base Model:** Mistral-7B (FP16 baseline, 13.49 GB model size)

These resource constraints (limited VRAM and compute) reflect the target deployment scenario for compressed models and ensure our findings are applicable to practical edge and consumer-grade hardware settings.

1.3 Compression Techniques

1.3.1 Quantization Methods

We evaluate four state-of-the-art post-training quantization techniques:

- **NF4 (4-bit NormalFloat):** Uses an information-theoretically optimal data type for normally distributed weights, implemented via bitsandbytes with double quantization enabled.

  Parameters:

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

  Implementation:

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

nf4_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.float16
)

model = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-Instruct-v0.1",
    quantization_config=nf4_config,
    device_map="auto"
)
```

Listing 1: NF4 Quantization with BitsAndBytes

- **GPTQ:** Layer-wise quantization that minimizes reconstruction error using Hessian information, applied at 4-bit precision with group size of 128. We use TheBloke's pre-quantized model to avoid the computationally expensive quantization process.

  Parameters:

  - Bits: 4
  - Group size: 128
  - Dataset: C4 (used during quantization)
  - Activation order: Optimized via Hessian

  Implementation:

```python
from auto_gptq import AutoGPTQForCausalLM

model = AutoGPTQForCausalLM.from_quantized(
    "TheBloke/Mistral-7B-Instruct-v0.1-GPTQ",
    device="cuda:0",
    use_safetensors=True
)
```

Listing 2: GPTQ Quantization (Pre-quantized Model)

- **AWQ (Activation-aware Weight Quantization):** Protects salient weights based on activation magnitudes, using 4-bit quantization with per-channel scaling. Similar to GPTQ, we use a pre-quantized model.

  Parameters:

  - Bits: 4
  - Group size: 128
  - Zero point: True
  - Version: GEMM (optimized matrix multiplication)

  Implementation:

```python
from awq import AutoAWQForCausalLM

model = AutoAWQForCausalLM.from_quantized(
    "TheBloke/Mistral-7B-Instruct-v0.1-AWQ",
    fuse_layers=True,
    safetensors=True
)
```

Listing 3: AWQ Quantization (Pre-quantized Model)

- **HQQ (Half-Quadratic Quantization):** Fast quantization method optimizing a custom loss function, configured for 4-bit weights with optimized zero-point placement. This method quantizes the model on-the-fly during loading.

  Parameters:

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

  Implementation:

```python
from hqq.models.hf.base import AutoHQQHFModel
from hqq.core.quantize import BaseQuantizeConfig

quant_config = BaseQuantizeConfig(
    nbits=4,
    group_size=64,
    axis=1
)

model = AutoHQQHFModel.from_pretrained(
    "mistralai/Mistral-7B-Instruct-v0.1",
    torch_dtype=torch.float16
)

model.quantize_model(
    quant_config=quant_config,
    device='cuda'
)
```

Listing 4: HQQ Quantization (On-the-fly)

All quantization methods target 4-bit precision to achieve similar compression ratios (approximately 3.6x) for fair comparison. NF4 and HQQ quantize on-the-fly during model loading, while GPTQ and AWQ use pre-quantized checkpoints from TheBloke's repository for efficiency.

1.4 Fine-tuning Strategy

1.4.1 Overview

Following compression, we apply targeted fine-tuning to recover performance degradation and adapt models to downstream tasks. Our fine-tuning strategy focuses on six key capability domains, selected to comprehensively evaluate the impact of quantization across diverse reasoning patterns and knowledge types.

1.4.2 Fine-tuning Domains and Datasets

We organize fine-tuning and evaluation around six capability categories, each testing different aspects of model knowledge and reasoning under compression:

Mathematical Reasoning **Domains:** Mathematics, physics, formal logic
What It Tests: Symbolic manipulation and logical reasoning under quantization. This category evaluates whether compressed models retain the ability to perform multi-step arithmetic, algebraic manipulation, and formal reasoning chains.
Datasets:

- **GSM8K:** Grade-school math word problems requiring multi-step reasoning (8-shot evaluation, exact match metric)
- **MATH:** Competition-level mathematics across algebra, geometry, number theory, and calculus (4-shot evaluation with majority voting)

Code Generation **Domains:** Python programming, algorithms, debugging
What It Tests: Syntax precision combined with algorithmic reasoning. Quantization can particularly impact code generation due to the need for exact token sequences and structured output.
Datasets:

- **HumanEval:** 164 hand-written Python programming problems with unit test evaluation (0-shot, pass@1 metric)
- **MBPP (Mostly Basic Python Problems):** 974 entry-level Python tasks with train/test splits (3-shot, pass@1 metric)
- **CodeAlpaca:** 20,000 instruction-tuning examples for code generation tasks, used for fine-tuning

World Knowledge **Domains:** Science, history, geography, general knowledge
What It Tests: Factual recall and conceptual understanding. This category assesses whether quantization affects the model's ability to retain and retrieve stored knowledge about the world.
Datasets:

- **MMLU (Massive Multitask Language Understanding):** 57 academic subjects spanning STEM, humanities, and social sciences (5-shot, accuracy metric)
- **TriviaQA:** Large-scale reading comprehension dataset with 95,000 question-answer pairs (5-shot, exact match metric)

Domain Expertise **Domains:** Medicine, law, finance, specialized technical fields
What It Tests: Specialized terminology and domain-specific reasoning patterns. Professional domains require both technical vocabulary preservation and complex reasoning chains.
Datasets:

- **MedQA:** Medical exam questions in USMLE style, testing clinical reasoning and medical knowledge
- **LegalBench:** Suite of legal reasoning tasks including contract interpretation, precedent analysis, and statutory reasoning
- **ArXiv Custom:** Custom-built evaluation sets from scientific papers in the target domain

Language Understanding and Summarization **Domains:** Reading comprehension, text summarization
What It Tests: Coherence and compression ability. This evaluates whether compressed models can maintain fluency and capture key information when generating or condensing text.
Datasets:

- **CNN/DailyMail:** News article summarization benchmark with 300,000+ article-summary pairs
- **BoolQ:** Yes/no question answering requiring reading comprehension (0-shot, accuracy metric)
- **QuAC:** Conversational question answering with context (0-shot, F1 metric)

Instruction Following **Domains:** Format control, role-play, task specification adherence
What It Tests: Model steerability post-quantization. Instruction following is critical for real-world deployment, where models must precisely follow user directives and formatting requirements.
Datasets:

- **Alpaca:** 52,000 instruction-following examples covering diverse task types and formats
- **BBH (Big Bench Hard):** 23 challenging tasks from BigBench (3-shot, accuracy metric)
- **AGI Eval:** Human-level exam questions testing general intelligence (3-5 shot, accuracy metric)

1.4.3 Fine-tuning Methodology

Training Configuration: We employ parameter-efficient fine-tuning techniques to avoid catastrophic forgetting and reduce computational requirements:

Table 1: Fine-tuning hyperparameters

Parameter	Value	Description
Method	LoRA	Low-Rank Adaptation
LoRA rank	8	Rank of adaptation matrices
LoRA alpha	16	Scaling factor
Target modules	q_proj, v_proj	Attention projection layers
Learning rate	3e-4	Peak learning rate
Batch size	8	Per-device batch size
Gradient accumulation	4	Effective batch size: 32
Epochs	3	Training epochs per domain
Warmup ratio	0.1	Learning rate warmup
LR scheduler	cosine	Cosine annealing schedule
Weight decay	0.01	L2 regularization
Max sequence length	2048	Context window

Domain-Specific Fine-tuning: For each capability domain, we fine-tune the compressed model on the corresponding training datasets and evaluate on held-out test sets. This allows us to measure:

- **Performance Recovery:** How much of the quantization-induced degradation is recovered through fine-tuning
- **Domain Adaptability:** Whether compressed models can still effectively adapt to specialized tasks
- **Training Stability:** If quantization affects gradient flow and optimization dynamics

Evaluation Protocol:

1. **Baseline:** Evaluate compressed model before fine-tuning
2. **Fine-tune:** Train on domain-specific datasets for 3 epochs
3. **Evaluate:** Test on held-out evaluation sets using standard benchmarks
4. **Compare:** Measure performance delta vs. FP16 baseline and pre-fine-tuning compressed model

1.4.4 Rationale for Domain Selection

The six capability domains are selected to provide comprehensive coverage of model capabilities:

- **Mathematical Reasoning** tests formal symbolic manipulation, which is highly sensitive to quantization due to the precision required in arithmetic operations
- **Code Generation** evaluates structured output generation and syntax precision, where small errors can break functionality
- **World Knowledge** assesses factual retention in model weights, directly testing whether quantization causes knowledge loss
- **Domain Expertise** examines specialized knowledge preservation in professional contexts with technical vocabulary

- **Language Understanding** measures fluency and coherence, which can degrade with aggressive compression
- **Instruction Following** evaluates controllability and steerability, critical for practical deployment

Together, these domains span the spectrum from precise symbolic reasoning (math, code) to open-ended generation (language, summarization), from general knowledge (world facts) to specialized expertise (medical, legal), and from zero-shot evaluation (code, comprehension) to few-shot adaptation (MMLU, GSM8K).

1.5 RAG Pipeline Configuration

1.5.1 Pipeline Architecture

Our RAG (Retrieval-Augmented Generation) pipeline implements a modular architecture consisting of five core components that work in sequence to enable context-aware question answering:

1. **Document Processing:** Extract and clean text from source documents (PDF, TXT, Markdown)
2. **Text Chunking:** Split documents into semantically coherent segments
3. **Embedding:** Convert text chunks into dense vector representations
4. **Vector Indexing:** Store and index embeddings for efficient similarity search
5. **Retrieval & Generation:** Query the index and generate contextualized answers

The pipeline is implemented as a reusable framework that can be applied to any of our compressed models, enabling direct comparison of RAG performance across quantization methods.

1.5.2 Document Processing

Text Extraction:

- **PDF Processing:** PyPDF2-based extraction with page-level granularity
- **Text Normalization:** Whitespace normalization, OCR error correction, quote standardization
- **Cleaning Operations:**
  - Remove page numbers and headers
  - Strip citation references ([1], (Author, 2020))
  - Remove URLs and excessive whitespace
  - Fix common ligature errors (fi, fl)

Table 2: Document processing parameters

Parameter	Default Value	Description
remove_headers	true	Strip page headers and numbers
remove_citations	true	Remove citation markers
extract_sections	false	Parse document sections

1.5.3 Text Chunking Strategy

We employ a **semantic chunking** strategy that respects natural document boundaries while maintaining optimal chunk sizes for embedding and retrieval. This approach outperforms fixed-size chunking by preserving contextual coherence.

Chunking Algorithm:

- **Strategy:** Semantic (paragraph-aware)
- **Primary Delimiter:** Double newlines (paragraph boundaries)
- **Fallback:** Sentence-level tokenization using NLTK punkt tokenizer
- **Overlap Mechanism:** Sliding window with configurable overlap to maintain context continuity across chunk boundaries

Table 3: Text chunking parameters

Parameter	Default Value	Description
strategy	semantic	Chunking strategy (semantic, sentence, fixed)
chunk_size	512	Target chunk size in tokens
chunk_overlap	50	Overlap between consecutive chunks
min_chunk_size	100	Minimum viable chunk size

Chunk Metadata: Each chunk includes metadata for traceability and filtering:

- **chunk_id:** Unique identifier (chunk_0, chunk_1, ...)
- **page_number:** Source page in original document
- **start_char / end_char:** Character offsets in source
- **tokens:** Word count for the chunk
- **section:** Optional section header (if extracted)

1.5.4 Embedding Model

We use **sentence-transformers/all-MiniLM-L6-v2** as our embedding model, balancing quality and efficiency for retrieval tasks.

Model Characteristics:

- **Architecture:** Distilled from Microsoft MiniLM
- **Embedding Dimension:** 384
- **Max Sequence Length:** 256 tokens
- **Training:** Fine-tuned on 1B+ sentence pairs
- **Performance:** SBERT benchmark score: 68.06 (semantic similarity)

Table 4: Embedding model parameters

Parameter	Default Value	Description
model_name	all-MiniLM-L6-v2	SentenceTransformer model identifier
batch_size	32	Batch size for embedding generation
normalize	true	L2 normalization of embeddings
device	cuda	Compute device (cuda, mps, cpu)

Implementation:

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer(
    'sentence-transformers/all-MiniLM-L6-v2',
    device='cuda'
)

embeddings = model.encode(
    texts,
    batch_size=32,
    show_progress_bar=True,
    normalize_embeddings=True,
    convert_to_numpy=True
)
```

Listing 5: Embedding Generation

1.5.5 Vector Store and Indexing

We use **ChromaDB** as our vector database, providing efficient similarity search with multiple distance metrics.

Vector Store Configuration:

- **Database:** ChromaDB (persistent or in-memory)
- **Index Type:** HNSW (Hierarchical Navigable Small World)
- **Distance Metric:** Cosine similarity (default)
- **Storage:** Optional persistent storage for index reuse

Table 5: Vector store parameters

Parameter	Default Value	Description
collection_name	rag_documents	Index collection identifier
persist_directory	null	Directory for persistent storage (null=in-memory)
distance_metric	cosine	Distance function (cosine, l2, ip)

Distance-to-Similarity Conversion: ChromaDB returns distances that must be converted to similarity scores [0, 1]:

- **Cosine:** similarity $= 1 - \frac{d^2}{2}$ where d is L2 distance of normalized vectors
- **L2:** similarity $= \frac{1}{1+d}$ (exponential decay)
- **Inner Product:** similarity $= \frac{d+2}{2}$ (normalized to [0, 1])

1.5.6 Retrieval Strategy

Our retrieval system implements advanced techniques beyond simple nearest-neighbor search:
Base Retrieval:

- **Top-K Selection:** Retrieve top-3 most similar chunks by default
- **Similarity Threshold:** Configurable minimum similarity score (default: 0.0)
- **Metadata Filtering:** Optional filtering by page number, section, etc.

Re-ranking (Optional): Hybrid retrieval combining semantic and lexical matching:

- **Semantic Score:** Original embedding similarity (70% weight)
- **Lexical Score:** Token overlap between query and chunk (30% weight)
- **Formula:** rerank_score $= 0.7 \times$ cosine_sim $+ 0.3 \times$ token_overlap

Diversity Mechanism (Optional): Maximal Marginal Relevance (MMR) to reduce redundancy:

- **Objective:** Balance relevance and diversity in retrieved chunks
- **Formula:** MMR $= \lambda \times \text{Sim}(q, c) - (1 - \lambda) \times \max[\text{Sim}(c, S)]$
- where q is query, c is candidate chunk, S is selected chunks, λ is diversity parameter

Table 6: Retrieval parameters

Parameter	Default Value	Description
top_k	3	Number of chunks to retrieve
similarity_threshold	0.0	Minimum similarity score
rerank	false	Enable hybrid re-ranking
diversity_penalty	0.0	MMR diversity parameter [0, 1]

1.5.7 Answer Generation

The generation component uses the compressed LLM with retrieved context to produce grounded answers.
Prompt Engineering: We design prompts to encourage faithful, concise answers based on retrieved context:

```
prompt = f"""Use the following context to answer the question.
Provide a clear, direct answer based on the information given.

Context:
{retrieved_context}

Question: {user_query}

Answer:"""
```

Listing 6: RAG Generation Prompt Template

Generation Parameters: Carefully tuned to balance faithfulness and naturalness:

- **Max New Tokens:** 128 (concise answers)
- **Temperature:** 0.3 (low for factual accuracy)
- **Top-p:** 0.9 (nucleus sampling)
- **Repetition Penalty:** 1.15 (prevent loops)
- **Sampling:** Enabled (allows natural phrasing)

Table 7: Answer generation parameters

Parameter	Default Value	Description
max_new_tokens	128	Maximum answer length
temperature	0.3	Sampling temperature
top_p	0.9	Nucleus sampling threshold
do_sample	true	Enable sampling vs greedy
repetition_penalty	1.15	Penalty for repeated tokens
use_chat_template	true	Use model's chat template if available

Answer Validation: Post-processing to ensure quality:

- **Truncation:** Limit to 4 sentences maximum
- **Context Truncation:** Cap context at 2000 characters to prevent overwhelming
- **Retry Mechanism:** If answer appears problematic (too short, repetitive, verbatim copying), retry with simplified prompt and lower temperature (0.2)
- **Fallback:** Return "The information is not provided in the given context" for empty/invalid responses

1.5.8 RAG Evaluation Dataset

We create a custom technical QA dataset from documentation corpora to evaluate RAG performance:
Dataset Characteristics:

- **Domain:** Technical documentation (ML frameworks, APIs)
- **Size:** 10-50 question-answer pairs per evaluation
- **Question Types:**
  - Factual: "What is the default learning rate?"
  - Procedural: "How do you initialize a model?"
  - Comparative: "What's the difference between X and Y?"
- **Answer Format:** Short-form answers (1-3 sentences)
- **Ground Truth:** Human-verified reference answers

Table 8: RAG evaluation dataset parameters

Parameter	Default Value	Description
num_questions	10	Number of QA pairs to evaluate
dataset_path	null	Path to custom QA JSON file
compare_no_rag	true	Evaluate without retrieval baseline
save_detailed_responses	false	Save individual responses to file

Evaluation Protocol:

1. Index source documents using the RAG pipeline
2. For each test question:
   - Retrieve top-K relevant chunks
   - Generate answer with context (RAG)
   - Generate answer without context (no-RAG baseline)
3. Compute metrics comparing predictions to reference answers
4. Aggregate results across all questions

1.5.9 Pipeline Integration with Compressed Models

The RAG pipeline is model-agnostic and interfaces with any compressed model through a unified ModelInterface:

```
from rag import RAGPipeline

# Load compressed model
model_interface = load_compressed_model("NF4")

# Initialize RAG pipeline
rag_config = {
    "chunking": {"strategy": "semantic", "chunk_size": 512},
    "embedding": {"model_name": "all-MiniLM-L6-v2"},
```

```
    "retrieval": {"top_k": 3, "rerank": False},
    "generation": {"temperature": 0.3, "max_new_tokens": 128}
}

pipeline = RAGPipeline(rag_config)
pipeline.setup(model_interface)

# Index documents
pipeline.index_documents("technical_docs.pdf")

# Evaluate
results = pipeline.evaluate(test_questions, compare_no_rag=True)
```

Listing 7: RAG Pipeline Initialization

This design enables fair comparison of RAG performance across all compression methods, as all components except the generation model remain constant.

1.6 Evaluation Metrics and Benchmarks

Our evaluation framework assesses compressed models across three dimensions: computational efficiency, task performance, and retrieval-augmented generation capabilities.

1.6.1 Efficiency Metrics

We measure computational efficiency through the following metrics:
Latency Measurements:

- **Average Latency:** Mean time per generated token (ms) measured across multiple inference runs with warmup iterations to ensure stable GPU states
- **Time to First Token (TTFT):** Initial response latency measuring the time from prompt submission to first token generation, critical for interactive applications
- **Prefill vs. Decode Latency:** Separate measurement of prompt processing time (prefill) and autoregressive generation time (decode) to identify optimization bottlenecks

Table 9: Latency measurement parameters

Parameter	Default Value	Description
num_warmup	3	Warmup iterations before measurement
num_runs	10	Number of measurement iterations
max_new_tokens	128	Maximum tokens to generate per prompt
prompts	8 prompts	List of test prompts for benchmarking

Throughput and Memory:

- **Throughput:** Sustained generation rate measured in tokens per second, averaged over extended generation sequences
- **Peak Memory:** Maximum GPU memory allocated during inference (MB), captured using CUDA memory profiling
- **Model Size:** Disk storage requirements (GB) including all parameters and buffers
- **Memory Efficiency:** Ratio of model size to peak memory usage, indicating memory overhead beyond model parameters (e.g., activations, KV cache)

Table 10: Throughput and memory measurement parameters

Parameter	Default Value	Description
num_runs	10	Number of measurement iterations
max_new_tokens	128	Tokens to generate per run
batch_size	1	Batch size for evaluation
measure_batch_throughput	false	Test multiple batch sizes
batch_sizes	[1, 2, 4, 8]	Batch sizes to test (if enabled)

Computational Efficiency:

- **Model FLOPs Utilization (MFU):** Percentage of theoretical peak hardware FLOPs achieved during inference, calculated as $\text{MFU} = \frac{\text{Achieved FLOPs/s}}{\text{Peak Hardware FLOPs/s}} \times 100\%$, where achieved FLOPs is the product of FLOPs per token and throughput
- **Energy Consumption:** Estimated energy per token (mJ) based on device Thermal Design Power (TDP) and measured latency, using the formula $E = (P_{\text{TDP}} - P_{\text{idle}}) \times t$, where P_{idle} is assumed to be 30% of TDP

Table 11: Computational efficiency parameters

Parameter	Default Value	Description
device_tdp_watts	Auto-detected	Device thermal design power
idle_power_ratio	0.3	Fraction of TDP at idle (30%)
peak_tflops	Auto-detected	Hardware peak TFLOPs (FP16)

1.6.2 Performance Benchmarks

We evaluate model quality using established language modeling benchmarks, organized by capability:
Language Modeling:

- **Perplexity:** Measured on WikiText-2 test set using sliding window evaluation with stride 512 to assess next-token prediction quality. Lower perplexity indicates better language understanding.

Table 12: Perplexity evaluation parameters

Parameter	Default Value	Description
dataset	wikitext	HuggingFace dataset name
dataset_config	wikitext-2-raw-v1	Dataset configuration
split	test	Dataset split to evaluate
num_samples	100	Number of samples to process
max_length	512	Maximum sequence length
stride	512	Sliding window stride (null=no sliding)
batch_size	1	Batch size for processing

Selected Core Tasks:
All core tasks use the Language Model Evaluation Harness [**?**] with consistent hyperparameters for reproducibility. We report accuracy (or pass@1 for code tasks) normalized to [0,1].

Table 13: Core task benchmark parameters

Task	Few-Shot	Metric	Description
HellaSwag	0	acc_norm	Commonsense reasoning via sentence completion in everyday scenarios
ARC-Easy	0	acc_norm	Grade-school level science question answering
ARC-Challenge	0	acc_norm	Challenge-level scientific reasoning questions
GSM8K	8	exact_match	Grade-school math word problems with step-by-step reasoning
MMLU	5	acc	Multi-domain knowledge across 57 academic subjects
HumanEval	0	pass@1	Python code generation evaluated on test case pass rate

Table 14: LM-Eval harness global parameters

Parameter	Default Value	Description
batch_size	1	Global batch size for all tasks
limit	null	Limit samples per task (null=all)
random_seed	1234	Random seed for reproducibility

Original Mistral-7B Benchmarks:

Table 15: Mistral-7B complete benchmark suite

Category	Tasks	Few-Shot	Metric
Commonsense Reasoning	HellaSwag	0	acc_norm
	Winogrande	0	acc
	PIQA	0	acc_norm
	SIQA	0	acc
	OpenbookQA	0	acc_norm
	ARC-Easy	0	acc_norm
	ARC-Challenge	0	acc_norm
	CommonsenseQA	0	acc
World Knowledge	NaturalQuestions	5	exact_match
	TriviaQA	5	exact_match
Reading Comprehension	BoolQ	0	acc
	QuAC	0	f1
Math	GSM8K	8 (maj@8)	exact_match
	MATH	4 (maj@4)	exact_match
Code	HumanEval	0	pass@1
	MBPP	3	pass@1
Aggregate Benchmarks	MMLU	5 (57 tasks)	acc
	BBH	3 (23 tasks)	acc
	AGI Eval	3-5	acc

1.6.3 RAG Evaluation

For retrieval-augmented generation, we evaluate both retrieval quality and answer generation using a custom question-answering dataset.

Retrieval Quality Metrics:

Table 16: Retrieval quality metrics and parameters

Metric	Description
Context Sufficiency	Fraction of queries where retrieved contexts contain sufficient information (80% token overlap threshold)
Context Precision	Relevance of retrieved chunks measured by query-context token overlap
Context Coverage	Fraction of answer tokens present in retrieved contexts
Retrieval Consistency	Standard deviation of retrieval scores, indicating stability
Precision@K	Fraction of top-K retrieved items that are relevant
Recall@K	Fraction of relevant items in top-K retrieved
F1@K	Harmonic mean of Precision@K and Recall@K
MRR	Mean reciprocal rank of first relevant item
MAP	Mean average precision across all queries

Table 17: Retrieval evaluation parameters

Parameter	Default Value	Description
top_k	3	Number of chunks to retrieve
k_values	[1, 3, 5, 10]	K values for precision@k, recall@k
similarity_threshold	0.3	Minimum similarity score threshold
relevance_token_threshold	0.3	Token overlap threshold for relevance
sufficiency_token_threshold	0.8	Token overlap threshold for sufficiency

Answer Generation Metrics:

Table 18: Answer generation metrics and parameters

Metric	Description
Exact Match (EM)	Binary correctness: perfect normalized string match
F1 Score	Token-level precision-recall harmonic mean
Answer Relevance	Query-answer token overlap (measures if answer addresses question)
Faithfulness	Token containment: fraction of answer tokens in retrieved context
ROUGE-1/2/L	N-gram overlap (unigram, bigram) and longest common subsequence
BERTScore	Semantic similarity using contextual BERT embeddings (F1)
BLEU	Translation-style matching with smoothing

Table 19: Answer generation parameters

Parameter	Default Value	Description
max_new_tokens	128	Maximum tokens in generated answer
temperature	0.3	Sampling temperature (lower=deterministic)
top_p	0.9	Nucleus sampling threshold
repetition_penalty	1.15	Penalty for repeated tokens
normalize_whitespace	true	Normalize whitespace in comparisons
case_sensitive	false	Case-sensitive matching
remove_punctuation	false	Remove punctuation before comparison
rouge_use_stemmer	true	Use Porter stemmer for ROUGE
bertscore_lang	en	Language for BERTScore

RAG Efficiency:

Table 20: RAG efficiency metrics

Metric	Description
Retrieval Time	Average time to retrieve top-k contexts (ms)
RAG Generation Time	Time to generate answers with retrieved context (ms)
No-RAG Generation Time	Baseline generation time without context (ms)
RAG Throughput	Generation speed with context (tokens/sec)
No-RAG Throughput	Generation speed without context (tokens/sec)
Generation Speedup	Ratio of no-RAG to RAG generation time
F1 Improvement	Delta between RAG and no-RAG F1 scores
EM Improvement	Delta between RAG and no-RAG exact match scores

Table 21: RAG evaluation dataset parameters

Parameter	Default Value	Description
num_questions	10	Number of QA pairs to evaluate
dataset_path	null	Path to custom QA dataset (JSON)
compare_no_rag	true	Compare with no-RAG baseline
save_detailed_responses	false	Save individual Q&A responses

All RAG metrics are averaged over the evaluation dataset sampled from a technical documentation corpus, with retrieval configured to return top-3 chunks per query by default.

2 Results

2.1 Overview

We present a comprehensive comparison of compression techniques across three dimensions: efficiency gains, performance preservation, and RAG capabilities. Our analysis focuses on understanding the trade-offs between model size reduction, inference speed, and task performance for each compression method.

2.2 Compression Efficiency Analysis

2.2.1 Quantization Methods

Table 22: Performance comparison of quantization methods 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	13.5	0.69	0.72	0.5	—	—	—

Table 23: Average accuracy and performance degradation for quantization methods

Method	**Average Accuracy**	**Perplexity Increase**	**Accuracy Drop**	**Tasks Evaluated**
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

2.3 Performance Preservation Analysis

2.3.1 Quantization Methods

Table 24: RAG answer quality evaluation of quantization methods

Method	F1	EM	Faithfulness	Relevance	ROUGE-1	ROUGE-2	ROUGE-L	BERTScore F1
FP16	0.217	0.0	0.559	0.094	0.279	0.092	0.197	0.658
NF4	0.205	0.0	0.539	0.083	0.244	0.086	0.177	0.614
GPTQ	—	—	—	—	—	—	—	—
AWQ	0.191	0.0	0.476	0.082	0.243	0.088	0.181	0.615
HQQ	—	—	—	—	—	—	—	—

Table 25: RAG vs no-RAG comparison for quantization methods

Method	No-RAG F1	No-RAG EM	F1 Improvement	EM Improvement	Avg Answer Length (RAG)	Avg Answer Length (No-RAG)
FP16	0.190	0.0	+0.027	0.0	33.75	—
NF4	0.181	0.0	+0.024	0.0	31.95	—
GPTQ	—	—	—	—	—	—
AWQ	0.165	0.0	+0.025	0.0	30.5	—
HQQ	—	—	—	—	—	—

2.4 RAG Performance Analysis

2.4.1 Quantization Methods

Table 26: RAG answer quality evaluation of quantization methods

Method	F1	EM	Faithfulness	Relevance	ROUGE-1	ROUGE-2	ROUGE-L	BERTScore F1
FP16	0.217	0.0	0.559	0.094	0.279	0.092	0.197	0.658
NF4	0.205	0.0	0.539	0.083	0.244	0.086	0.177	0.614
GPTQ	—	—	—	—	—	—	—	—
AWQ	0.191	0.0	0.476	0.082	0.243	0.088	0.181	0.615
HQQ	—	—	—	—	—	—	—	—

Table 27: RAG vs no-RAG comparison for quantization methods

Method	No-RAG F1	No-RAG EM	F1 Improvement	EM Improvement	Avg Answer Length (RAG)	Avg Answer Length (No-RAG)
FP16	0.190	0.0	+0.027	0.0	33.75	—
NF4	0.181	0.0	+0.024	0.0	31.95	—
GPTQ	—	—	—	—	—	—
AWQ	0.165	0.0	+0.025	0.0	30.5	—
HQQ	—	—	—	—	—	—

Table 28: Context retrieval quality across quantization methods

Method	Sufficiency	Precision	Coverage	Consistency (std)	Avg Score	Avg Chunks Retrieved	Avg Context Length
FP16	0.796	0.564	0.756	0.090	0.800	3.0	1308.5
NF4	0.796	0.564	0.756	0.090	0.800	3.0	1308.5
GPTQ	—	—	—	—	—	—	—
AWQ	0.796	0.564	0.756	0.090	0.800	3.0	1308.5
HQQ	—	—	—	—	—	—	—

Table 29: RAG efficiency metrics for quantization methods

Method	Retrieval Time (ms)	RAG Gen Time (ms)	No-RAG Gen Time (ms)	RAG Throughput (tok/s)	No-RAG Throughput (tok/s)	Generation Speedup
FP16	24.7	8435.5	7855.2	5.33	9.34	0.93x
NF4	28.0	10443.9	9965.0	4.37	7.06	0.95x
GPTQ	—	—	—	—	—	—
AWQ	26.1	9993.9	7744.2	4.36	7.94	0.77x
HQQ	—	—	—	—	—	—