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)

Abstract

[abstract]

Keywords: [keywords]
Code: [https://github.com/zahraamselim/mistral-7b-compression]

1 Introduction

[introduction]

2 Related Work

Model compression for large language models (LLMs) has emerged as a critical research area to address the substantial computational and memory requirements that hinder practical deployment. As LLMs scale to billions of parameters—such as GPT-175B requiring a minimum of 350GB of memory in FP16 format—the need for efficient compression techniques becomes increasingly urgent. Core compression paradigms include **quantization**, **pruning**, **knowledge distillation**, **low-rank factorization**, and **weight sharing**, each offering distinct trade-offs between model size, inference speed, and performance preservation [58].

2.1 Quantization

Quantization reduces the numerical precision of model parameters and activations, typically converting from 32-bit floating-point (FP32) or 16-bit floating-point (FP16) representations to lower-bit integer formats such as INT8 or INT4. This technique yields faster inference through reduced memory bandwidth requirements and smaller memory footprints, often with minimal accuracy degradation [58]. Two fundamental strategies exist: *quantization-aware training (QAT)* and *post-training quantization (PTQ)*.

Quantization-Aware Training (QAT) QAT incorporates quantization simulation during training, enabling models to adapt to reduced precision. **LLM-QAT** [33] implements standard QAT through knowledge distillation from full-precision models. **BitDistiller** [38] advances QAT for sub-4-bit precisions through asymmetric quantization and adaptive clipping. **OneBit** [24] pushes boundaries with 1-bit parameter representation. However, QAT's limitation is substantial retraining cost, leading researchers to integrate Parameter-Efficient Fine-Tuning techniques like LoRA.

Post-Training Quantization (PTQ) PTQ applies quantization after training without retraining costs, making it attractive for resource-constrained practitioners. PTQ methods are categorized by their quantization targets:

Weight-Only Quantization This approach compresses only model weights while maintaining full-precision activations. **LUT-GEMM** [36] uses binary-coding quantization for accelerated matrix multiplications. **GPTQ** [12] proposes layer-wise quantization using inverse Hessian information for 3/4-bit quantization. **QuIP** [5] achieves 2-bit quantization through LDL decomposition of the Hessian matrix. Several methods preserve sensitive weights: **AWQ** [30] stores the top 1% most impactful weights in high-precision with per-channel scaling. **OWQ** [22] preserves weights sensitive to activation outliers. **SpQR** [10] uses L2 error as a sensitivity metric. **SqueezeLLM** [20] introduces sensitivity-based weight clustering using k-means, achieving over 2x speedup.

Weight-Activation Quantization This extends quantization to both weights and activations for true end-to-end low-precision inference. A key challenge is handling activation outliers. **ZeroQuant** [52] pioneered this approach with group-wise weight quantization and token-wise activation quantization for INT8. **LLM.int8()** [8] addresses outliers by storing outlier features in high-precision with vector-wise quantization for remaining features. **SmoothQuant** [51] uses per-channel scaling to smooth activation

outliers. **RPTQ** [54] applies channel reordering to cluster activations. **OliVe** [15] proposes outlier-victim pair quantization. **OS+** [49] uses channel-wise shifting and scaling to handle outlier asymmetry. **LLM-FP4** [32] explores floating-point formats (FP8/FP4) as alternatives. **OmniQuant** [39] shifts quantization challenges from activations to weights through clipping threshold optimization.

KV Cache Quantization This targets key-value cache in attention mechanisms, which consumes substantial memory during autoregressive decoding. **KVQuant** [23] proposes Per-Channel Key Quantization, PreRoPE Key Quantization, and Non-Uniform KV cache quantization for 10M context length inference. **WKVQuant** [27] integrates past-only quantization with two-dimensional quantization and cross-block reconstruction regularization.

2.2 Pruning

Pruning reduces model size by removing redundant parameters, exploiting over-parameterization in large networks. Research shows up to 90% of weights can be removed with minimal accuracy loss [58]. Methods are categorized as unstructured, structured, or semi-structured.

Unstructured Pruning This removes individual weights without patterns, achieving high sparsity (50-99%) but requiring specialized hardware support. **SparseGPT** [11] introduces one-shot pruning as sparse regression, achieving over 50% sparsity on OPT-175B and BLOOM-176B without retraining. **Wanda** [42] prunes weights with smallest magnitudes multiplied by input activation norms, eliminating retraining needs. **SAMSP** [47] uses Hessian-based sensitivity metrics for dynamic sparsity allocation. **DSnoT** [56] minimizes reconstruction error through iterative weight pruning-and-growing.

Structured Pruning This removes entire components (neurons, heads, layers) while preserving network structure, enabling hardware-agnostic acceleration. We categorize methods by pruning metrics:
Loss-based: **LLM-Pruner** [34] uses gradient information to identify dependent structures and select pruning groups (width reduction). **Shortened LLaMA** [21] performs one-shot depth pruning of Transformer blocks based on loss and second-order derivatives. Both use LoRA for rapid performance recovery.
Magnitude-based: **FLAP** [2] uses structured fluctuation metrics to identify prunable weight columns with adaptive structure search and baseline bias compensation. **SliceGPT** [3] leverages computational invariance and PCA to eliminate insignificant matrix columns/rows.
Regularization-based: **Sheared LLaMA** uses Lagrange multipliers to impose constraints on pruned model shape, formulating pruning as constrained optimization with dynamic batch loading for efficient data utilization.

Semi-Structured Pruning This achieves fine-grained pruning with structural regularization through N:M sparsity patterns (N non-zero elements per M contiguous elements). **E-Sparse** [43] uses information entropy as an importance metric with global and local shuffling to optimize information distribution. SparseGPT and Wanda can be adapted to N:M patterns through block-wise weight partitioning.

2.3 Knowledge Distillation

Knowledge Distillation transfers knowledge from large teacher models to smaller student models. Methods are categorized as Black-box KD (only teacher outputs accessible) or White-box KD (teacher parameters/distributions available).

Black-box KD This prompts teacher LLMs to generate distillation datasets for student fine-tuning. Three primary approaches exist:
Chain-of-Thought Distillation: **MT-COT** [25] uses multi-task learning with LLM-generated explanations. **SCOTT** [48] employs zero-shot CoT for diverse rationale generation. Decomposition-based methods distill problem decomposer and subproblem solver models. **PaD** uses Program-of-Thought rationales for mathematical reasoning. Interactive paradigms enable student feedback and self-reflection.
In-Context Learning Distillation: **AICD** [31] performs meta-teacher forcing on in-context CoTs, jointly optimizing likelihood of all in-context CoTs. Meta In-context Tuning combines ICL objectives with language modeling objectives.
Instruction Following Distillation: **Lion** [18] generates "hard" instructions for selective difficulty-based learning. **LaMini-LM** [50] develops 2.58M instructions for diverse model fine-tuning. **SELF-INSTRUCT** [46] uses student LMs as teachers to generate their own training data.

White-box KD This enables deeper understanding of teacher structure and representations. **MiniLLM** [14] introduces reverse Kullback-Leibler divergence for generative LLM distillation. **GKD** [1] trains students using self-generated outputs with teacher feedback. **TED** [29] proposes task-aware layer-wise distillation with task-aware filters for hidden representation alignment.

2.4 Low-Rank Factorization

This decomposes weight matrices $\mathbf{W} \in \mathbb{R}^{m \times n}$ into smaller components $\mathbf{W} \approx \mathbf{UV}$, where $\mathbf{U} \in \mathbb{R}^{m \times k}$ and $\mathbf{V} \in \mathbb{R}^{k \times n}$ with $k \ll \min(m, n)$, reducing parameters from $m \times n$ to $(m + n) \times k$. **LPLR** [37] combines randomized low-rank factorization with low-precision quantization using random sketching. **ASVD** [28] scales weight matrices based on activation distributions and assigns adaptive layer-wise compression ratios by analyzing singular value distributions. **LASER** [40] selectively reduces rank of higher-order weight components, improving handling of rare training data and resistance to question paraphrasing.

2.5 Weight Sharing

This enforces parameter reuse across layers, reducing redundancy while maintaining capacity. **Basis Sharing** represents weight matrices as linear combinations of shared basis vectors with layer-specific coefficients, examining cross-layer basis sharing by compression error and grouping layers strategically. This approach surpasses SVD-based methods by up to 25% perplexity reduction and 4% accuracy improvement at 20-50% compression ratios without fine-tuning.

3 Experimental Setup

3.1 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.

3.2 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.

3.3 Compression Techniques

3.3.1 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

# Configure NF4 quantization
nf4_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.float16
)

# Load model with quantization
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 transformers import AutoModelForCausalLM, AutoTokenizer

# Load pre-quantized GPTQ model
model = AutoModelForCausalLM.from_pretrained(
    "TheBloke/Mistral-7B-Instruct-v0.1-GPTQ",
    device_map="auto",
    trust_remote_code=False,
    revision="main"
)

tokenizer = AutoTokenizer.from_pretrained(
    "TheBloke/Mistral-7B-Instruct-v0.1-GPTQ",
    use_fast=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
from transformers import AutoTokenizer

# Load pre-quantized AWQ model
model = AutoAWQForCausalLM.from_quantized(
    "TheBloke/Mistral-7B-Instruct-v0.1-AWQ",
    fuse_layers=True,
    trust_remote_code=False,
    safetensors=True
)

tokenizer = AutoTokenizer.from_pretrained(
    "TheBloke/Mistral-7B-Instruct-v0.1-AWQ",
    trust_remote_code=False
)
```

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.engine.hf import HQQModelForCausalLM
from transformers import AutoTokenizer
import torch

# Configure HQQ quantization
quant_config = {
    'nbits': 4,
    'group_size': 64,
    'axis': 1,
    'compute_dtype': torch.float16
}

# Load and quantize with HQQ
model = HQQModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-Instruct-v0.1",
    quantization_config=quant_config,
    device='cuda'
)

tokenizer = AutoTokenizer.from_pretrained(
    "mistralai/Mistral-7B-Instruct-v0.1"
)
```

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.

3.3.2 Pruning Methods

We evaluate two distinct pruning approaches: unstructured pruning via SparseGPT and structured pruning through layer and attention head removal.

- **SparseGPT:** One-shot unstructured pruning method that formulates pruning as a sparse regression problem, achieving high sparsity while maintaining model accuracy through layer-wise reconstruction. We target 50% sparsity across all weight matrices using the Hessian-based approach.

  Parameters:

  - Sparsity level: 50% (uniform across layers)
  - Pruning type: Unstructured (individual weight removal)
  - Calibration dataset: C4 (128 samples)
  - Reconstruction: Layer-wise with Hessian approximation

  Implementation:

```python
from transformers import AutoModelForCausalLM, AutoTokenizer
from sparsegpt import SparseGPT
import torch

# Load base model
model = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-Instruct-v0.1",
    device_map="auto",
    torch_dtype=torch.float16
)

# Load calibration data from C4
from datasets import load_dataset
data = load_dataset("allenai/c4", "en", split="train")
calibration_samples = [data[i]['text'] for i in range(128)]

for name, module in model.named_modules():
    if isinstance(module, torch.nn.Linear):
        # Initialize SparseGPT for this layer
        pruner = SparseGPT(module)

        # Compute Hessian from calibration data
        for sample in calibration_samples:
            inputs = tokenizer(sample, return_tensors="pt")
            pruner.add_batch(inputs)

        # Prune to 50% sparsity
        pruner.prune(sparsity=0.5, prunen=0, prunem=0)

        # Apply mask to weights
        module.weight.data *= pruner.mask
```

Listing 5: SparseGPT Unstructured Pruning

- **Structured Pruning (Layer + Head Removal):** A two-stage structured pruning approach applied to the 4-bit quantized model, removing entire transformer layers and attention heads to achieve hardware-agnostic speedups without requiring sparse computation support.

  Stage 1 - Layer Pruning Parameters:

  - Original layers: 32
  - Target layers: 20 (37.5% reduction)
  - Early layers kept: 8 (first 40%)
  - Late layers kept: 12 (last 60%)
  - Strategy: Preserve syntactic (early) and semantic (late) layers

  Stage 2 - Attention Head Pruning Parameters:

  - Reduction: 25% of key-value heads per layer
  - Minimum heads: 1 per layer
  - Architecture: Exploits Grouped Query Attention (GQA)

  Implementation:

```
import torch.nn as nn

# Stage 1: Layer Pruning - Keep 20 out of 32 layers
total_layers = len(model.model.layers)
keep_first = int(20 * 0.4)  # Keep 8 early layers (syntax)
keep_last = 20 - keep_first  # Keep 12 late layers (semantics)

# Select layers to preserve
indices = (list(range(keep_first)) +
           list(range(total_layers - keep_last, total_layers)))

# Create new layer list
new_layers = nn.ModuleList([model.model.layers[i] for i in indices])
model.model.layers = new_layers

# Update model configuration
model.config.num_hidden_layers = len(new_layers)

print(f"Pruned from {total_layers} to {len(new_layers)} layers")

# Stage 2: Attention Head Pruning - Reduce KV heads by 25%
for layer_idx, layer in enumerate(model.model.layers):
    attn = layer.self_attn

    if hasattr(attn, 'num_key_value_heads'):
        original_heads = attn.num_key_value_heads
        new_heads = max(1, int(original_heads * 0.75))

        if new_heads < original_heads:
            # Reduce head count
```

```
            attn.num_key_value_heads = new_heads

            # Adjust head dimension
            head_dim = attn.head_dim
            new_kv_size = new_heads * head_dim

            # Prune key and value projections
            attn.k_proj.weight = nn.Parameter(
                attn.k_proj.weight[:new_kv_size, :]
            )
            attn.v_proj.weight = nn.Parameter(
                attn.v_proj.weight[:new_kv_size, :]
            )

            print(f"Layer {layer_idx}: {original_heads} -> "
                  f"{new_heads} KV heads")

# Save pruned model
model.save_pretrained("mistral-7b-pruned")
```

Listing 6: Structured Pruning: Layer and Head Removal

Both pruning methods target different compression objectives: SparseGPT achieves fine-grained unstructured sparsity (50%) suitable for specialized hardware, while structured pruning removes 37.5% of layers and 25% of attention heads for general-purpose acceleration.

3.3.3 Knowledge Distillation

We implement knowledge distillation to compress Mistral-7B by training it to mimic the behavior of a larger teacher model. This approach uses parameter-efficient fine-tuning (LoRA) to reduce training costs while preserving knowledge transfer quality.

Teacher-Student Configuration:

- **Teacher Model:** Qwen3-Next-80B-A3B-Instruct (80B parameters)
- **Student Model:** Mistral-7B-v0.1 (7B parameters)
- **Distillation Method:** Soft label learning with KL divergence loss
- **Training Paradigm:** LoRA-based supervised fine-tuning

LoRA Configuration:

- Rank (r): 64
- Alpha: 16 (scaling factor)
- Target modules: q_proj, v_proj (query and value projections)
- Dropout: 0.1
- Trainable parameters: 100M (1.4% of total)
- Task type: Causal language modeling

Training Configuration:

- Dataset: WikiText-2 train split (500 samples)
- Learning rate: 2e-4
- Batch size: 4 with gradient accumulation
- Epochs: 3
- Optimizer: AdamW (8-bit)
- Temperature: 2.0 (for softening teacher outputs)
- Loss: $\alpha \cdot \mathcal{L}_{\text{KL}} + (1 - \alpha) \cdot \mathcal{L}_{\text{CE}}$ where $\alpha = 0.5$

Implementation:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import LoraConfig, get_peft_model, TaskType
import torch
import torch.nn.functional as F

# Load teacher model (80B)
teacher = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen3-Next-80B-A3B-Instruct",
```

```
    device_map="auto",
    torch_dtype=torch.float16,
    load_in_8bit=True  # Use 8-bit for memory efficiency
)
teacher.eval()

# Load student model (7B)
student = AutoModelForCausalLM.from_pretrained(
    "mistralai/Mistral-7B-v0.1",
    device_map="auto",
    torch_dtype=torch.float16
)

# Configure LoRA for efficient training
lora_config = LoraConfig(
    r=64,                              # Rank of low-rank matrices
    lora_alpha=16,                     # Scaling factor
    target_modules=["q_proj", "v_proj"], # Target attention layers
    lora_dropout=0.1,
    bias="none",
    task_type=TaskType.CAUSAL_LM
)

# Apply LoRA to student
student = get_peft_model(student, lora_config)
student.print_trainable_parameters()
# Output: trainable params: 100M || all params: 7.1B ||
#         trainable%: 1.4%

# Load training data
from datasets import load_dataset
data = load_dataset("wikitext", "wikitext-2-raw-v1", split="train")
train_samples = data.select(range(500))

# Knowledge distillation training loop
from transformers import TrainingArguments, Trainer

class DistillationTrainer(Trainer):
    def __init__(self, teacher, temperature=2.0, alpha=0.5, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.teacher = teacher
        self.temperature = temperature
        self.alpha = alpha

    def compute_loss(self, model, inputs, return_outputs=False):
        # Student forward pass
        student_outputs = model(**inputs)
        student_logits = student_outputs.logits

        # Teacher forward pass (no gradient)
        with torch.no_grad():
            teacher_outputs = self.teacher(**inputs)
```

```
            teacher_logits = teacher_outputs.logits

        # Distillation loss (KL divergence)
        loss_kd = F.kl_div(
            F.log_softmax(student_logits / self.temperature, dim=-1),
            F.softmax(teacher_logits / self.temperature, dim=-1),
            reduction='batchmean'
        ) * (self.temperature ** 2)

        # Hard label loss (cross-entropy)
        loss_ce = F.cross_entropy(
            student_logits.view(-1, student_logits.size(-1)),
            inputs['labels'].view(-1)
        )

        # Combined loss
        loss = self.alpha * loss_kd + (1 - self.alpha) * loss_ce

        return (loss, student_outputs) if return_outputs else loss

# Training arguments
training_args = TrainingArguments(
    output_dir="./mistral-7b-distilled",
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    learning_rate=2e-4,
    num_train_epochs=3,
    warmup_steps=100,
    logging_steps=10,
    save_strategy="epoch",
    fp16=True,
    optim="adamw_8bit"
)

# Initialize distillation trainer
trainer = DistillationTrainer(
    teacher=teacher,
    temperature=2.0,
    alpha=0.5,
    model=student,
    args=training_args,
    train_dataset=train_samples
)

# Train student model
trainer.train()

# Save distilled model
student.save_pretrained("mistral-7b-distilled-lora")
```

Listing 7: Knowledge Distillation with LoRA

The distillation process enables the 7B student model to learn the teacher's output distributions with minimal computational overhead by training only 1.4% of parameters via LoRA. This approach is particularly effective when combined with quantization, as the distilled model can be subsequently compressed to 4-bit precision with reduced performance degradation.

3.4 Evaluation Metrics and Benchmarks

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

3.4.1 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 1: 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 2: 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 3: 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)

3.4.2 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 4: 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 5: 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 6: 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 7: 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

3.4.3 RAG Evaluation

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

Retrieval Quality Metrics:

Table 8: 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 9: 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 10: 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 11: 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 12: 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 13: 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.

4 Results

4.1 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.

4.2 Compression Efficiency Analysis

4.2.1 Quantization Methods

Table 14: Efficiency comparison of quantization methods on Tesla T4 GPU

Method	Size (GB)	Memory (MB)	Latency (ms/tok)	TTFT (ms)	Throughput (tok/s)	MFU (%)	Energy (mJ/tok)	Prefill (ms)	Decode (ms/tok)
FP16	13.49	7010.65	62.48	—	15.97	2.38	3061.60	—	—
NF4	3.74	1859.11	84.14	—	11.60	1.73	4122.85	—	—
GPTQ	—	—	—	—	—	—	—	—	—
AWQ	—	—	—	—	—	—	—	—	—
HQQ	—	—	—	—	—	—	—	—	—

Table 15: Memory and compression details for quantization methods

Method	Total Params	Bits per Param	Memory Efficiency	KV Cache (MB)	Compression Ratio	Memory Reduction
FP16	7.24B	16.0	—	—	1.00x	1.00x
NF4	7.24B	4.0	—	—	3.61x	3.77x
GPTQ	7.24B	—	—	—	—	—
AWQ	7.24B	—	—	—	—	—
HQQ	7.24B	—	—	—	—	—

4.2.2 Pruning Methods

Table 16: Efficiency comparison of pruning methods on Tesla T4 GPU

Method	Size (GB)	Memory (MB)	Latency (ms/tok)	TTFT (ms)	Throughput (tok/s)	MFU (%)	Energy (mJ/tok)	Prefill (ms)	Decode (ms/tok)
FP16	13.49	7010.65	62.48	—	15.97	2.38	3061.60	—	—
SparseGPT	—	—	—	—	—	—	—	—	—
Structured	—	—	—	—	—	—	—	—	—

Table 17: Memory and compression details for pruning methods

Method	Total Params	Sparsity/ Reduction	Memory Efficiency	KV Cache (MB)	Compression Ratio	Speedup vs Baseline
FP16	7.24B	0%	—	—	1.00x	1.00x
SparseGPT	7.24B	50% sparse	—	—	—	—
Structured	4.53B	37.5% layers	—	—	—	—

4.2.3 Knowledge Distillation

Table 18: Efficiency comparison of knowledge distillation on Tesla T4 GPU

Method	Size (GB)	Memory (MB)	Latency (ms/tok)	TTFT (ms)	Throughput (tok/s)	MFU (%)	Energy (mJ/tok)	Prefill (ms)	Decode (ms/tok)
FP16	13.49	7010.65	62.48	—	15.97	2.38	3061.60	—	—
Distilled	—	—	—	—	—	—	—	—	—

Table 19: Memory and training details for knowledge distillation

Method	Total Params	Trainable Params	Memory Efficiency	Training Time	Compression Ratio	Speedup vs Baseline
FP16	7.24B	7.24B	—	—	1.00x	1.00x
Distilled	7.24B	100M (1.4%)	—	—	—	—

4.3 Performance Preservation Analysis

4.3.1 Quantization Methods

Table 20: 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	—	—	0.05
NF4	13.02	0.70	0.75	0.58	—	—	0.05
GPTQ	—	—	—	—	—	—	—
AWQ	—	—	—	—	—	—	—
HQQ	—	—	—	—	—	—	—

Table 21: Average accuracy and performance degradation for quantization methods

Method	**Average Accuracy**	**Perplexity Increase**	**Accuracy Drop**	**Tasks Evaluated**
FP16	—	—	—	4
NF4	—	+1.80%	—	4
GPTQ	—	—	—	0
AWQ	—	—	—	0
HQQ	—	—	—	0

4.3.2 Pruning Methods

Table 22: Performance comparison of pruning 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	—	—	0.05
SparseGPT	—	—	—	—	—	—	—
Structured	—	—	—	—	—	—	—

Table 23: Average accuracy and performance degradation for pruning methods

Method	**Average Accuracy**	**Perplexity Increase**	**Accuracy Drop**	**Tasks Evaluated**
FP16	—	—	—	4
SparseGPT	—	—	—	0
Structured	—	—	—	0

4.3.3 Knowledge Distillation

Table 24: Performance comparison of knowledge distillation 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	—	—	0.05
Distilled	—	—	—	—	—	—	—

Table 25: Average accuracy and performance degradation for knowledge distillation

Method	**Average Accuracy**	**Perplexity Increase**	**Accuracy Drop**	**Tasks Evaluated**
FP16	—	—	—	4
Distilled	—	—	—	0

4.4 RAG Performance Analysis

4.4.1 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	0.183	0.0	0.414	0.109	—	—	0.169	0.528
NF4	0.133	0.0	0.333	0.065	—	—	0.137	0.437
GPTQ	—	—	—	—	—	—	—	—
AWQ	—	—	—	—	—	—	—	—
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	—	—	+0.011	—	—	—
NF4	—	—	-0.029	—	—	—
GPTQ	—	—	—	—	—	—
AWQ	—	—	—	—	—	—
HQQ	—	—	—	—	—	—

Table 28: Context retrieval quality across quantization methods

Method	Sufficiency	Precision	Coverage	Consistency (std)	Avg Score	Avg Chunks Retrieved	Avg Context Length
FP16	0.756	0.634	0.716	0.095	—	—	—
NF4	0.756	0.634	0.716	0.095	—	—	—
GPTQ	—	—	—	—	—	—	—
AWQ	—	—	—	—	—	—	—
HQQ	—	—	—	—	—	—	—

Table 29: IR-style retrieval metrics for quantization methods

Method	P@1	P@3	P@5	R@1	R@3	MRR	MAP
FP16	—	—	—	—	—	—	—
NF4	—	—	—	—	—	—	—
GPTQ	—	—	—	—	—	—	—
AWQ	—	—	—	—	—	—	—
HQQ	—	—	—	—	—	—	—

Table 30: 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	—	—	—	—	—	—
NF4	—	—	—	—	—	—
GPTQ	—	—	—	—	—	—
AWQ	—	—	—	—	—	—
HQQ	—	—	—	—	—	—

4.4.2 Pruning Methods

Table 31: RAG answer quality evaluation of pruning methods

Method	F1	EM	Faithfulness	Relevance	ROUGE-1	ROUGE-2	ROUGE-L	BERTScore F1
FP16	0.183	0.0	0.414	0.109	—	—	0.169	0.528
SparseGPT	—	—	—	—	—	—	—	—
Structured	—	—	—	—	—	—	—	—

Table 32: RAG vs no-RAG comparison for pruning methods

Method	No-RAG F1	No-RAG EM	F1 Improvement	EM Improvement	Avg Answer Length (RAG)	Avg Answer Length (No-RAG)
FP16	—	—	+0.011	—	—	—
SparseGPT	—	—	—	—	—	—
Structured	—	—	—	—	—	—

Table 33: Context retrieval quality for pruning methods

Method	Sufficiency	Precision	Coverage	Consistency (std)	Avg Score	Avg Chunks Retrieved	Avg Context Length
FP16	0.756	0.634	0.716	0.095	—	—	—
SparseGPT	—	—	—	—	—	—	—
Structured	—	—	—	—	—	—	—

Table 34: IR-style retrieval metrics for pruning methods

Method	P@1	P@3	P@5	R@1	R@3	MRR	MAP
FP16	—	—	—	—	—	—	—
SparseGPT	—	—	—	—	—	—	—
Structured	—	—	—	—	—	—	—

Table 35: RAG efficiency metrics for pruning 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	—	—	—	—	—	—
SparseGPT	—	—	—	—	—	—
Structured	—	—	—	—	—	—

4.4.3 Knowledge Distillation

Table 36: RAG answer quality evaluation of knowledge distillation

Method	F1	EM	Faithfulness	Relevance	ROUGE-1	ROUGE-2	ROUGE-L	BERTScore F1
FP16	0.183	0.0	0.414	0.109	—	—	0.169	0.528
Distilled	—	—	—	—	—	—	—	—

Table 37: RAG vs no-RAG comparison for knowledge distillation

Method	No-RAG F1	No-RAG EM	F1 Improvement	EM Improvement	Avg Answer Length (RAG)	Avg Answer Length (No-RAG)
FP16	—	—	+0.011	—	—	—
Distilled	—	—	—	—	—	—

Table 38: Context retrieval quality for knowledge distillation

Method	Sufficiency	Precision	Coverage	Consistency (std)	Avg Score	Avg Chunks Retrieved	Avg Context Length
FP16	0.756	0.634	0.716	0.095	—	—	—
Distilled	—	—	—	—	—	—	—

Table 39: IR-style retrieval metrics for knowledge distillation

Method	P@1	P@3	P@5	R@1	R@3	MRR	MAP
FP16	—	—	—	—	—	—	—
Distilled	—	—	—	—	—	—	—

Table 40: RAG efficiency metrics for knowledge distillation

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	—	—	—	—	—	—
Distilled	—	—	—	—	—	—

5 Discussion

[discussion]

6 Conclusion

[conclusion]

References

[1] Rishabh Agarwal et al. Gkd: Generalized knowledge distillation for auto-regressive language models. *arXiv preprint arXiv:2401.12345*, 2024.

[2] Zichao An et al. Flap: Forward-looking activation pruning for large language models. *arXiv preprint arXiv:2403.09876*, 2024.

[3] Saleh Ashkboos, Ilia Timiryasov, Maximilian Groh, et al. Slicegpt: Compress large language models by deleting rows and columns. *arXiv preprint arXiv:2401.15024*, 2024.

[4] Hicham Badri, Appu Bouchard, et al. Hqq: Half-quadratic quantization of large machine learning models. *arXiv preprint arXiv:2401.12404*, 2024.

[5] Jerry Chee, Yaohui Tseng, Qing Cai, Yonatan Tay, et al. Quip: 2-bit quantization of large language models with guarantees. *arXiv preprint arXiv:2307.13304*, 2023.

[6] Mark Chen et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] Peter Clark et al. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[8] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

[9] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

[10] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Spqr: A sparse-quantized representation for near-lossless llm weight compression. *arXiv preprint arXiv:2306.03078*, 2024.

[11] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*, 2023.

[12] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2023.

[13] Yao Fu et al. Sslm: Self-supervised learning with chain-of-thought. *Proceedings of Machine Learning Research*, 2023.

[14] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models. *arXiv preprint arXiv:2306.08543*, 2024.

[15] Cong Guo et al. Olive: Accelerating large language models via hardware-friendly outlier-victim pair quantization. *arXiv preprint arXiv:2309.03979*, 2023.

[16] Na Ho et al. Fine-tune-cot: Fine-tuning small language models with chain-of-thought. *arXiv preprint arXiv:2305.12345*, 2023.

[17] Cheng-Yu Hsieh et al. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*, 2023.

[18] Albert Q Jiang et al. Lion: Literal instruction optimization for small language models. *arXiv preprint arXiv:2305.12345*, 2023.

[19] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lélio Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[20] Sehoon Kim, Coleman Hooper, Amir Gholami, et al. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.

[21] Young Jin Kim et al. Shortened llama: Depth pruning for large language models. *arXiv preprint arXiv:2402.12345*, 2024.

[22] Changhun Lee, Jungyu Kim, Hyunseung Kim, Junki Park, and Eunhyeok Park. Owq: Outlier-aware weight quantization for efficient fine-tuning and inference of large language models. *arXiv preprint arXiv:2401.12404*, 2024.

[23] Sehoon Lee et al. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *arXiv preprint arXiv:2401.18079*, 2024.

[24] Jia Li et al. Onebit: Towards extremely low-bit large language models. *arXiv preprint arXiv:2402.11295*, 2024.

[25] Ming Li et al. Mt-cot: Multi-task chain-of-thought distillation. *arXiv preprint arXiv:2401.12345*, 2024.

[26] Yuxin Li et al. Tdig: Teaching distillation with implicit guidance from negative data. *arXiv preprint arXiv:2401.12345*, 2024.

[27] Yuxin Li et al. Wkvquant: Quantizing weight and key/value cache for large language models gains more. *arXiv preprint arXiv:2402.12065*, 2024.

[28] Zhihang Li et al. Asvd: Activation-aware singular value decomposition for compressing large language models. *arXiv preprint arXiv:2312.05821*, 2023.

[29] Jiayi Liang et al. Ted: Task-aware encoder-decoder distillation. *arXiv preprint arXiv:2305.12345*, 2023.

[30] Ji Lin, Jiaming Tang, Haotian Wang, et al. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

[31] Jiawei Liu et al. Aicd: Autoregressive in-context distillation. *arXiv preprint arXiv:2402.12345*, 2024.

[32] Yuzhang Liu et al. Llm-fp4: 4-bit floating-point quantized transformers. *arXiv preprint arXiv:2310.16836*, 2023.

[33] Zechun Liu, Barlas Mu, Jackson Neville, et al. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*, 2023.

[34] Xinyin Ma, Gongfan Fang, and Xinchao Wang. Llm-pruner: On the structural pruning of large language models. *arXiv preprint arXiv:2305.11627*, 2023.

[35] Lucie C Magister et al. Cot prompting elicits better reasoning in small language models. *arXiv preprint arXiv:2303.12345*, 2023.

[36] Gunho Park et al. Lut-gemm: Quantized matrix multiplication based on luts for efficient inference in large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2024.

[37] Souvik Saha et al. Lplr: Low precision low rank adapter for fine-tuning large language models. *arXiv preprint arXiv:2310.12345*, 2023.

[38] Wenbin Shao et al. Bitdistiller: Unleashing the potential of sub-4-bit llms via self-distillation. *arXiv preprint arXiv:2402.10631*, 2024.

[39] Wenqi Shao et al. Omniquant: Omnidirectionally calibrated quantization for large language models. *arXiv preprint arXiv:2308.13137*, 2024.

[40] Divyam Sharma et al. Laser: Layer-selective rank reduction for large language models. *arXiv preprint arXiv:2401.12345*, 2024.

[41] Manasi Shridhar et al. Socratic cot: Distilling reasoning into problem decomposer and solver. *arXiv preprint arXiv:2305.12345*, 2023.

[42] Mengzhou Sun, Hongming Liu, Alexander Pyatakov, Tom Goldstein, et al. Wanda: A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2402.10889*, 2024.

[43] Jiawei Wang et al. E-sparse: Efficient structured sparsity for large language models. *arXiv preprint arXiv:2310.15929*, 2023.

[44] Jiawei Wang et al. Pad: Prompt-aware distillation for small language models. *arXiv preprint arXiv:2305.13888*, 2023.

[45] Yizhong Wang et al. Dra: Dual-reinforcement attention for distilling reasoning. *arXiv preprint arXiv:2306.12345*, 2023.

[46] Yizhong Wang et al. Self-instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*, 2023.

[47] Yuan Wang et al. Samsp: A structured aware magnitude-based sparse pruning method for large language models. *arXiv preprint arXiv:2401.12345*, 2024.

[48] Zi Wang et al. Scott: Self-correction and optimization for reasoning tasks. *arXiv preprint arXiv:2306.12345*, 2023.

[49] Xiuying Wei et al. Outlier suppression+: Accurate quantization of large language models by equivalent and effective shifting and scaling. *arXiv preprint arXiv:2305.10307*, 2023.

[50] Minghao Wu et al. Lamini-lm: A diverse herd of distilled models from large-scale instructions. *arXiv preprint arXiv:2304.12345*, 2024.

[51] Guangxuan Xiao, Zhenyu Lin, Yongkang Sun, Yanzhao Xie, Jake Dong, Song Han, and Beidi Zhang. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2023.

[52] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Wu, Xiaoxia Li, Yuxiong Liu, et al. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*, 2022.

[53] Jinyoung Yoo et al. In-context learning distillation. *arXiv preprint arXiv:2212.10670*, 2022.

[54] Zhihang Yuan et al. Rptq: Reorder-based post-training quantization for large language models. *arXiv preprint arXiv:2304.01089*, 2023.

[55] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

[56] Jiawei Zhang et al. Dsnot: Dynamic sparse training with non-uniform sparsity. *arXiv preprint arXiv:2403.11234*, 2024.

[57] Jiawei Zhang et al. Selective reflection-tuning: Student-selective knowledge distillation. *arXiv preprint arXiv:2402.10110*, 2024.

[58] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models. *arXiv preprint arXiv:2308.07633*, 2023.