POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Application of Approximate Computing Techniques in Large Language Models

Supervisors

Prof. Alessandro SAVINO

Prof. Stefano DI CARLO

Candidate

Utku KEPIR

December 2025

Summary

Large Language Models (LLMs) have recently achieved state-of-the-art performance in a wide range of natural language processing tasks, but their rapid growth in size has introduced severe challenges in terms of computational cost, memory consumption, and energy efficiency. This makes their deployment on resource-constrained environments increasingly difficult, and has motivated research into approximation strategies that trade exactness for efficiency.

The first half of this thesis presents an extensive survey of approximate computing methods for transformer-based architectures, focusing on techniques such as quantization, pruning, low-rank approximation (LoRA), stochastic perturbations, and stochastic memory masking. Alongside the survey, a benchmarking framework was developed to evaluate these approaches in a consistent and comparable manner. The framework integrates support for multiple datasets, including Alpaca, Databricks-Dolly-15k, and AgentInstruct, and provides metrics such as BLEU score, ROUGE-L score, F1 score, SBERT similarity, inference time, output size, model size and perplexity. Experiments were conducted on two representative models, LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct, to investigate the efficiency–accuracy trade-offs of different approximation methods.

The second half of this thesis focuses on combining multiple approximation methods to further reduce computational overhead while preserving task performance. In particular, the work investigates the integration of LoRA with other methods to minimize the number of trainable parameters and improve training efficiency. This stage of the work emphasizes the importance of evaluating approximation techniques not only in isolation but also in combination, highlighting scenarios in which hybrid approaches achieve better efficiency–accuracy trade-offs than single methods.

Overall, this thesis provides a systematic exploration of approximation strategies for LLMs and their impact on both training and inference. The results demonstrate that lightweight approaches such as LoRA and quantization achieve substantial reductions in memory usage and computational load with minimal performance degradation, while more aggressive approximations require careful tuning to maintain robustness.

Acknowledgements

In this section, I want to thank all the people who have supported me during these years of study and throughout the preparation of this thesis. Without their help, encouragement, and presence, this work would not have been possible.

I would like to express my deepest gratitude to my supervisor, Prof. Alessandro SAVINO, for his guidance, support, and encouragement throughout the development of this work. His insights and advice have been invaluable in shaping this thesis.

I am sincerely thankful to my family for their unconditional love, patience, and support during these years. Their encouragement has given me the strength to face challenges and pursue my goals.

Finally, I would like to thank all the people who have been close to me during this journey, whose presence and kindness have made this period of my life truly meaningful.

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XI

1 Introduction 1
- 1.1 Objectives . . . 2
- 1.2 Contributions . . . 3

2 Related Works 5
- 2.1 Quantization . . . 5
  - 2.1.1 Affine Quantization . . . 5
  - 2.1.2 Integer-Only Inference . . . 6
  - 2.1.3 Post-Training Quantization, Quantization-Aware Training, and Zero-Shot Quantization . . . 8
  - 2.1.4 Simulated Quantization (Fake Quantization) . . . 10
  - 2.1.5 Advanced Quantization Formats for LLMs . . . 13
- 2.2 Pruning . . . 16
  - 2.2.1 Unstructured Pruning . . . 16
  - 2.2.2 Structured Pruning . . . 16
  - 2.2.3 Pruning and the Lottery Ticket Hypothesis . . . 17
- 2.3 Low-Rank Adaptation (LoRA) . . . 18
- 2.4 Stochastic Perturbations . . . 20
  - 2.4.1 Perturbation-Driven Variance . . . 20
  - 2.4.2 Uncertainty Aggregation . . . 21
  - 2.4.3 Stochastic Regularization Methods . . . 21
  - 2.4.4 Implementation in This Thesis . . . 22
- 2.5 Stochastic Memory Masking . . . 23
  - 2.5.1 Checkpointing and Sparsification . . . 23
  - 2.5.2 Stochastic Memory Masking for Attention . . . 24

2.5.3 Implementation in This Thesis 24
2.6 Evaluation Metrics and Efficiency Measures 24
2.6.1 Quality Metrics 25
2.6.2 Efficiency Measures 26

3 Methodology 29
3.1 Datasets 31
3.1.1 Alpaca 31
3.1.2 Databricks-Dolly-15k 32
3.1.3 AgentInstruct 33
3.2 Models 34
3.2.1 LLaMA-3.2-1B-Instruct 34
3.2.2 Gemma-3-1B-Instruct 35
3.3 Approximation Techniques 37
3.3.1 Quantization 37
3.3.2 Pruning 39
3.3.3 LoRA 40
3.3.4 Stochastic Perturbations 41
3.3.5 Stochastic Memory Masking 41
3.3.6 Combinations of Approximations 42

4 Experiments and Results 45
4.1 Experimental Setup 45
4.2 Evaluation Results 47
4.2.1 Inference Results for the Dolly-15k Dataset(Fine-tuned with Alpaca Dataset) 47
4.2.2 Inference Results for the Dolly-15k Dataset(Fine-tuned with Agent Dataset) 50
4.2.3 Inference Results for the AgentInstruct Dataset(Fine-tuned with Alpaca Dataset) 52

5 Conclusion 57

Bibliography 60

List of Tables

4.1 Inference results for LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct on the Dolly-15k dataset (Fine-tuned with Alpaca dataset). 48
4.2 Top-performing approximation techniques for LLaMA-3.2-1B and Gemma-3-1B on the Dolly-15k dataset (Fine-tuned with Alpaca dataset). 49
4.3 Inference results for LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct on the Dolly-15k dataset (Fine-tuned with Agent dataset). 51
4.4 Top-performing approximation techniques for LLaMA-3.2-1B and Gemma-3-1B on the Dolly-15k dataset (Fine-tuned with Agent dataset). 52
4.5 Inference results for LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct on the AgentInstruct dataset (Fine-tuned with Alpaca dataset). . . 53
4.6 Top-performing approximation techniques for LLaMA-3.2-1B and Gemma-3-1B on the AgentInstruct dataset (Fine-tuned with Alpaca dataset). 54

List of Figures

2.1 Illustration of Quantization-Aware Training procedure [4]. 8
2.2 Comparison between QAT (left, requires retraining) and PTQ (right, data-free) [4]. 10
2.3 Comparison between full-precision inference (Left), simulated quantization with float operations (Middle), and integer-only quantization with fixed-point arithmetic (Right) [4]. 12
2.4 Synapses and neurons before and after pruning [34]. 17

Acronyms

AI
artificial intelligence

BLEU
bilingual evaluation understudy

CRFM
center for research on foundation models

DPO
direct preference optimization

FFN
feed-forward network

GQA
grouped-query attention

LLM
large language model

LoRA
low-rank adaptation

NLP
natural language processing

PEFT
parameter-efficient fine-tuning

PTQ

post-training quantization

QAT

quantization-aware training

RLHF

reinforcement learning from human feedback

ROUGE-L

recall-oriented understudy for gisting evaluation

RoPE

rotary positional embeddings

RevNets

reversible residual networks

ZSQ

zero-shot quantization

Chapter 1

Introduction

Artificial Intelligence (AI) has recently achieved remarkable progress, largely enabled by the Transformer architecture [1], which replaced recurrence and convolutions with self-attention, resulting in both faster training and state-of-the-art performance across language tasks. Building upon this foundation, LLMs such as GPT-3 [2] have demonstrated unprecedented few-shot and zero-shot capabilities, highlighting the power of scale in model training. Scaling laws further reveal that performance in language modeling follows smooth power-law trends with respect to model size, dataset size, and compute [3]. While these trends suggest that ever larger models can achieve better generalization, they also imply unsustainable growth in computational cost, energy usage, and memory demand. The resulting tension between performance and efficiency forms a central challenge in the continued development and deployment of LLMs.

Approximate computing has emerged as a promising paradigm to address this challenge by deliberately introducing controlled imprecision into computation. Instead of performing all operations with exact, high-precision arithmetic, approximate methods aim to reduce the redundancy inherent in over-parameterized models and exploit the tolerance of neural networks to small perturbations. Techniques such as quantization [4], pruning [5], LoRA [6], stochastic perturbation (noise injection)[7], and stochastic memory masking [8] each target different aspects of efficiency. Quantization reduces bit-width, trading a small drop in accuracy for dramatic improvements in latency and memory footprint. Pruning eliminates redundant parameters, reducing model size and energy consumption [5]. LoRA introduces low-rank reparameterizations into pre-trained transformers, achieving task adaptation with only a fraction of trainable parameters and negligible inference overhead [6]. Stochastic perturbation and memory masking further explore non-traditional representations and reduced memory precision to unlock additional gains in efficiency [8].

The motivation of this thesis lies in bridging the gap between theoretical advances in approximation methods and their systematic application to modern large language models. Rather than retraining models from scratch, which is computationally prohibitive, this work adopts a training-time approximation framework. In this setup, techniques such as quantization (implemented through a hybrid approach where weights are quantized once during initialization and activations are quantized dynamically during inference), LoRA, pruning, stochastic perturbations, and stochastic memory masking are integrated directly into the fine-tuning process. The quantization strategy follows a BF16 computation pipeline: quantized weights and activations are dequantized before matrix multiplication, computed in BF16 precision, and then requantized so that the next layer receives quantized activations. This unified approach extends beyond classical post-training quantization PTQ or QAT, enabling a systematic study of how these approximation strategies individually and in combination affect training dynamics, generalization, and deployment-time efficiency. By leveraging prior insights on model scaling, compression, and efficient adaptation, the thesis establishes a benchmark framework for approximated LLMs, providing a clear view of the trade-offs between accuracy, energy consumption, and computational cost.

To ensure a fair comparison, the evaluation methodology distinguishes between training and inference efficiency. While training cost is influenced by optimizer states, gradient accumulation, and checkpointing strategies, inference time and generated output size directly reflect real-world deployment scenarios. Besides, this thesis emphasizes inference metrics such as BLEU, ROUGE-L, SBERT and F1 score [9, 10, 11]. This balance between efficiency and quality provides a reliable basis for assessing approximation techniques in practice.

1.1 Objectives

The remarkable progress of large LLMs has been accompanied by a rapid growth in model size, dataset requirements, and computational cost [2, 3]. While scaling laws provide strong evidence that larger models yield improved performance, they also reveal a widening gap between model capabilities and the resources required to train and deploy them. This gap makes it increasingly difficult to reproduce state-of-the-art results outside of large industrial labs, raising concerns about accessibility, fairness, and sustainability in the development of AI.

In this context, approximate computing has emerged as a promising research direction. The central objective of this thesis is to investigate how approximation techniques including quantization [4, 12], pruning [5, 13], low-rank adaptation through LoRA [6, 14, 15], stochastic perturbation[16, 17, 18], and memory-efficient mechanisms [8] affect the training dynamics and deployment efficiency of modern

large language models. These methods target different aspects of efficiency, respectively reducing arithmetic precision, eliminating redundant connections, introducing low-rank reparameterizations, or exploiting alternative noise and memory-optimized representations.

Another key objective is to design and implement a unified benchmarking framework that allows these approximation methods to be consistently applied across different transformer-based models. Current literature often evaluates approximations in isolation, using distinct datasets, baselines, and metrics, which makes comparisons difficult and conclusions fragmented. By contrast, this work provides a standardized platform where approximations can be directly compared, facilitating a clearer understanding of their strengths, weaknesses, and synergies.

Finally, the thesis aims to go beyond isolated analysis by exploring combinatorial approximation strategies. While most prior studies focus on one technique at a time, real-world deployment often benefits from layered approaches such as combining quantization with pruning, or integrating LoRA with stochastic memory masking. An important research question addressed here is whether such combinations unlock efficiency gains without excessively degrading model quality. The evaluation balances performance metrics such as perplexity, BLEU, ROUGE-L, F1, SBERT scores with efficiency measures like inference time and generated text output size, providing a holistic view of the trade-offs involved.

1.2 Contributions

Following these objectives, the contributions of this thesis advance both the methodological and empirical understanding of approximation for LLMs. The first contribution is the development of a unified and extensible benchmarking framework that integrates multiple approximation methods into a single experimental environment. The framework supports quantization, pruning, LoRA, stochastic perturbation, and stochastic memory masking. Its modular design enables fair, side-by-side comparisons of diverse techniques and allows straightforward extension to new approximation methods, architectures, and datasets. By consolidating previously fragmented practices in the literature, this framework lays the foundation for standardized evaluation of efficiency-oriented methods in large language models.

The second contribution lies in applying the framework to modern instruction-tuned transformer models, specifically LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct. This demonstrates portability across architectures and highlights model-specific sensitivities to approximation techniques. While this work focuses on decoder-only architectures, it also acknowledges that encoder–decoder models such as T5 and BART may interact differently with approximation methods like LoRA and quantization due to their two-stage structure. Concentrating on decoder-only

models ensures that the findings are directly applicable to the current generation of widely deployed LLMs.

The third contribution is a systematic analysis of inference-time efficiency versus accuracy trade-offs. By evaluating not only task performance (perplexity, BLEU, ROUGE-L, F1 , SBERT) but also inference time and generated output size, the study explicitly acknowledges efficiency as a first-class evaluation dimension. This approach aligns with the growing demand for sustainable and resource-efficient AI.

The fourth contribution is an exploration of combined approximation strategies. By investigating whether layering techniques such as quantization with pruning or LoRA with stochastic memory masking can achieve superior efficiency without unacceptable accuracy loss, the thesis contributes practical insights into whether approximations interact additively, synergistically, or adversarially in real-world deployments.

Finally, the contributions are validated across multiple instruction-tuned datasets, including Alpaca, Databricks-Dolly-15k, and AgentInstruct. This cross-dataset evaluation ensures that the conclusions are not tied to a single benchmark but instead generalize across different training regimes. Taken together, these contributions represent both a methodological advance through the creation of a unified approximation benchmark and a substantive empirical contribution through the systematic study of approximation trade-offs in modern LLMs.

Chapter 2

Related Works

The rapid expansion of LLMs has sparked diverse efforts to improve efficiency in both training and inference. Approximate computing strategies play a central role, deliberately trading small amounts of accuracy for substantial reductions in computation, memory, and energy. This chapter surveys the theoretical foundations and recent advances of the techniques explored in this thesis: quantization, pruning, low-rank adaptation, stochastic perturbations, and stochastic memory masking. Each section reviews classical approaches, extensions to transformers, and connections to our own framework.

2.1 Quantization

Quantization is one of the most effective approximation techniques for reducing the computational and memory footprint of neural networks. Instead of storing weights and activations in BF16 or FP32, quantization maps them to low-bit representations such as INT8 or INT4, which are directly supported on modern accelerators. This section reviews the theoretical foundation of quantization, following the integer-arithmetic formulation introduced by Jacob et al. [12], and describes recent advances tailored for LLMs.

2.1.1 Affine Quantization

The central idea is to approximate a real value $r \in \mathbb{R}$ by an integer q within a fixed range $[q_{\min}, q_{\max}]$ through an affine mapping:

$$r \approx S(q - Z), \tag{2.1}$$

where $S \in \mathbb{R}^+$ is the scale factor and $Z \in \mathbb{Z}$ is the zero-point. The zero-point ensures that zero in real space can be exactly represented in quantized space, which is important for avoiding bias shifts.

Given a real value r, its quantized integer representation is obtained as:

$$q = \text{clip}\left(\text{round}\left(\frac{r}{S} + Z\right), q_{\min}, q_{\max}\right), \tag{2.2}$$

where round$(\cdot)$ is nearest integer rounding and clip$(\cdot)$ ensures that q remains within the target bit range.

The dequantization step maps q back into floating-point:

$$\hat{r} = S \cdot (q - Z). \tag{2.3}$$

This $\hat{r}$ is the quantized approximation of r [12].

2.1.2 Integer-Only Inference

A basic requirement for efficient deployment of quantized neural networks is that all arithmetic during inference can be implemented using integer only operations on quantized values. For example, for the 8-bit quantization, $q \in \{0, \ldots, 255\}$ (unsigned) or $q \in \{-128, \ldots, 127\}$ (signed). Biases are typically quantized to 32-bit integers to avoid overflow. This mapping ensures that zero-padding operations in convolution or matrix multiplication can be implemented exactly, since the zero-point Z guarantees a quantized representation of $r = 0$.

Integer-Only Matrix Multiplication. Consider the multiplication of two square matrices of size $N \times N$:

$$r_3 = r_1 r_2, \tag{2.4}$$

where $r_1, r_2, r_3 \in \mathbb{R}^{N \times N}$.

Let (S_m, Z_m) denote the quantization parameters for matrix r_m, with quantized entries $q_{ij}^{(m)}$. Applying Eq. (2.3):

$$r_{ij}^{(m)} = S_m\Big(q_{ij}^{(m)} - Z_m\Big), \qquad m \in \{1,2,3\}. \tag{2.5}$$

By definition of matrix multiplication:

$$r_{ik}^{(3)} = \sum_{j=1}^{N} r_{ij}^{(1)} \cdot r_{jk}^{(2)}. \tag{2.6}$$

Substituting Eq. (2.5):

$$S_3\Big(q_{ik}^{(3)} - Z_3\Big) = \sum_{j=1}^{N} S_1\Big(q_{ij}^{(1)} - Z_1\Big) \cdot S_2\Big(q_{jk}^{(2)} - Z_2\Big). \tag{2.7}$$

Factoring out the scales:

$$S_3\Big(q_{ik}^{(3)} - Z_3\Big) = S_1 S_2 \sum_{j=1}^{N} \Big(q_{ij}^{(1)} - Z_1\Big)\Big(q_{jk}^{(2)} - Z_2\Big). \tag{2.8}$$

Rearranging, the integer-only update rule becomes:

$$q_{ik}^{(3)} = Z_3 + M \cdot \sum_{j=1}^{N} \Big(q_{ij}^{(1)} - Z_1\Big)\Big(q_{jk}^{(2)} - Z_2\Big), \tag{2.9}$$

where the multiplier M is defined as:

$$M = \frac{S_1 S_2}{S_3}. \tag{2.10}$$

Fixed-Point Multiplier Representation. The only non-integer term in Eq. (2.9) is the constant M. Jacob et al.[12] show that $M \in (0,1)$ in practice, and it can be expressed as:

$$M = 2^{-n} M_0, \quad M_0 \in [0.5, 1),\ n \in \mathbb{Z}_{\geq 0}. \tag{2.11}$$

Here:

- M_0 is stored as a fixed-point integer (e.g., 32-bit). Since $M_0 \geq 0.5$, it always has at least 30 bits of relative accuracy when represented in 32-bit fixed point.
- 2^{-n} is implemented as an efficient `bit-shift`, ensuring integer-only hardware execution.

Thus, multiplication by M is implemented as:

$$\text{int32\_accum} \cdot M \approx \Big(\text{int32\_accum} \cdot M_0^{(\text{int})}\Big) \gg n, \tag{2.12}$$

where $\gg n$ denotes bit-shift by n bits.

- The inner sum in Eq. (2.9) involves only integer additions and multiplications.
- The multiplier M is a single precomputed constant, applied with fixed-point arithmetic and shifts.
- This makes inference fully integer-only, with scales and zero-points applied as lightweight rescaling operations.
- Bias terms are usually quantized to INT32, ensuring no loss of accuracy in accumulation.

This affine quantization scheme underpins integer-only inference in TensorFlow Lite, ONNX Runtime, and PyTorch QNNPACK, and remains the basis for modern PTQ and QAT pipelines [19].

2.1.3 Post-Training Quantization, Quantization-Aware Training, and Zero-Shot Quantization

Quantization often requires model adaptation after weights are mapped to low-precision. This can be achieved either by fine-tuning the model (QAT), or directly quantizing a pretrained model without retraining (PTQ). A more recent category, ZSQ, eliminates the need for training data altogether, using either analytical approximations or synthetic data generation.

Quantization-Aware Training (QAT). Given a pretrained model, quantization introduces perturbations to parameters, potentially shifting the model away from the local optimum found during full-precision training. To address this, QAT retrains the model with quantization simulated in both the forward and backward passes. Specifically, weights are updated in floating point, but after each gradient update they are projected back into the quantized space as in Figure 2.1.

$$W \leftarrow \text{Quantize}(W - \eta \nabla L), \tag{2.13}$$

where η is the learning rate.

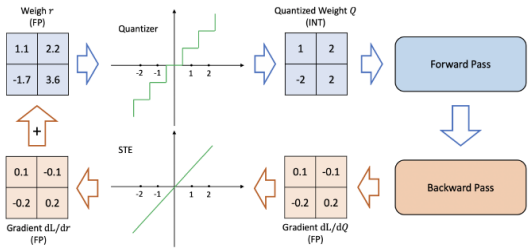

Figure 2.1: Illustration of Quantization-Aware Training procedure [4].

A central difficulty in QAT is that the quantization operator is non-differentiable. For example, rounding is piecewise constant and its derivative is zero almost everywhere. To circumvent this, QAT uses the Straight-Through Estimator (STE) [16], which approximates the gradient of quantization as the identity function:

$$\frac{\partial \,\text{round}(x)}{\partial x} \approx 1. \tag{2.14}$$

This coarse approximation nonetheless works well in practice, as it correlates with the true population gradient in expectation [4]. Figure 2.2 illustrates the difference between PTQ and QAT.

Several refinements to QAT have been proposed. PACT [20] learns activation clipping ranges as part of training, LSQ [21] learns scale factors through gradient

descent, and AdaRound [22] adapts rounding boundaries instead of using naive round-to-nearest. Despite these advances, QAT remains computationally expensive, as it often requires retraining for hundreds of epochs to recover accuracy, particularly at ultra-low precision (e.g., 2–4 bits).

Post-Training Quantization (PTQ). In PTQ, pretrained weights are quantized directly without additional retraining:

$$\hat{W} = \text{clip}\left(\text{round}\left(\frac{W}{S}\right), -q_{\max}, q_{\max}\right) \cdot S, \tag{2.15}$$

where S is the scaling factor. PTQ is computationally efficient and requires no labeled training data, making it attractive for deployment. However, it often suffers from accuracy degradation at low bit-widths, especially below INT8.

To mitigate this, several improvements have been proposed:

- **Bias correction:** Adjusting mean and variance shifts induced by quantization [23].
- **Range equalization:** Aligning weight ranges across layers to reduce scale mismatch [22].
- **Analytical Clipping (ACIQ):** Computing optimal clipping thresholds using an assumed Gaussian or Laplace distribution of activations [24].
- **Outlier Channel Splitting (OCS):** Duplicating channels with extreme values to reduce quantization error [25].
- **Adaptive rounding (AdaRound):** Learning rounding decisions to minimize quantization error [22].

These refinements enable PTQ to approach QAT-level accuracy in some cases, particularly at 8-bit and 6-bit precision, but PTQ still struggles at ultra-low bit-widths (e.g., INT4) without retraining [19].

Zero-Shot Quantization (ZSQ). In many practical scenarios, access to the original training data is impossible due to privacy, proprietary constraints, or size limitations (e.g., large web-scale datasets). ZSQ methods address this by quantizing models without access to training data.

There are two main levels:

1. **ZSQ+PTQ:** No data, no fine-tuning. Quantization parameters are chosen analytically or with heuristics (e.g., bias correction, equalization).
2. **ZSQ+QAT:** No real data, but fine-tuning is performed on synthetic data.

A popular branch of research generates synthetic calibration data. Early work used GANs to synthesize samples that approximate the model's decision boundaries. More advanced methods (e.g., ZeroQ, GDFQ) directly optimize synthetic inputs such that the internal statistics (e.g., BatchNorm means and variances) match those observed in the pretrained model:

$$\min_x D_{\mathrm{KL}}\Big(\mu_{\mathrm{BN}}(x), \mu_{\mathrm{real}}\Big) + D_{\mathrm{KL}}\Big(\sigma^2_{\mathrm{BN}}(x), \sigma^2_{\mathrm{real}}\Big). \tag{2.16}$$

These synthetic inputs are then used to calibrate quantization scales or even fine-tune the quantized model via knowledge distillation from the full-precision teacher. Recent work also integrates generative models to produce higher-fidelity calibration data [4].

Summary.

- **QAT** achieves the best accuracy, especially at 4-bit or below, but requires expensive retraining.
- **PTQ** is fast and data-free, suitable for deployment when INT8 suffices, but often suffers at ultra-low precision.
- **ZSQ** extends PTQ/QAT to data-free scenarios, relying on synthetic data or analytical approximations.

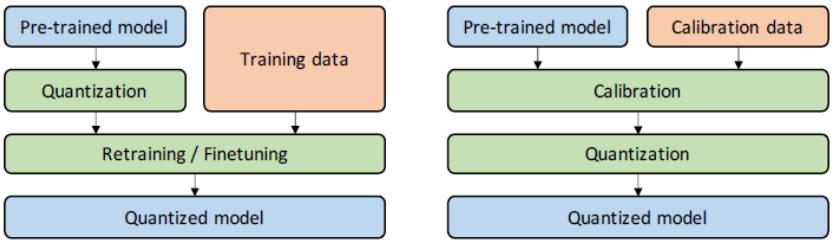

Figure 2.2: Comparison between QAT (left, requires retraining) and PTQ (right, data-free) [4].

2.1.4 Simulated Quantization (Fake Quantization)

Simulated quantization, also known as fake quantization, is a technique for emulating low-precision arithmetic within a floating-point computational framework. Unlike integer-only quantization, which performs all multiplications and accumulations using fixed-point arithmetic on quantized tensors, simulated quantization

preserves floating-point computation but introduces quantization and dequantization operations during the forward pass to replicate the precision loss of integer quantization [4, 12]. This approach is widely used in quantization-aware training (QAT) and mixed-precision fine-tuning to enable gradient-based optimization of quantized models without requiring hardware support for low-bit arithmetic.

Mathematical Formulation. Given a real-valued tensor $r \in \mathbb{R}^n$, quantization is characterized by a scale factor S and discrete quantization levels $Q = \{q_{\min}, \ldots, q_{\max}\}$. In integer quantization, the quantized tensor q is represented and stored in integer form. In contrast, simulated quantization applies both quantization and dequantization within the floating-point domain:

$$\hat{r} = \text{DeQuantize}(\text{Quantize}(r)) = S \cdot \text{clip}\left(\text{round}\left(\frac{r}{S}\right), q_{\min}, q_{\max}\right), \tag{2.17}$$

where $\hat{r}$ is the quantized approximation of r in floating-point format. This operation simulates the effect of integer quantization by injecting discretization noise while maintaining differentiability of the computational graph.

Forward and Backward Pass. During the forward pass, rounding introduces a quantization error

$$\varepsilon_q = \hat{r} - r, \tag{2.18}$$

which models the information loss due to finite precision. However, since the quantization function is non-differentiable, simulated quantization relies on the *Straight-Through Estimator* (STE) [16] to propagate gradients during backpropagation:

$$\frac{\partial \hat{r}}{\partial r} \approx 1. \tag{2.19}$$

This approximation treats the quantization function as an identity mapping in the backward pass, allowing gradient flow to proceed as if the quantization step were continuous. Although simplistic, the STE has been shown to effectively align the training dynamics of quantized models with their full-precision counterparts [4].

Computation Model. The computational structure of simulated quantization, depicted in Figure 2.3 (Middle), follows a hybrid FP32 workflow:

1. Weights W are quantized once to W_q and dequantized to FP32 (W_d) for computation.

2. Activations x are quantized dynamically at runtime to x_q and dequantized to x_d.

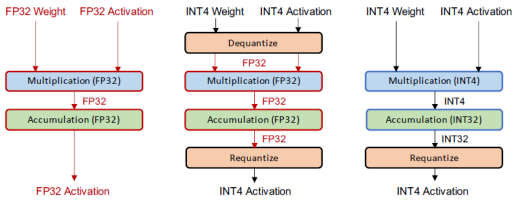

Figure 2.3: Comparison between full-precision inference (Left), simulated quantization with float operations (Middle), and integer-only quantization with fixed-point arithmetic (Right) [4].

3. Matrix multiplication and accumulation are performed in full precision:

$$y = \text{Linear}(x_d, W_d) + b. \tag{2.20}$$

4. The outputs are then re-quantized to produce low-precision activations for the next layer.

This sequence ensures that each layer experiences quantization-induced noise consistent with its low-bit equivalent, even though the arithmetic remains FP32.

Advantages and Limitations. Simulated quantization provides several theoretical and practical benefits:

- **Hardware independence:** It can be performed on any floating-point hardware, without integer matrix-multiplication units.
- **Differentiability:** Through STE, it enables gradient-based optimization of quantized networks.
- **Faithful quantization noise modeling:** It accurately emulates rounding and clipping errors seen in integer arithmetic.
- **Layer-wise flexibility:** It supports a mixture of static (for weights) and dynamic (for activations) quantization strategies.

However, simulated quantization does not yield real latency or energy savings during training or inference, since all computations still occur in floating-point. It is primarily an analytical tool for approximating quantization behavior, useful for model fine-tuning, mixed-precision training, and benchmarking [4, 19].

Comparison with Integer-Only Inference. The difference between simulated and integer-only quantization can be summarized as follows:

$$\textbf{Integer-only:} \quad y_q = \text{ReQuantize}\left(\sum_j (x_q - Z_x)(w_q - Z_w)\right), \tag{2.21}$$

$$\textbf{Simulated:} \quad y = D(x_q)^\top D(w_q), \tag{2.22}$$

where $D(\cdot)$ denotes dequantization to FP32. Thus, while integer-only inference executes integer multiplications and accumulations on hardware accelerators, simulated quantization performs identical arithmetic in expectation but remains entirely in the FP32 domain.

Implementation in this Thesis. Simulated quantization provides a theoretically grounded bridge between low-bit arithmetic and BF16 computation. It allows analysis of quantization noise, gradient behavior, and numerical stability under controlled conditions. For this reason, it is the standard paradigm in quantization-aware training frameworks [4, 12], as well as the basis of the approximation strategy employed in this thesis.

2.1.5 Advanced Quantization Formats for LLMs

Scaling large language models has motivated the development of advanced quantization schemes that go beyond standard INT8. These methods enable sub-8-bit precision, improve hardware efficiency, and reduce the accuracy gap traditionally observed at ultra-low precision.

Simulated vs. Integer-Only Quantization. There are two primary modes of deploying quantized models. In simulated quantization (also called fake quantization), weights and activations are stored in reduced precision, but computations such as matrix multiplications are carried out in floating-point. This requires dequantization before each operation and limits hardware efficiency. In contrast, integer-only quantization (fixed-point quantization) performs all operations using integer arithmetic, eliminating floating-point overheads and enabling direct use of low-precision logic on accelerators [12]. Integer-only quantization can be further enhanced with dyadic quantization, where all scaling factors are constrained to dyadic numbers (i.e., rational values with powers of two in the denominator). This makes scaling efficient via integer multiplications and bit-shifts.

Mixed-Precision Quantization. Uniformly quantizing an entire model to a fixed bit-width (e.g., INT4) can cause severe accuracy degradation. Mixed-precision

quantization alleviates this problem by assigning higher precision (e.g., INT8) to sensitive layers and lower precision (e.g., INT4) to less sensitive ones. The sensitivity of layers can be determined by second-order information such as the trace of the Hessian, as in HAWQ [26], or by reinforcement learning and differentiable NAS approaches. For LLMs, mixed-precision has been shown to deliver up to 50% inference speedup on GPUs while minimizing accuracy loss.

Hardware-Aware Quantization. Not all quantization configurations map equally well to hardware. For example, some accelerators support INT8 efficiently but lack optimized kernels for INT6 or INT2. Hardware-aware quantization incorporates latency and energy models of the target device when selecting bitwidths, ensuring that accuracy–efficiency trade-offs translate into real-world gains. This approach is increasingly important for deploying LLMs in production environments, where GPU memory bandwidth and cache hierarchies dominate performance [12].

Distillation-Assisted Quantization. Knowledge distillation can help quantized models recover accuracy lost at low precision. A high-precision teacher supervises the low-precision student by matching soft probability distributions or intermediate features. The combined loss is:

$$\mathcal{L} = \alpha\, H(y, \sigma(z_s)) + \beta\, H(\sigma(z_t, T), \sigma(z_s, T)), \tag{2.23}$$

where z_s and z_t are logits of the student and teacher, T is the distillation temperature, and H is cross-entropy [27]. Distillation has been shown to substantially improve quantization accuracy in LLMs, particularly below 8 bits.

Extreme Quantization. Binarization (1-bit) and ternarization (2-bit) represent the most aggressive forms of quantization, reducing model size by up to 32× compared to FP32. While early methods such as BinaryConnect and XNOR-Net demonstrated the feasibility of bit-wise operations, applying extreme quantization to LLMs remains challenging due to severe accuracy loss. Recent advances mitigate this degradation using scaling factors, wider architectures, and better gradient approximations [28]. Despite the challenges, extreme quantization is an active research direction for bringing transformer-based models to edge devices.

Vector Quantization. Inspired by classical signal processing, vector quantization clusters groups of weights and replaces them with centroids from a learned codebook:

$$\min_{c_1,\ldots,c_k} \sum_i \|w_i - c_j\|^2, \quad w_i \in \text{cluster } j. \tag{2.24}$$

This approach achieves up to 8× compression with limited accuracy loss [5]. Extensions such as product quantization apply clustering to submatrices and

recursively quantize residuals. For LLMs, vector quantization can be combined with pruning and Huffman coding for further compression.

Implementation in this Thesis. In this work, all quantization schemes (INT8, INT4, and NF4) are implemented as GPU-based simulated quantizers integrated into the model's fine-tuning pipeline. The implementation performs quantization and dequantization directly on the GPU to avoid CPU bottlenecks and excessive memory transfers. For uniform quantization (INT8/INT4), tensors are quantized symmetrically per tensor using the function `quantize_tensor_uniform()`, which computes the scale parameter s as:

$$s = \frac{\max |W|}{q_{\max}}, \tag{2.25}$$

where $q_{\max}$ is 127 for INT8 or 7 for INT4. Each tensor element is then mapped to its nearest quantized level using rounding and clamping operations, followed by dequantization via `dequantize_tensor()` to restore floating-point form for computation. This procedure preserves the model's numerical behavior while emulating quantization effects during training and evaluation.

For NF4 quantization, the method follows the NormalFloat-4 specification introduced by Dettmers et al. [29], but is reimplemented to operate in a fully GPU-streamed manner, and because of our GPU memory limitations. The quantization function `quantize_tensor_nf4()` divides each tensor into contiguous blocks of 64 elements, computes per-block scaling factors, and performs nearest-neighbor assignment to the fixed 16-level NF4 codebook:

$$\begin{aligned} \mathcal{C}_{\text{NF4}} = \{ &-1.0000, -0.6962, -0.5251, -0.3949, \\ &-0.2844, -0.1848, -0.0911, 0.0000, \\ &0.0911, 0.1848, 0.2844, 0.3949, \\ &0.5251, 0.6962, 0.8682, 1.0000\}. \end{aligned} \tag{2.26}$$

Quantization is computed per element by iterating over the 16 codebook entries, which avoids forming large 4D broadcast tensors that could lead to GPU memory exhaustion. The corresponding `dequantize_tensor_nf4()` function restores BF16 weights by mapping each quantized index back to its codebook value and rescaling with the stored blockwise scale.

Unlike uniform quantization, which uses a linear mapping between real and integer domains, NF4 performs nonuniform quantization with higher resolution near zero, aligning with the Gaussian distribution of transformer weights. In all cases, quantized weights are computed once (static quantization), while activations are quantized dynamically during each forward pass, followed by BF16 computation and

re-quantization of outputs. This ensures that each layer experiences quantization-induced effects without compromising training stability.

Overall, these quantization operators provide an efficient, memory-safe, and hardware-agnostic foundation for evaluating low-bit approximation in large language models. They combine realistic quantization error modeling with the flexibility of BF16 computation, enabling consistent benchmarking across INT8, INT4, and NF4 configurations.

2.2 Pruning

Pruning is a classical model compression technique that reduces the number of parameters in a neural network by removing redundant or less important connections. The motivation is rooted in the observation that overparameterized models contain significant redundancy, and that a sparse subnetwork can often achieve comparable accuracy to the dense model. Early work such as Optimal Brain Damage [30] introduced second-order methods for pruning, while Deep Compression [5] popularized magnitude-based pruning, showing that large neural networks can be reduced by up to an order of magnitude in size without significant accuracy loss.

2.2.1 Unstructured Pruning

Unstructured pruning removes individual weights based on a saliency criterion, most commonly weight magnitude. A typical rule is:

$$\hat{W}_{ij} = \begin{cases} W_{ij}, & |W_{ij}| > \tau, \\ 0, & \text{otherwise}, \end{cases} \tag{2.27}$$

where τ is a threshold chosen to achieve the desired sparsity level. This approach is highly flexible and can achieve very high sparsity ratios [5]. However, because the resulting sparsity pattern is irregular, specialized hardware kernels are often required to realize actual inference-time speedups.

2.2.2 Structured Pruning

Structured pruning instead removes entire groups of weights, such as neurons, channels, or attention heads. This preserves dense tensor structures and is therefore more compatible with modern hardware accelerators. For transformers, structured pruning has been applied in several forms:

- **Attention head pruning** selectively removes full heads, leveraging the redundancy across multi-head attention [31].

- **Feedforward pruning** eliminates full hidden neurons in the MLP layers [32].

Structured pruning is less flexible in terms of sparsity ratios compared to unstructured pruning, but has the advantage of translating directly into inference-time efficiency gains without specialized kernels.

2.2.3 Pruning and the Lottery Ticket Hypothesis

The Lottery Ticket Hypothesis [33] further strengthened the theoretical foundation of pruning by showing that large networks contain "winning subnetworks" sparse subnetworks that, when trained in isolation with the same initialization, can match the performance of the full dense model. This observation connects pruning to broader discussions on overparameterization and generalization in deep learning.

Implementation in this Thesis. In this thesis, `L1-unstructured pruning` is implemented within the transformer feedforward layers. Specifically, pruning is applied to the gate projection `gate_proj` inside each MLP block. Following the `torch.nn.utils.prune` API, we prune 5% of the smallest-magnitude weights using L1-norm saliency:

$$\text{prune}(W) = W \odot M, \quad M_{ij} = \mathbf{1}(|W_{ij}| > \tau). \tag{2.28}$$

This approach creates a binary pruning mask that zeroes out low-magnitude weights while preserving higher-magnitude connections. The method corresponds to magnitude-based unstructured pruning as in Deep Compression [5]. Although this does not yield direct inference-time speedups on commodity hardware, it provides a lightweight yet principled approximation method that integrates seamlessly into the unified benchmarking framework developed in this thesis.

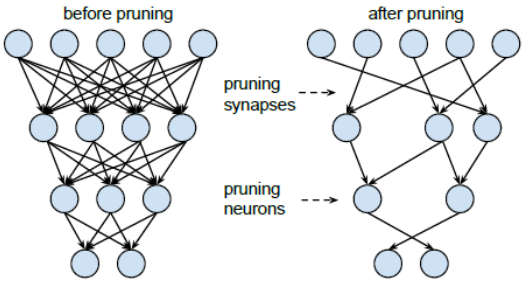

Figure 2.4: Synapses and neurons before and after pruning [34].

2.3 Low-Rank Adaptation (LoRA)

LoRA is a parameter-efficient fine-tuning (PEFT) method that freezes the pretrained weights and learns task-specific, low-rank updates alongside them. Consider a pretrained weight $W_0 \in \mathbb{R}^{d\times k}$. LoRA reparametrizes the adaptation as a rank-r matrix product,

$$W \;=\; W_0 + \Delta W \;=\; W_0 + \frac{\alpha}{r}\, BA, \qquad B \in \mathbb{R}^{d\times r},\; A \in \mathbb{R}^{r\times k}, \tag{2.29}$$

where only A, B are trainable, B is typically initialized to zero and A to a small Gaussian so that ΔW starts at zero; α is a scaling hyperparameter that stabilizes training across different r [6]. The forward pass is then

$$h \;=\; W_0 x \;+\; \frac{\alpha}{r}\, BAx, \tag{2.30}$$

i.e., a residual low-rank correction on top of the frozen base. In transformers, LoRA is commonly applied to the attention projections W_q, W_k, W_v, W_o, while keeping MLP blocks frozen for maximal parameter efficiency [6]. Inference can be latency-free: at serve time one may merge BA into W_0 ($W \leftarrow W_0 + BA$), so the compute graph is identical to a dense model [6].

Recent theory analyzes when low-rank adapters suffice to match a target model exactly and how approximation error scales with rank. For fully connected networks, under mild non-singularity assumptions, a frozen model augmented with rank-R adapters can exactly realize a smaller target model once R exceeds a layerwise threshold; below threshold, the best-achievable error relates to the $(RL{+}1)$-st singular value of a certain layerwise "gap" matrix (extending Eckart–Young–Mirsky to products of low-rank-updated matrices) [14]. For transformers, the analysis shows that, again under mild conditions, rank on the order of half the embedding size is sufficient to exactly match a target transformer of the same size, giving concrete guidance on adapter rank selection [14].

Practical efficiency. Relative to full fine-tuning, LoRA reduces optimizer state and gradient traffic for frozen parameters, cutting training VRAM and storage substantially. The original report shows up to $\sim 3\times$ lower GPU memory during training and $\sim 10{,}000\times$ fewer trainable parameters at rank $r{=}4$ when adapting only W_q/W_v (e.g., GPT-3 175B: 1.2 TB $\rightarrow$ 350 GB training memory; checkpoints $\sim$350 GB $\rightarrow$ 35 MB for adapters) while introducing no extra inference latency when merged [6].

Scalability. Because adapters are tiny and swappable, serving multiple fine-tuned behaviors on a single shared base becomes practical. An open-source Multi-LoRA

server (LoRAX) demonstrates dynamic adapter loading, multi-adapter batching, and tiered caching to host many LoRA models concurrently on one GPU with robust time-to-first-token and throughput under load. In the LoRA Land deployment, 25 LoRA-adapted Mistral-7B models were served to thousands of users on a single 80 GB A100, with detailed latency benchmarks reported [15].

Empirical effectiveness. A large-scale study across 10 base models and 31 tasks (310 LoRA fine-tunes) reports that LoRA adapters trained on 4-bit-quantized bases "QLoRA-style" consistently improve task performance; on average, 4-bit LoRA models exceed their untuned bases and, on that task suite, even surpass GPT-4 under the study's prompting and scoring setup, while enabling economical multi-adapter serving [15]. On the other hand, the authors also note limitations in prompt engineering, evaluation scope, and model variety, encouraging replication [15].

Connections to other PEFT variants. LoRA is compatible with other PEFT ideas: SVD-pruned updates and decomposition variants (e.g., DoRA) target improved conditioning; QLoRA combines LoRA with 4-bit NF4 and paged optimizers to cut memory further; in practice, many works apply LoRA primarily to attention projections for the best quality–efficiency trade-off [6, 14, 15].

Implementation in this Thesis. In this work, LoRA is applied to the self-attention layers of the model following the configuration used in [6]. Specifically, LoRA is integrated into the query and value projection matrices (`q_proj` and `v_proj`) of the transformer attention mechanism, while all other parameters remain frozen. Each modified projection layer is decomposed into a pair of trainable low-rank matrices (A, B), where $A \in \mathbb{R}^{r\times d}$ and $B \in \mathbb{R}^{d\times r}$, initialized such that $BA \approx 0$ at the start of fine-tuning:

$$W' = W + \Delta W, \qquad \Delta W = BA, \tag{2.31}$$

where W denotes the frozen pretrained weight and ΔW the learned low-rank update. In this implementation, the LoRA configuration is defined by the tuple (r, α, p), corresponding respectively to the rank, scaling factor, and dropout probability. The adopted parameters are $r = 8$, $\alpha = 16$, and dropout $p = 0.05$, as shown in the code excerpt below:

```
lora_cfg = LoraConfig(
    r=8, lora_alpha=16, lora_dropout=0.05, bias="none",
    target_modules=["q_proj", "v_proj"],
    task_type=TaskType.CAUSAL_LM)
model = get_peft_model(model, lora_cfg)
model.print_trainable_parameters()
```

The LoRA scaling factor α is set proportionally to the rank ($\alpha = 2r$), which stabilizes the magnitude of the low-rank updates during training. This design ensures that only a small subset of parameters (the low-rank adapters) are trainable, while the vast majority of the model remains frozen, leading to efficient fine-tuning with minimal computational and memory overhead.

2.4 Stochastic Perturbations

LLMs often produce overconfident predictions, even when they are wrong, a phenomenon commonly linked to hallucinations [35]. To address this, uncertainty quantification techniques aim to measure the reliability of model outputs. While traditional sampling methods target aleatoric uncertainty (inherent data variability), they often fail to capture epistemic uncertainty, which stems from the model's limited knowledge or sensitivity to perturbations [36].

Stochastic perturbations provide a lightweight means to approximate epistemic uncertainty by injecting controlled randomness into model inputs or computations. The SPUQ framework [7] formalizes this by generating perturbed inputs (T_i, x_i) from an original input (T_0, x_0), where T denotes temperature and x the prompt:

$$\{(T_i, x_i)\}_{i=1}^{k} = \mathcal{P}(T_0, x_0), \tag{2.32}$$

with $\mathcal{P}$ denoting the perturbation module (e.g., paraphrasing, dumour tokens, system message variations). The perturbed inputs are fed into the LLMs to yield predictions $\{y_i\}_{i=0}^{k}$, where $i = 0$ corresponds to the unperturbed input.

2.4.1 Perturbation-Driven Variance

For additive noise perturbations, as used in our implementation, inputs are modeled as:

$$\tilde{x} = x + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2 I). \tag{2.33}$$

The perturbed output of a linear layer becomes:

$$\tilde{y} = \tilde{x}W^\top + b = (x + \epsilon)W^\top + b. \tag{2.34}$$

The expectation is unbiased:

$$\mathbb{E}[\tilde{y}] = xW^\top + b, \tag{2.35}$$

but the variance encodes sensitivity to perturbations:

$$\mathrm{Var}[\tilde{y}_j] = \sigma^2 \|W_{:,j}\|_2^2. \tag{2.36}$$

Hence, columns of W with larger norms amplify epistemic uncertainty.

2.4.2 Uncertainty Aggregation

SPUQ generalizes stochastic perturbations by combining them with sampling-based uncertainty. Confidence is derived by comparing outputs $\{y_i\}$ across perturbed and unperturbed prompts. Two aggregation methods are proposed [7]:

Inter-sample aggregation. Confidence for output y_j is computed via textual similarity with other samples:

$$c_{\text{inter}}(y_j) = \frac{\sum_{i=0, i\neq j}^{k} s(y_j, y_i)\, w_i}{\sum_{i=0, i\neq j}^{k} w_i}, \tag{2.37}$$

where $s(\cdot, \cdot)$ is a similarity metric (e.g., RougeL, BERTScore), and w_i weighs perturbations based on their closeness to the original prompt.

Intra-sample aggregation. Confidence is averaged across sample-wise uncertainty estimates:

$$c_{\text{intra}} = \frac{1}{k+1} \sum_{i=0}^{k} c(x_i, y_i), \tag{2.38}$$

where $c(x_i, y_i)$ may be derived from likelihood, perplexity, or even self-verbalized uncertainty.

Noise injection complements sampling (aleatoric) by revealing instability across perturbed inputs (epistemic). SPUQ reduces Expected Calibration Error (ECE) by up to 50% across LLMs. Repeated stochastic forward passes provide estimates of mean and variance (Monte Carlo Estimation):

$$\hat{\mu}(x) = \frac{1}{M} \sum_{m=1}^{M} \tilde{y}^{(m)}, \tag{2.39}$$

$$\hat{\sigma}^2(x) = \frac{1}{M} \sum_{m=1}^{M} \left(\tilde{y}^{(m)} - \hat{\mu}(x)\right)^2, \tag{2.40}$$

enabling epistemic uncertainty quantification.

2.4.3 Stochastic Regularization Methods

The idea of injecting noise into neural networks has a long history as a form of stochastic regularization. Rather than perturbing prompts as in SPUQ, our approach is closer to activation-level noise injection, which has been widely explored in deep learning for both efficiency and robustness.

Connection to Dropout. Dropout [37] is perhaps the most well-known stochastic regularization method, where a random binary mask $m \sim \text{Bernoulli}(p)$ is applied to activations:

$$\tilde{x} = m \odot x, \tag{2.41}$$

forcing the network to learn redundant representations. Our Gaussian perturbation variant can be viewed as a smoother alternative, replacing discrete masking with continuous noise:

$$\tilde{x} = x + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2 I). \tag{2.42}$$

Both approaches trade deterministic activations for stochastic ones, encouraging robustness to perturbations.

Bayesian Interpretation. Gal and Ghahramani [38] interpreted dropout as approximate Bayesian inference, where multiple stochastic forward passes correspond to Monte Carlo sampling from a posterior distribution over models. Analogously, Gaussian perturbations applied during inference provide a lightweight Monte Carlo estimate of predictive uncertainty:

$$\hat{\mu}(x) = \frac{1}{M} \sum_{m=1}^{M} f(x + \epsilon_m), \tag{2.43}$$

$$\hat{\sigma}^2(x) = \frac{1}{M} \sum_{m=1}^{M} \Big(f(x + \epsilon_m) - \hat{\mu}(x) \Big)^2, \tag{2.44}$$

where ϵ_m are independent Gaussian samples. Thus, our method extends stochastic regularization into the transformer MLP blocks and enables uncertainty-aware evaluation without modifying prompts.

Approximate Computing Perspective. From an efficiency viewpoint, stochastic perturbations reduce the need for exact, high-precision computation by introducing noise-tolerant approximations [5]. Since the injected noise dominates small floating-point errors, the model can tolerate lower-precision arithmetic in perturbed layers. This aligns with the goals of approximate computing: trading precision for efficiency while preserving accuracy in expectation.

2.4.4 Implementation in This Thesis

In this work, we adopt a simplified stochastic perturbation scheme by injecting Gaussian noise into activations before linear projections:

```
def stochastic_linear(x, weight, bias=None, noise_std=0.01):
    noise = torch.randn_like(x) * noise_std
    return F.linear(x + noise, weight, bias)
```

While SPUQ leverages input-level perturbations for epistemic uncertainty, our activation-level Gaussian noise injection connects naturally to stochastic regularization and Bayesian interpretations of deep networks. It offers a simple yet effective approximation method that integrates seamlessly into the model architecture and can be combined with quantization, pruning, or LoRA in our unified framework.

2.5 Stochastic Memory Masking

Scaling LLMs inevitably runs into memory bottlenecks. During both training and inference, the dominant cost is not always arithmetic computation, but rather storing and moving activations, parameters, and intermediate results across limited GPU memory and bandwidth. For example, the quadratic complexity of self-attention means that even for moderate sequence lengths, memory usage grows prohibitively large. This motivates research into memory approximation methods that deliberately reduce the memory footprint of models at the expense of exactness, thereby enabling deployment at larger scales or under resource-constrained conditions.

While system-level techniques such as FlashAttention [39] address the problem by optimizing tiling and IO patterns, in this thesis we focus on algorithmic approaches. Among these, stochastic memory masking represents a lightweight yet effective strategy to approximate storage by introducing controlled noise directly into activations. Unlike checkpointing or reversible layers, which reduce memory through recomputation, stochastic masking directly enforces sparsity in stored values.

2.5.1 Checkpointing and Sparsification

One classical strategy for memory reduction is activation checkpointing [40]. The key idea is to avoid storing all intermediate activations during the forward pass. Instead, only a small subset of "checkpoints" is retained, and missing activations are recomputed during backpropagation. This reduces memory usage from $O(L)$ to $O(\sqrt{L})$ for L layers, at the cost of extra forward computations.

Reversible residual networks (RevNets) [41] take a different approach by designing layers that are invertible. Here, activations can be reconstructed exactly from later outputs, eliminating the need for explicit storage.

Stochastic methods, by contrast, sacrifice exactness. For example, QSGD [8] quantizes each gradient coordinate to a low-bit stochastic representation while preserving unbiasedness, and DropConnect [42] randomly masks weights during training. These approaches highlight the broader principle: random masking or quantization can approximate limited storage conditions while retaining useful learning signals.

2.5.2 Stochastic Memory Masking for Attention

Building on this line of work, we define stochastic memory masking as a simple but powerful approximation technique applied to attention activations in transformer models. Given an attention output vector $y \in \mathbb{R}^d$, we apply an element-wise Bernoulli mask:

$$\hat{y}_i = \begin{cases} \dfrac{y_i}{1-p}, & \text{with probability } 1-p, \\ 0, & \text{with probability } p, \end{cases} \tag{2.45}$$

where p is the drop probability. The scaling factor ensures that the expectation is unbiased:

$$\mathbb{E}[\hat{y}_i] = y_i, \tag{2.46}$$

while the variance grows with p:

$$\mathrm{Var}[\hat{y}_i] = \frac{p}{1-p} y_i^2. \tag{2.47}$$

In practice, we often omit the rescaling, deliberately introducing a small bias that simulates lossy compression. This makes the approximation closer to realistic storage errors, where magnitudes are systematically underestimated.

2.5.3 Implementation in This Thesis

From an approximate computing perspective, stochastic memory masking trades determinism for efficiency. It does not physically reduce GPU memory allocation, but it conceptually simulates a reduced-memory regime by discarding activations at random. This provides a tunable knob (p) that controls the fidelity–efficiency trade-off.

Unlike structured methods, stochastic masking varies across forward passes, probing the model's robustness to dynamic activation loss. Thus, it offers insight into how resilient large language models are to lossy memory approximations, complementing system-level optimizations like FlashAttention with an algorithmic robustness test.

2.6 Evaluation Metrics and Efficiency Measures

Evaluating approximation techniques for LLMs requires both quality-oriented metrics, which capture linguistic fidelity, and efficiency-oriented measures, which quantify computational resource demands. This dual perspective reflects the standard in NLP evaluation and robustness analysis [9, 10, 11].

2.6.1 Quality Metrics

BLEU. Bilingual Evaluation Understudy (BLEU) is an n-gram precision-based metric with a brevity penalty to avoid favoring short hypotheses:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\frac{1}{N}\sum_{n=1}^{N}\log p_n\right), \tag{2.48}$$

where p_n denotes the modified n-gram precision and $\text{BP} = \min\left(1, e^{1-|R|/|C|}\right)$ is the brevity penalty for candidate length $|C|$ against reference length $|R|$. BLEU remains widely used in text generation benchmarks, although its sensitivity to exact n-gram overlap has motivated several adaptations [9].

ROUGE-L. ROUGE-L evaluates recall-oriented overlap using the longest common subsequence (LCS):

$$\text{ROUGE-L} = \frac{(1+\gamma^2)\cdot P_{\text{LCS}}\cdot R_{\text{LCS}}}{R_{\text{LCS}} + \gamma^2 P_{\text{LCS}}}, \tag{2.49}$$

where $P_{\text{LCS}} = \frac{\text{LCS}(C,R)}{|C|}$ and $R_{\text{LCS}} = \frac{\text{LCS}(C,R)}{|R|}$. ROUGE-L has been shown effective for summarization and generative evaluation where ordering matters[9].

F1 Score. Precision and recall can be applied at the token level to capture overlap between candidate tokens C and reference tokens R:

$$\text{F1} = \frac{2\cdot\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}, \tag{2.50}$$

with $\text{Precision} = \frac{|C\cap R|}{|C|}$ and $\text{Recall} = \frac{|C\cap R|}{|R|}$. Probabilistic extensions generalize this formulation by weighting overlaps with confidence values, making it suitable for stochastic generation settings [11].

SBERT Similarity. Semantic textual similarity was evaluated using SBERT, a transformer-based model that maps sentences into a shared embedding space optimized through contrastive and siamese training objectives [43]. Unlike lexical metrics such as BLEU or ROUGE that rely on n-gram overlap, SBERT computes cosine similarity between dense vector representations:

$$\text{SBERT}(C,R) = \frac{\langle \mathbf{e}_C, \mathbf{e}_R\rangle}{\|\mathbf{e}_C\|\cdot\|\mathbf{e}_R\|}, \tag{2.51}$$

where $\mathbf{e}_C$ and $\mathbf{e}_R$ denote the embeddings of candidate and reference sentences, respectively. Values range from -1 to 1, but well-aligned natural language pairs typically fall in the interval [0,1], making the metric effective for capturing paraphrastic

and semantic equivalence even in the presence of substantial lexical variation. This makes SBERT particularly suitable for evaluating approximated models, whose outputs may deviate at the token level while maintaining semantic fidelity.

Perplexity. Perplexity measures the cross-entropy between the model distribution P and the empirical sequence x_1^T:

$$\mathrm{PPL} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log P(x_t \mid x_{<t})\right). \tag{2.52}$$

Lower perplexity indicates better predictive power. It is also used for anomaly detection and robustness assessment in LLMs [10].

2.6.2 Efficiency Measures

Evaluating approximation techniques requires not only accuracy-based metrics but also measurements of computational and memory efficiency. In this work, efficiency is analyzed in terms of inference latency and model size. These measures provide complementary views of runtime cost and resource utilization.

Inference Latency. Latency represents the total time required for model inference and serves as a direct measure of computational efficiency. Inference time T is defined as the wall-clock difference between the start and end of a single evaluation loop:

$$T = t_{\mathrm{end}} - t_{\mathrm{start}}, \tag{2.53}$$

where t_{start} and t_{end} correspond to timestamps collected immediately before and after the forward pass of the inference, respectively. In practice, the evaluation loop measures this duration using `time.time()` surrounding the `model.generate()` call. The forward generation includes token sampling with top-p nucleus sampling and temperature scaling. This timing reflects end-to-end inference, including decoding and post-processing.

Output File Size. To assess the effect of approximation on textual generation volume, all model predictions, references, and prompts were stored in serialized JSONL format. The file size in kilobytes (KB) is computed as:

$$S_{\mathrm{out}} = \frac{\mathrm{Bytes(file)}}{1024}. \tag{2.54}$$

This measure indirectly reflects verbosity and compression effects induced by quantization and stochastic methods.

Model Size. To quantify memory efficiency, the effective model size is computed as the sum of all parameter and buffer tensors, weighted by their numerical precision. Let the model contain K tensors with element counts n_k and bit-widths b_k. The total memory footprint is:

$$S_{\text{model}} = \sum_{k=1}^{K} \frac{n_k \cdot b_k}{8 \cdot 1024^2} \quad \text{(MB)}. \tag{2.55}$$

Full-precision models (e.g., BF16 or FP16) use uniform bit-widths, while approximation techniques modify b_k through quantization (INT8, INT4, NF4), low-rank adaptation, or structural pruning. Quantization reduces b_k, shrinking each tensor, whereas pruning reduces n_k by removing weights. LoRA introduces small rank-decomposition matrices whose contribution is negligible compared to the full dense layers. Thus, the final model size captures the cumulative effect of compression, precision scaling, and structural sparsity, providing a hardware-agnostic indicator of memory efficiency for deployment-oriented model design.

Chapter 3

Methodology

This chapter outlines the methodology adopted in this thesis. Building upon the theoretical background presented in Chapter 2, it describes the datasets, baseline models, implemented approximation techniques, evaluation metrics, and experimental protocol. The goal is to establish a reproducible framework for systematically studying efficiency–accuracy trade-offs in instruction-tuned LLMs.

A modular approximation pipeline was developed to integrate and evaluate efficiency-oriented techniques in transformer architectures. The framework supports five primary approximation methods, which are quantization, pruning, stochastic perturbation, memory masking, and LoRA, each of which can be activated individually or in combination. This modular design enables controlled experimentation, allowing direct comparison of approximation effects across datasets and models. The workflow consists of four major stages which are model selection and initialization, approximation integration, fine-tuning and evaluation, and metric collection and analysis.

Firsly, two compact instruction-tuned transformer models were selected which are LLaMA-3.2-1B-Instruct and Gemma-3-1B-it. Both were chosen for their open availability, manageable computational footprint, and compatibility with the approximation methods. These architectures provide a representative testbed for studying approximation in mid-scale LLMs. Secondly, the following approximate computing methods were implemented.

- **Quantization** is implemented as a simulated low-precision scheme (INT8, INT4, NF4) in which weights are quantized statically and activations dynamically, while computations remain in BF16 to ensure numerical stability and hardware independence. Weights are quantized once at model initialization or via a dedicated `quantize_model()` pass, whereas activations are quantized dynamically before each forward pass. Matrix multiplications are performed in BF16, and outputs are subsequently requantized before propagation to the

next layer.

- **Pruning** enforces structured sparsity by removing a fixed fraction of weights with the lowest magnitudes according to an L_1-norm criterion. This reduces the parameter count and computational load while maintaining the overall network topology and representational capacity.

- **Stochastic perturbation** injects Gaussian noise into activations before self-attention projections(before query, key and value projections) within the MLP gate layers, simulating arithmetic uncertainty and the behavior of stochastic hardware accelerators. By targeting activation projections in the MLP rather than self-attention layers, this approach enhances robustness to noise-induced variability while maintaining training stability and controllable experimental conditions.

- **Memory masking** randomly deactivates a small subset of tensor elements within self-attention outputs, simulating approximate or lossy memory access.

- **LoRA** introduces efficient trainable subspaces for fine-tuning large models by reparameterizing each pretrained weight matrix W_0 as

$$W = W_0 + \frac{\alpha}{r} BA,$$

  where $A \in \mathbb{R}^{r \times d}$ and $B \in \mathbb{R}^{d \times r}$ are low-rank adapters with rank $r \ll d$. LoRA is applied to the query and value projection matrices (`q_proj`, `v_proj`) within each attention block to enable efficient adaptation with minimal parameter overhead. Only the LoRA parameters (A, B) are trainable, while all pretrained weights remain frozen. The applied configuration uses $r = 8$, $\alpha = 16$, and dropout $p = 0.05$, corresponding to $\alpha = 2r$.

All modules were parameterized through a unified configuration system, allowing flexible combinations such as INT8 + LoRA, Pruned + LoRA, or INT8 + Pruned + LoRA. This ensured reproducible comparisons under identical training conditions.

Thirdly, each approximated model underwent fine-tuning on three instruction-following datasets which are Alpaca, Databricks-Dolly-15k, and AgentInstruct. Due to GPU memory and runtime constraints, the Alpaca and Dolly datasets were subsampled to 15,000 examples each while maintaining diversity across instruction types. Fine-tuning was conducted using the AdamW optimizer with cosine learning rate decay and linear warmup. Models were trained for 3–5 epochs, using mixed batch sizes (2–16 samples) depending on memory availability. All experiments used a sequence length of 128 tokens. Approximation parameters were fixed throughout training to ensure stability and fair comparison.

Finally, to assess the effectiveness of approximation, complementary quality and efficiency metrics, which are BLEU score, ROUGE-L score, F1 score, SBERT score inference time, output size and final model size, were collected. The evaluation protocol was identical for all configurations, measuring both linguistic quality and efficiency.

3.1 Datasets

To evaluate approximation techniques under diverse instruction-following conditions, we employ three representative instruction-tuning datasets with complementary properties: **Alpaca**, **Databricks-Dolly-15k**, and **AgentInstruct**. These datasets differ in scale, construction methodology, and task coverage, which together enable a systematic assessment of robustness and efficiency trade-offs.

3.1.1 Alpaca

The Alpaca dataset [44] was introduced by Stanford's Center for Research on Foundation Models (CRFM) as an accessible, low-cost resource for instruction tuning. It contains approximately 52,000 instruction–response pairs generated synthetically using OpenAI's text-davinci-003 (GPT-3.5). The construction pipeline follows the Self-Instruct paradigm [45], where a seed set of 175 manually written prompts was used to bootstrap GPT-3.5 into producing thousands of novel instructions and corresponding demonstrations.

To reduce cost and maximize diversity, several modifications were made compared to the original Self-Instruct pipeline: (i) using the more powerful text-davinci-003 engine, (ii) employing aggressive batch decoding (20 generations per batch), (iii) discarding the distinction between classification and non-classification instructions, and (iv) generating a single demonstration per instruction rather than multiple. This yielded a large corpus for under $500, making Alpaca one of the most cost-effective open instruction datasets.

Data Fields. Each Alpaca entry has:

- `instruction`: natural-language description of the task.
- `input`: optional context (present in $\sim$40% of examples).
- `output`: the response generated by GPT-3.5.
- `text`: concatenation of instruction, input, and output using a standardized prompt template.

From a theoretical standpoint, Alpaca exemplifies the synthetic data distillation paradigm, where a strong proprietary teacher model transfers its knowledge into an open student model. While synthetic generation ensures fluency and uniformity, it also propagates biases and factual errors from the teacher model. Furthermore, Alpaca's license restricts it to non-commercial use. Despite these caveats, Alpaca is widely adopted for controlled studies of approximation, since its scale and relative cleanliness allow systematic exploration of pruning stability, quantization degradation, and LoRA.

3.1.2 Databricks-Dolly-15k

The Databricks-Dolly-15k dataset [46] represents a contrasting design philosophy. Instead of synthetic generations, it consists of approximately 15,000 human-authored instruction–response pairs contributed by thousands of Databricks employees. Prompts were crafted across eight categories, closely following the taxonomy outlined in the InstructGPT paper [47]: brainstorming, classification, closed QA, open QA, summarization, information extraction, creative writing, and free-form instructions. Annotators were instructed not to use generative AI systems when writing responses, and to ground factual categories (e.g., QA, summarization) in Wikipedia.

The annotation process was deliberately lightweight: employees were given short descriptions and examples but little formal rubric, leading to linguistic richness and pragmatic variety. Midway through collection, contributors could also answer prompts written by colleagues, further increasing lexical diversity and paraphrasing variety. As a result, Dolly-15k reflects human creativity, ambiguity, and stylistic idiosyncrasies often missing in synthetic corpora.

Data Fields. Each Dolly entry has:

- `instruction`: the human-authored prompt.
- `context`: optional supporting text (e.g., a Wikipedia passage).
- `response`: the human-written answer.

Dolly is notable for three reasons. First, it is the first large-scale, open-source, human-generated instruction dataset released under a permissive CC-BY-SA 3.0 license, enabling both academic and commercial use. Second, its linguistic diversity makes it a stronger testbed for approximation robustness: stochastic masking, pruning, and quantization must handle real-world variation. This positions Dolly-15k as a benchmark for generalization under authentic, human-like conditions.

3.1.3 AgentInstruct

AgentInstruct [48] originates from the AgentTuning project, designed to equip LLMs with agentic capabilities. Unlike Alpaca and Dolly, which are primarily single-turn, AgentInstruct emphasizes multi-step interaction trajectories involving planning, reasoning, and tool use. A typical entry may require decomposing a high-level instruction into sub-tasks, invoking external tools (e.g., search or calculator), and integrating intermediate results into a coherent final answer.

The dataset was constructed by generating candidate agentic trajectories (partly via GPT-4), filtering them through execution success and alignment criteria, and mixing them with general-purpose sources such as ShareGPT. The final release includes $\sim$1,800 high-quality trajectories. Although smaller in scale, each trajectory is semantically dense, capturing structured reasoning and long-context dependencies.

Data Fields. Each entry in the AgentInstruct dataset is structured as a multi-turn conversation between a human (or user) and an AI assistant. The data is represented as a sequence of messages, each of which contains three core attributes:

- **from**: the speaker role, which is either the *human/user* (corresponding to the instruction) or the *assistant/gpt* (corresponding to the model's response).
- **value**: the textual content of the message. For user turns, this contains the natural language instruction or follow-up question. For assistant turns, it contains the model's intermediate reasoning, tool invocation, or final answer.
- **loss**: a supervision flag (Boolean or null), which specifies whether a given assistant step is included as part of the supervised training signal.

For downstream formatting, the dataset can be distilled into instruction–response pairs by selecting the latest message authored by the *human/user* as the instruction, and the subsequent *assistant/gpt* message as the response. This ensures a standardized alignment of prompts with their corresponding model-generated answers while preserving the conversational and multi-step reasoning nature of the dataset.

From an approximation perspective, AgentInstruct is a natural stress test for memory-saving techniques such as stochastic masking or low-rank compression. Multi-step reasoning requires retaining intermediate states over long contexts, and lossy approximations may compromise logical consistency and planning depth.

Summary. Together, these datasets span complementary aspects of instruction-following:

- **Alpaca:** Large-scale, synthetic, and clean. Facilitates controlled experiments on approximation sensitivity.

- **Dolly-15k:** Medium-scale, human-authored, and diverse. Tests robustness against natural human variation.

- **AgentInstruct:** Small-scale, multi-step, agentic reasoning. Evaluates approximation impact on reasoning depth and memory.

By combining synthetic uniformity (Alpaca), authentic human diversity (Dolly), and structured agentic complexity (AgentInstruct), this suite provides a balanced foundation for analyzing approximation methods' training in LLMs.

3.2 Models

This work employs two compact instruction-tuned LLMs as baselines: **LLaMA-3.2-1B-Instruct**, released by Meta, and **Gemma-3-1B-Instruct**, released by Google DeepMind. Both models are representative of modern transformer-based LLMs, while being sufficiently lightweight to enable experimentation with approximate training techniques under constrained resources. Despite their relatively small size compared to frontier-scale LLMs (tens or hundreds of billions of parameters), they preserve the architectural sophistication of current-generation autoregressive transformers, making them highly suitable for controlled benchmarking of approximation methods such as pruning, quantization, and stochastic memory masking.

3.2.1 LLaMA-3.2-1B-Instruct

The LLaMA (Large Language Model Meta AI) family has become a cornerstone of open research in generative AI, providing a transparent and accessible alternative to proprietary models [49]. LLaMA-3.2-1B-Instruct is a compact, instruction-tuned variant of the LLaMA 3.2 family, with approximately 1 billion parameters. Although far smaller than the largest models in the LLaMA-3.2 series (e.g., 70B), the 1B variant incorporates the same core architectural innovations, ensuring structural comparability across scales.

Architecture. LLaMA-3.2 models adopt a decoder-only transformer architecture, optimized for autoregressive next-token prediction. Key architectural components include:

- **Grouped-Query Attention (GQA)**: Instead of allocating a unique set of key-value projections per attention head, GQA shares key-value projections across groups of heads [50]. This reduces memory footprint and improves inference efficiency while maintaining competitive quality.

- **Rotary Positional Embeddings (RoPE)**: Positional encodings are incorporated directly into the attention mechanism via rotation in complex space, enabling extrapolation to longer context windows compared to absolute embeddings [51].
- **SwiGLU Activation Functions**: The feed-forward networks (FFNs) employ a SwiGLU nonlinearity [52], which has been shown to improve training stability and generalization compared to ReLU or GELU.
- **Long-Context Extensions**: LLaMA-3.2 incorporates efficient scaling of context length, enabling windows of up to 128k tokens in larger variants, with proportionally scaled-down limits for smaller models.

Instruction Tuning. The Instruct variant is obtained by supervised fine-tuning of the pretrained model on prompt–response pairs derived from instruction datasets (e.g., Alpaca, Dolly-15k). Further refinements may include reinforcement learning from human feedback (RLHF) or Direct Preference Optimization (DPO), which align model behavior with human-preferred responses [47]. Instruction tuning improves zero-shot usability, ensuring that the model interprets prompts as natural instructions rather than arbitrary text completion tasks.

Role in Benchmark. LLaMA-3.2-1B-Instruct serves as a canonical open-weight, instruction-tuned baseline. Its compact size allows us to fine-tune with multiple approximation strategies under realistic hardware constraints. Furthermore, its widespread use in open-source research ensures that approximation results are broadly comparable to the literature.

3.2.2 Gemma-3-1B-Instruct

Gemma is Google DeepMind's family of lightweight, open-source language models introduced in 2025 [53]. The Gemma-3-1B-Instruct model is the 1B-parameter instruction-tuned variant of Gemma 3, explicitly designed for efficiency, multilinguality, and modularity. It is positioned as a counterpart to LLaMA , offering an alternative open foundation with strong performance in constrained environments.

Architecture. Like LLaMA, Gemma models are decoder-only transformers. However, Gemma introduces a number of design choices that emphasize deployment efficiency and adaptability:

- **Grouped-Query Attention (GQA)** and **multi-query variants** are employed to further reduce memory usage during inference.

- **Extended Context Windows**: The 1B-it variant supports sequences of up to 32k tokens, which is unusually long for such a compact model, enabling experiments on long-context approximation trade-offs.
- **Optimized Embedding Layers**: Gemma models apply more parameter-efficient embedding layers, reducing memory bottlenecks for multilingual vocabularies.
- **Parameter-Efficient Scaling**: Architectural choices are explicitly tuned to maximize FLOPs utilization at small parameter scales, providing a more competitive baseline compared to other lightweight open models.

Instruction Tuning. The Gemma-3-1B-Instruct model undergoes structured instruction tuning using both supervised datasets (e.g., human-authored instructions, synthetic prompt–response pairs) and reinforcement signals. The instruction format is standardized around explicit role tokens (`<start_of_turn>`, `<end_of_turn>`), ensuring consistency across tasks and improving robustness to conversational multi-turn alignment [53]. This contrasts with LLaMA's simpler Alpaca-style instruction-response formatting, highlighting methodological diversity in instruction-tuning pipelines.

Role in Benchmark. Gemma-3-1B-Instruct provides an architectural and methodological contrast to LLaMA-3.2-1B-Instruct. While both models are decoder-only transformers, differences in training data, tokenization, and instruction formatting allow us to probe the portability of approximation methods. For instance, quantization strategies may interact differently with Gemma's embedding layers compared to LLaMA.

Comparative Analysis. The two models complement each other along several dimensions:

- **Scale**: Both are $\sim$1B parameter models, light enough for systematic benchmarking, yet architecturally representative of frontier-scale LLMs.
- **Instruction-Tuning Paradigm**: LLaMA emphasizes Alpaca-style distillation and DPO alignment, whereas Gemma uses structured conversation templates and RLHF pipelines. This diversity ensures our approximations are tested across multiple alignment strategies.
- **Architectural Efficiency**: LLaMA prioritizes efficient scaling via GQA and RoPE, while Gemma emphasizes parameter-efficient embeddings and long-context extensions.

- **Research Relevance**: LLaMA serves as a widely-used, standard benchmark model, while Gemma represents a newer generation of compact, multilingual, and efficiency-oriented open models.

Together, these models establish a balanced evaluation environment: LLaMA-3.2-1B-Instruct as a stable, widely-adopted open benchmark, and Gemma-3-1B-Instruct as a forward-looking, efficiency-driven alternative. Their shared compactness makes them well suited for the experimental focus of this thesis, benchmarking approximation-aware training, while their architectural and methodological contrasts provide insight into the generalizability of approximation techniques across model families.

3.3 Approximation Techniques

The approximation techniques applied in this thesis directly follow the theoretical foundations outlined in Chapter 2. Each method is adapted into a practical implementation pipeline.

3.3.1 Quantization

Quantization in this thesis is implemented using a simulated quantization. Pretrained transformer models such as LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct are quantized into low-precision representations (NF4, INT4, or INT8) while maintaining floating-point computations for numerical stability.

Quantization pipeline. Weights are quantized once either at model initialization or via a dedicated `quantize_model()` pass while activations are quantized dynamically before each forward pass. Computations, including matrix multiplications, are performed in BF16, and outputs are requantized before being propagated to subsequent layers. This scheme strikes a balance between numerical efficiency and hardware independence, preserving the accuracy advantages of simulated quantization while avoiding integer-only computation constraints [4, 12].

Quantization operator. Let $W \in \mathbb{R}^{m \times n}$ denote a weight matrix. Quantization maps W to a discrete set of representable values $\hat{W}$ using a scale factor s and zero-point Z:

$$\hat{W} = \text{Quantize}(W; s, Z) = \text{clip}\left(\left\lfloor \frac{W}{s} \right\rceil + Z, q_{\min}, q_{\max}\right), \tag{3.1}$$

where $q_{\min}$ and $q_{\max}$ define the integer range (e.g., $[-127,127]$ for INT8), and $\lfloor \cdot \rceil$ denotes rounding to the nearest integer. The corresponding dequantization step is

given by:

$$\tilde{W} = s \cdot (\hat{W} - Z), \tag{3.2}$$

yielding an approximation error $\varepsilon = W - \tilde{W}$. In this work, such quantization is performed for all linear weights, while dynamic quantization is applied to activations during inference.

Comparison with PTQ and QAT. The proposed method differs from classical approaches:

- **PTQ:** Post-training quantization applies Quantize$(\cdot)$ directly to pretrained weights without retraining. While computationally efficient, it often degrades accuracy below 8-bit precision [19].
- **QAT:** Quantization-aware training retrains all model parameters under simulated quantization in both forward and backward passes [4]. It achieves high fidelity but is computationally prohibitive for billion-scale LLMs.

Uniform quantization. Uniform symmetric quantization maps each real-valued weight w_i into an integer $\hat{w}_i$ according to:

$$\hat{w}_i = \text{round}\left(\frac{w_i}{s}\right), \quad s = \frac{\max(|W|)}{2^{b-1} - 1}, \tag{3.3}$$

where b denotes the bit width. For 8-bit and 4-bit quantization, the representable ranges are $[-127,127]$ and $[-7,7]$, respectively. Although INT8 provides stable approximations with minimal degradation, INT4 may introduce higher quantization error, particularly in attention and feed-forward layers with skewed weight distributions. Implementation follows:

```
def quantize_tensor_uniform(tensor, bits=8):
    qmax = 2 ** (bits - 1) - 1
    max_val = tensor.abs().max(dim=-1, keepdim=True).values
    scale = torch.clamp(max_val / qmax, min=1e-8)
    q = torch.clamp((tensor / scale).round(), -qmax, qmax)
    return q, scale
```

NF4 quantization. While the original NF4 (NormalFloat-4) scheme [29] defines a statistically optimal 4-bit codebook derived from a Gaussian prior, our implementation adopts an NF4-inspired design that follows similar principles but employs a distinct computational structure for practical GPU execution. Specifically, the quantizer still relies on a fixed 16-value codebook which approximates

equal-probability bins of a standard normal distribution, providing higher resolution around zero where most weights concentrate.

$$\begin{aligned}\mathcal{C}_{\text{NF4}} = \{ &-1.0000, -0.6962, -0.5251, -0.3949, \\ &-0.2844, -0.1848, -0.0911, 0.0000, \\ &0.0911, 0.1848, 0.2844, 0.3949, \\ &0.5251, 0.6962, 0.8682, 1.0000\}.\end{aligned} \tag{3.4}$$

$$\alpha_b = \max_{i,j} |W_{b,ij}|, \tag{3.5}$$

$$\tilde{W}_b = \frac{W_b}{\alpha_b}, \tag{3.6}$$

$$\hat{W}_b = Q_{\text{NF4}}(\tilde{W}_b), \tag{3.7}$$

$$W_b^q = \alpha_b \cdot \hat{W}_b. \tag{3.8}$$

Unlike the original QLoRA formulation, which performs blockwise quantization with precomputed codebook mappings, our NF4 variant implements element-wise nearest-neighbor assignment directly on GPU. Each tensor is divided into blocks of 64 elements for scaling, followed by a streaming search over the 16 codebook entries per block. This eliminates large intermediate tensors and avoids out-of-memory (OOM) errors common in naive 4D broadcasting approaches. The quantized indices and per-block scales are stored compactly and later dequantized via direct codebook lookup. This approach preserves the statistical intuition of NF4 while offering a GPU-efficient, memory-safe quantization routine that can be applied dynamically during training or inference. In contrast to Dettmers et al. [29], our variant prioritizes runtime efficiency over exact Gaussian optimality, making it well-suited for large-scale experimentation under resource constraints.

3.3.2 Pruning

Pruning reduces model complexity by eliminating redundant parameters. In this benchmark, we implement unstructured magnitude pruning, where individual weights are removed based on their absolute magnitude.

Unstructured magnitude pruning. Weights are pruned under an L_1 threshold:

$$\hat{W}_{ij} = \begin{cases} W_{ij}, & |W_{ij}| \geq \tau, \\ 0, & |W_{ij}| < \tau, \end{cases} \tag{3.9}$$

where τ is dynamically chosen to achieve a target sparsity level. This fine-grained approach maximizes parameter removal but leads to irregular sparsity patterns that are difficult to exploit on standard hardware.

The following Python snippet shows our implementation of unstructured pruning integrated into the training pipeline.

```
if config.get("prune"):
    prune.l1_unstructured(gate, name="weight", amount=0.05)
```

In our implementation, pruning is applied exclusively to the MLP components of transformer blocks, specifically the gate projection. This choice is motivated by three factors: (i) the MLP layers constitute the majority of parameters in LLaMA and Gemma models, so sparsification here yields the largest reduction in size; (ii) attention projections are highly sensitive to pruning and even small perturbations can destabilize the softmax distributions, whereas MLP transformations exhibit higher redundancy and pruning tolerance; and (iii) PyTorch's pruning utilities integrate seamlessly with linear MLP layers, making them a natural starting point. Concentrating pruning in MLPs thus balances efficiency gains with model robustness, consistent with observations in prior pruning studies [19].

Note on structured pruning. Structured pruning (removing entire neurons, channels, or projections) is not implemented in the current benchmark but remains an important future extension, as it produces hardware-friendly sparsity.

3.3.3 LoRA

LoRA introduces efficient trainable subspaces to fine-tune large models [6]. Given a pretrained weight matrix W_0, LoRA reparameterizes:

$$W = W_0 + \frac{\alpha}{r} BA, \tag{3.10}$$

where $A \in \mathbb{R}^{r \times d}$ and $B \in \mathbb{R}^{d \times r}$ are low-rank adapters with rank $r \ll d$. During fine-tuning, only A and B are updated, while W_0 remains frozen.

Application to Transformers. In this thesis, LoRA is applied to the query and value projection matrices (`q_proj`, `v_proj`) within each attention block, following empirical best practices for instruction-tuned models. This ensures efficient adaptation with minimal parameter overhead. The applied configuration uses $r = 8$, $\alpha = 16$, and dropout $p = 0.05$, corresponding to $\alpha = 2r$. The implementation via the Hugging Face's PEFT library is as follows:

```
lora_cfg = LoraConfig(
    r=8, lora_alpha=16, lora_dropout=0.05, bias="none",
    target_modules=["q_proj", "v_proj"],
    task_type=TaskType.CAUSAL_LM)
```

```
model = get_peft_model(model, lora_cfg)
model.print_trainable_parameters()
```

Only LoRA adapters (A, B) are trainable, while all pretrained parameters are frozen.

3.3.4 Stochastic Perturbations

To probe robustness against arithmetic noise, we inject Gaussian perturbations into activations before self-attention projections (the `q_proj`, `k_proj`, and `v_proj` matrices) within the MLP gate layers.

$$\hat{y} = (x + \epsilon)W^\top + b, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2 I). \tag{3.11}$$

This simulates the uncertainty of stochastic computing and approximate arithmetic units, which may arise in low-power hardware accelerators. The following Python snippet shows our implementation of stochastic perturbations injected into MLP gate projections using a custom stochastic linear operator.

```
def stochastic_linear(x, weight, bias=None, noise_std=0.01):
    noise = torch.randn_like(x) * noise_std
    return F.linear(x + noise, weight, bias)
```

In our benchmark, Gaussian noise was injected only into the MLP gate projections. This design reflects three considerations: (i) the MLP layers account for the majority of model parameters and provide redundancy that makes them more tolerant to stochastic perturbations; (ii) attention projections are highly sensitive to noise, where even small perturbations can destabilize token alignment and degrade coherence; and (iii) implementation is straightforward in MLP gate projections, which are single linear transformations, while attention layers involve multiple interdependent projections. Thus, targeting MLP layers ensures both robustness and experimental controllability when evaluating stochastic approximation.

3.3.5 Stochastic Memory Masking

While true memory approximation typically involves lossy compression of activations or quantization of key-value (KV) caches, we adopt a simplified proxy approach to study robustness. Specifically, we introduce stochastic masking in the outputs of attention layers, mimicking the effect of unreliable or lossy memory propagation.

Formally, given attention activations y, a Bernoulli mask M is sampled element-wise:

$$\hat{y} = y \odot M, \quad M \sim \text{Bernoulli}(1 - p), \tag{3.12}$$

where p is the masking probability. This resembles dropout applied during inference, but is interpreted as random activation loss rather than a training regularizer. Unlike pruning, which permanently removes weights, stochastic masking introduces random erasures at each forward pass. To integrate this into the benchmark, we wrap the forward pass of self-attention modules such that a fraction $p = 0.05$ of the activations are masked:

```
if config.get("memory_approx") and hasattr(layer, "self_attn"):
    drop_p = 0.05
    def wrapped_forward(*args, orig=orig, drop_p=drop_p, **kw):
        out = orig(*args, **kw)
        mask = (torch.rand_like(out[0]) > drop_p)
        return (out[0] * mask,) + out[1:]
    layer.self_attn.forward = wrapped_forward
```

This stochastic memory masking does not perform actual compression, but it provides a controlled way to approximate the behavior of lossy memory. By introducing transient activation loss, it allows us to probe the resilience of LLMs under degraded memory fidelity.

3.3.6 Combinations of Approximations

A major contribution of this work lies in enabling composable approximation strategies. Rather than evaluating each method in isolation, the benchmark systematically explores combinations of techniques, mirroring real-world scenarios where efficiency, compression, and adaptability must be jointly optimized. By composing quantization, pruning, stochastic perturbations, memory masking, and LoRA, we study both pairwise and higher-order interactions between methods.

Double Combinations. Pairwise combinations are used to test complementary effects between structural, arithmetic, and representational approximations:

- **Quantization + LoRA:** The base model is quantized (INT8, INT4, or NF4) while the LoRA adapters are applied. This hybrid configuration maintains the adaptability of LoRA while substantially reducing memory and storage requirements.
- **Pruning + LoRA:** Sparse pruned backbones are augmented with trainable LoRA adapters, allowing the model to recover expressivity lost through weight removal.
- **Stochastic Perturbations + LoRA:** Perturbations are applied to the linear layers while LoRA adapts to the induced stochasticity during training, testing robustness under noisy computation.

- **Memory Masking + LoRA:** Stochastic masking is applied to attention activations while LoRA adapters fine-tune the frozen backbone, capturing adaptation under partial information degradation.

Triple Combinations. To investigate cumulative interactions, aggressive three-way configurations are explored:

- **INT8 + Pruned + LoRA:** A jointly compressed setup where pruning enforces sparsity, INT8 quantization reduces memory bandwidth, and LoRA fine-tunes the model for recovery.
- **INT4 + Pruned + LoRA:** Represents the most extreme configuration, combining aggressive quantization and structural sparsity with minimal fine-tuning overhead.

This compositional evaluation demonstrates how hybrid approximations can jointly optimize memory, latency, and performance, providing a unified framework for controlled degradation and recovery in LLMs.

Chapter 4

Experiments and Results

4.1 Experimental Setup

We evaluate the approximation techniques introduced in Chapter 3.3 on two instruction-tuned transformer models that are LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct. Both models were chosen for their compact size, open availability, and compatibility with approximate computing methods.

Datasets. Three instruction-following datasets were used in our experiments: Alpaca, Databricks-Dolly-15k, and AgentInstruct. To reduce overfitting and better evaluate the model's ability to generalize across different instruction styles, we adopted a cross-dataset evaluation strategy. In this setting, the model is fine-tuned on one dataset and evaluated on a different one. For example, in the Alpaca→Agent configuration, the model is trained on a 5,000-sample subset of Alpaca and validated on the AgentInstruct dataset. Similarly, Alpaca→Dolly and Agent→Dolly configurations were used to introduce domain shift between training and evaluation. These cross-dataset evaluations allow us to assess whether the model learns transferable instruction-following behavior rather than memorizing dataset-specific patterns.

Baselines. Baseline inference results were taken using the pretrained checkpoints of both models. This serves as reference points for comparing relative improvements achieved through the application of approximate computing techniques.

Training Configuration. Fine-tuning was carried out using the AdamW optimizer, applied only to parameters marked as trainable (for example, LoRA update matrices when LoRA was enabled). A cosine learning rate schedule with a warmup ratio of 20% of the total training steps was used. Training was performed for

3-5 epochs depending on the experiment, with gradient accumulation (typically 2 steps) employed to accommodate GPU memory limitations. The loss function was the standard cross-entropy loss with masking applied to padding or ignored labels. Perplexity was computed at the end of each epoch to monitor the progression of training.

Evaluation Protocol. The effectiveness of each approximation method was evaluated using metrics that quantify both linguistic accuracy and model efficiency. Linguistic performance was measured using BLEU, ROUGE-L, F1, and SBERT scores. BLEU captures n-gram precision between generated responses and reference outputs. ROUGE-L measures the longest common subsequence, reflecting semantic coverage and structural coherence. The F1 score balances precision and recall at the token level,

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}, \tag{4.1}$$

ensuring robustness to differences in response length. In addition to these token-overlap metrics, we report a SBERT-based similarity score, obtained by computing the cosine similarity between sentence embeddings of the generated output and the reference response. SBERT values lie in [0,1] and provide a semantic measure of alignment that is more tolerant to paraphrasing and variation in surface form, complementing BLEU and ROUGE-L.

To characterize model efficiency, we tracked three complementary indicators. Firstly, model size was estimated by accounting for parameter and buffer representations, including reduced-precision formats introduced by quantization. Secondly, the size of the generated outputs was recorded as an indicator of response verbosity and determinism. Thirdly, the average inference time was measured to capture the latency impact of each approximation technique and quantify the trade-off between computational efficiency and linguistic performance.

4.2 Evaluation Results

4.2.1 Inference Results for the Dolly-15k Dataset(Fine-tuned with Alpaca Dataset)

Model	Approximation Technique	BLEU	ROUGE-L	SBERT	Inf. Time (s)	Out. Size (KB)	Model Size (MB)
Baseline							
LLaMA-3.2-1B	——	0.58	0.67	0.88	2.2	163.8	2,858
Gemma-3-1B	——	0.25	0.46	0.68	7.9	121.3	2,483
Fine-tuned							
LLaMA-3.2-1B	Pruned	0.62	0.69	0.90	2.0	164.9	3,370
Gemma-3-1B	Pruned	0.26	0.46	0.69	8.0	122.4	2,878
LLaMA-3.2-1B	INT8	0.86	0.90	0.94	1.3	154.3	1,681
Gemma-3-1B	INT8	0.25	0.46	0.68	15.3	125.5	1,532
LLaMA-3.2-1B	INT4	0.36	0.48	0.81	7.6	184.9	1,092
Gemma-3-1B	INT4	0.24	0.42	0.77	15.0	199.4	1,055
LLaMA-3.2-1B	NF4	0.47	0.56	0.83	8.1	187.7	1,164
Gemma-3-1B	NF4	0.23	0.46	0.73	19.5	163.5	1,112
LLaMA-3.2-1B	LoRA	0.55	0.65	0.89	2.7	162.8	2,859
Gemma-3-1B	LoRA	0.25	0.45	0.66	8.9	116.2	2,484
LLaMA-3.2-1B	Perturbed	0.60	0.70	0.89	1.9	159.8	2,858
Gemma-3-1B	Perturbed	0.21	0.40	0.65	8.1	119.4	2,483
LLaMA-3.2-1B	Mem-masked	0.58	0.66	0.87	2.2	158.0	2,858
Gemma-3-1B	Mem-masked	0.22	0.43	0.67	8.1	129.8	2,483
Fine-tuned							
LLaMA-3.2-1B	Pruned + LoRA	0.57	0.66	0.87	2.7	165.2	3,371
Gemma-3-1B	Pruned + LoRA	0.23	0.41	0.63	9.2	163.7	2,879
LLaMA-3.2-1B	INT8 + LoRA	0.75	0.80	0.91	4.1	141.5	1,682
Gemma-3-1B	INT8 + LoRA	0.21	0.37	0.73	20.2	187.9	1,533
LLaMA-3.2-1B	INT4 + LoRA	0.32	0.44	0.83	10.5	169.6	1,092
Gemma-3-1B	INT4 + LoRA	0.32	0.53	0.84	20.1	222.8	1,056
LLaMA-3.2-1B	NF4 + LoRA	0.46	0.56	0.83	12.1	173.4	1,164
Gemma-3-1B	NF4 + LoRA	0.23	0.42	0.75	20.6	172.5	1,113
LLaMA-3.2-1B	Perturbed + LoRA	0.55	0.65	0.87	2.7	165.9	2,859
Gemma-3-1B	Perturbed + LoRA	0.23	0.40	0.61	9.3	164.9	2,484
LLaMA-3.2-1B	Masked + LoRA	0.49	0.61	0.85	3.1	171.7	2,859
Gemma-3-1B	Masked + LoRA	0.21	0.39	0.61	9.4	173.9	2,484

Continued on next page

Model	Approximation Technique	BLEU	ROUGE-L	SBERT	Inf. Time (s)	Out. Size (KB)	Model Size (MB)
Fine-tuned							
LLaMA-3.2-1B	INT8+Pruned+LoRA	0.71	0.77	0.88	4.1	169.9	1,682
Gemma-3-1B	INT8+Pruned+LoRA	0.31	0.53	0.72	20.3	151.9	1,533
LLaMA-3.2-1B	INT4+Pruned+LoRA	0.35	0.46	0.82	10.4	183.6	1,092
Gemma-3-1B	INT4+Pruned+LoRA	0.26	0.45	0.82	20.2	193.3	1,056

Table 4.1: Inference results for LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct on the Dolly-15k dataset (Fine-tuned with Alpaca dataset).

Accuracy. The accuracy characteristics of LLaMA-3.2-1B and Gemma-3-1B differ substantially across approximation methods. LLaMA-3.2-1B exhibits strong improvements from quantization, with `INT8` achieving the highest scores (BLEU = 0.86, ROUGE-L = 0.90, SBERT = 0.94). These results confirm LLaMA's robustness to 8-bit quantization, where weights are stored in INT8 but dequantized back to FP16 before matrix multiplications, resulting in minimal loss of numerical fidelity. Other approximation methods such as pruning, memory masking, and perturbation remain close to the baseline, while more aggressive quantization (`INT4`, `NF4`) leads to visible degradation.

Gemma-3-1B shows a markedly different behavior. Neither `INT8` nor `INT8+LoRA` yields improvements over the baseline, and in several cases performance degrades significantly. Surprisingly, Gemma's best-performing configurations are 4-bit variants (`INT4` and `INT4+LoRA`), which reach SBERT similarities of 0.84–0.85, outperforming all INT8 results. This effect is consistent with prior observations that small transformer models with sharp activation distributions may be more sensitive to uniform INT8 quantization, while structured 4-bit formats (e.g., INT4, NF4) preserve semantic similarity more effectively.

Overall, LLaMA-3.2-1B consistently benefits from INT8 quantization, whereas Gemma-3-1B achieves its strongest semantic fidelity under 4-bit approximations.

Efficiency. LLaMA-3.2-1B also demonstrates favorable efficiency properties under INT8 quantization. The `INT8` model reduces the model size from 2,858 MB to 1,681 MB and achieves the lowest inference latency (1.3 s), confirming the presence of efficient 8-bit kernel support. Although INT4 and NF4 provide higher compression ratios, they introduce substantial latency increases (7.6–12.1 s), largely due to the lack of optimized low-bit kernels for these formats.

Gemma-3-1B displays the opposite latency trend. While INT8 reduces the model size to 1,532 MB, the inference latency increases sharply (15–20 s), and NF4 results in even slower execution (up to 20.6 s). These results indicate that Gemma's

Model	Approximation Technique	ROUGE-L	SBERT	Inference Time(s)	Model Size(MB)
LLaMA-3.2-1B	INT8	0.90	0.94	1.3	1,681
LLaMA-3.2-1B	INT8 + LoRA	0.80	0.91	4.1	1,682
LLaMA-3.2-1B	Pruned	0.69	0.90	2.0	3,370
Gemma-3-1B	INT4 + LoRA	0.53	0.84	20.1	1,056
Gemma-3-1B	INT4+Pruned+LoRA	0.45	0.82	20.2	1,056
Gemma-3-1B	INT4	0.42	0.77	15.0	1,055

Table 4.2: Top-performing approximation techniques for LLaMA-3.2-1B and Gemma-3-1B on the Dolly-15k dataset (Fine-tuned with Alpaca dataset).

architecture interacts less favorably with available INT8/INT4 kernels, leading to severely degraded speed despite reduced model size. Thus, approximation efficiency in Gemma is constrained more by kernel availability and activation-range sensitivity than by parameter precision.

Summary. Across the evaluated approximation techniques, INT8 quantization provides the most favorable accuracy–efficiency trade-off for LLaMA-3.2-1B. It achieves the highest semantic fidelity (SBERT = 0.94), strong n-gram overlap (BLEU = 0.86; ROUGE-L = 0.90), and the lowest inference latency, while also reducing the memory footprint. More aggressive 4-bit formats further compress the model but yield slower inference and reduced accuracy.

Gemma-3-1B exhibits a qualitatively different profile: INT8 offers no measurable accuracy improvement and significantly increases latency. Instead, INT4-based methods, in particular `INT4+LoRA` achievse the strongest semantic alignment and outperform all INT8 variants in terms of accuracy. These findings align with prior reports that small Gemma models are more sensitive to INT8 quantization, while 4-bit structured quantization preserves relative performance more effectively.

The results collectively show that approximation effectiveness is architecture-dependent. LLaMA-3.2-1B is best approximated through INT8 quantization, whereas Gemma-3-1B achieves its best balance of accuracy under INT4+LoRA, albeit with a significant inference-time penalty.

4.2.2 Inference Results for the Dolly-15k Dataset(Fine-tuned with Agent Dataset)

Model	Approximation Technique	BLEU	ROUGE-L	SBERT	Inf. Time (s)	Out. Size (KB)	Model Size (MB)
Baseline							
LLaMA-3.2-1B	——	0.54	0.64	0.86	2.2	152.4	2,858
Gemma-3-1B	——	0.29	0.49	0.75	7.7	133.6	2,483
Fine-tuned							
LLaMA-3.2-1B	Pruned	0.59	0.68	0.88	2.5	157.4	3,370
Gemma-3-1B	Pruned	0.27	0.48	0.73	7.7	131.3	2,878
LLaMA-3.2-1B	INT8	0.53	0.62	0.83	5.5	174.5	1,681
Gemma-3-1B	INT8	0.27	0.47	0.74	14.7	129.7	1,532
LLaMA-3.2-1B	INT4	0.34	0.44	0.77	7.7	190.9	1,092
Gemma-3-1B	INT4	0.28	0.47	0.85	14.8	187.9	1,055
LLaMA-3.2-1B	NF4	0.42	0.53	0.83	9.4	170.4	1,164
Gemma-3-1B	NF4	0.22	0.44	0.69	18.6	176.9	1,112
LLaMA-3.2-1B	LoRA	0.56	0.66	0.86	2.7	160.5	2,859
Gemma-3-1B	LoRA	0.26	0.45	0.68	8.5	129.0	2,484
LLaMA-3.2-1B	Perturbed	0.56	0.66	0.89	2.5	158.3	2,858
Gemma-3-1B	Perturbed	0.26	0.46	0.73	7.7	130.5	2,483
LLaMA-3.2-1B	Mem-masked	0.54	0.63	0.87	2.8	161.2	2,858
Gemma-3-1B	Mem-masked	0.26	0.45	0.74	8.3	139.5	2,483
Fine-tuned							
LLaMA-3.2-1B	Pruned + LoRA	0.54	0.63	0.83	2.8	175.6	3,371
Gemma-3-1B	Pruned + LoRA	0.29	0.46	0.68	8.8	154.0	2,879
LLaMA-3.2-1B	INT8 + LoRA	0.55	0.63	0.85	8.0	175.9	1,682
Gemma-3-1B	INT8 + LoRA	0.30	0.47	0.72	19.6	146.1	1,533
LLaMA-3.2-1B	INT4 + LoRA	0.30	0.42	0.77	10.8	193.3	1,092
Gemma-3-1B	INT4 + LoRA	0.26	0.45	0.83	19.7	176.6	1,056
LLaMA-3.2-1B	NF4 + LoRA	0.43	0.54	0.85	12.0	153.9	1,164
Gemma-3-1B	NF4 + LoRA	0.25	0.43	0.73	26.4	161.5	1,113
LLaMA-3.2-1B	Perturbed + LoRA	0.50	0.58	0.82	3.4	182.3	2,859
Gemma-3-1B	Perturbed + LoRA	0.32	0.52	0.74	8.8	157.8	2,484
LLaMA-3.2-1B	Masked + LoRA	0.49	0.58	0.82	3.3	178.8	2,859
Gemma-3-1B	Masked + LoRA	0.24	0.47	0.73	8.9	163.2	2,484
Fine-tuned							
LLaMA-3.2-1B	INT8+Pruned+LoRA	0.59	0.67	0.86	7.5	159.5	1,682

Continued on next page

Model	Approximation Technique	BLEU	ROUGE-L	SBERT	Inf. Time (s)	Out. Size (KB)	Model Size (MB)
Gemma-3-1B	INT8+Pruned+LoRA	0.25	0.43	0.69	19.3	164.4	1,533
LLaMA-3.2-1B	INT4+Pruned+LoRA	0.31	0.42	0.84	10.6	165.8	1,092
Gemma-3-1B	INT4+Pruned+LoRA	0.31	0.48	0.85	19.6	220.4	1,056

Table 4.3: Inference results for LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct on the Dolly-15k dataset (Fine-tuned with Agent dataset).

Accuracy. The accuracy trends for the Agent-fine-tuned Dolly-15k dataset differ markedly from those obtained using the Alpaca dataset. LLaMA-3.2-1B shows only modest accuracy gains across most approximation methods. Perturbed remains the strongest performer among the simple approximations (BLEU = 0.56, ROUGE-L = 0.66, SBERT = 0.89), while the Pruning method preserves semantic similarity close to the baseline (SBERT = 0.88). However, in contrast to the Alpaca-trained setting, the `INT8` configuration does not improve accuracy on LLaMA. Its scores (BLEU = 0.53, SBERT = 0.83) fall below the baseline, indicating that quantization interacts less favorably fine-tuning with the Agent dataset.

Gemma-3-1B displays a completely different behavior. As in the Alpaca case, 8-bit quantization is ineffective. However, 4-bit variants significantly outperform all other approximations. `INT4` and `INT4+Pruned+LoRA` achieve the highest semantic similarity (SBERT = 0.85), and `INT4+LoRA` also performs strongly (SBERT = 0.83). Thus, Gemma again shows a preference for structured 4-bit quantization, maintaining or improving semantic quality while 8-bit methods remain suboptimal.

Efficiency. From an efficiency perspective, LLaMA and Gemma behave very differently. LLaMA's `INT8` and `INT8+LoRA` methods produce substantial increases in inference latency (5.5–8.0 s), reflecting the absence of kernel-level optimizations for 8-bit execution in the Agent-trained setting. Meanwhile, pruning and memory masking produce latencies similar to the baseline (2.5–3.0 s) while providing moderate accuracy benefits.

Gemma-3-1B experiences severe latency slowdowns across nearly all approximation methods. All INT8 and INT4 variants exceed 14–20 s per inference, and NF4-based methods are even slower (up to 26.4 s). Only pruning and memory masking maintain baseline-level speed (7–9 s), but these provide only mild accuracy gains. Thus, Gemma lacks an approximation technique that simultaneously improves accuracy and reduces latency.

Summary. The Agent-fine-tuned results reveal two distinct approximation profiles. For LLaMA-3.2-1B, pruning and hybrid variants such as `INT8+Pruned+LoRA`

Model	Approximation Technique	ROUGE-L	SBERT	Inference Time(s)	Model Size(MB)
LLaMA-3.2-1B	Perturbed	0.66	0.89	2.5	2,858
LLaMA-3.2-1B	Pruned	0.68	0.88	2.5	3,370
LLaMA-3.2-1B	INT8 + Pruned + LoRA	0.67	0.86	7.5	1,682
Gemma-3-1B	INT4	0.47	0.85	14.8	1,055
Gemma-3-1B	INT4 + Pruned + LoRA	0.48	0.85	19.6	1,056
Gemma-3-1B	INT4 + LoRA	0.45	0.83	19.7	1,056

Table 4.4: Top-performing approximation techniques for LLaMA-3.2-1B and Gemma-3-1B on the Dolly-15k dataset (Fine-tuned with Agent dataset).

provide the best accuracy while maintaining reasonable inference times. Unlike the Alpaca dataset, INT8 quantization does not enhance accuracy and frequently increases latency. In contrast, Gemma-3-1B achieves its strongest accuracy under 4-bit approximations, despite their substantial computational overhead. INT4- and INT4+LoRA-based methods consistently reach the highest SBERT similarity (0.85) among all Gemma variants, confirming that Gemma's architecture is more compatible with structured 4-bit quantization than with 8-bit precision.

4.2.3 Inference Results for the AgentInstruct Dataset(Fine-tuned with Alpaca Dataset)

Model	Approximation Technique	BLEU	ROUGE-L	SBERT	Inf. Time (s)	Out. Size (KB)	Model Size (MB)
Baseline							
LLaMA-3.2-1B	——	0.59	0.68	0.85	2.7	34.3	2,858
Gemma-3-1B	——	0.26	0.63	0.75	8.1	36.7	2,483
Fine-tuned							
LLaMA-3.2-1B	Pruned	0.59	0.67	0.86	2.4	34.6	3,370
Gemma-3-1B	Pruned	0.22	0.54	0.71	8.3	35.4	2,878
LLaMA-3.2-1B	INT8	0.93	0.95	0.98	0.57	29.8	1,681
Gemma-3-1B	INT8	0.30	0.58	0.82	15.4	41.9	1,532
LLaMA-3.2-1B	INT4	0.23	0.37	0.71	7.5	39.8	1,092
Gemma-3-1B	INT4	0.19	0.46	0.87	15.3	41.5	1,055
LLaMA-3.2-1B	NF4	0.49	0.61	0.75	8.2	37.3	1,164
Gemma-3-1B	NF4	0.17	0.39	0.71	19.2	37.9	1,112

Continued on next page

Model	Approximation Technique	BLEU	ROUGE-L	SBERT	Inf. Time (s)	Out. Size (KB)	Model Size (MB)
LLaMA-3.2-1B	LoRA	0.57	0.68	0.82	3.5	33.2	2,859
Gemma-3-1B	LoRA	0.25	0.56	0.70	9.2	35.3	2,484
LLaMA-3.2-1B	Perturbed	0.48	0.58	0.75	2.8	34.8	2,858
Gemma-3-1B	Perturbed	0.22	0.53	0.73	8.4	36.6	2,483
LLaMA-3.2-1B	Mem-masked	0.48	0.60	0.75	3.3	33.4	2,858
Gemma-3-1B	Mem-masked	0.22	0.48	0.72	8.3	38.8	2,483
Fine-tuned							
LLaMA-3.2-1B	Pruned + LoRA	0.44	0.57	0.75	3.8	34.9	3,371
Gemma-3-1B	Pruned + LoRA	0.25	0.58	0.73	9.3	35.2	2,879
LLaMA-3.2-1B	INT8 + LoRA	0.86	0.88	0.95	1.8	31.6	1,682
Gemma-3-1B	INT8 + LoRA	0.26	0.55	0.82	20.3	38.5	1,533
LLaMA-3.2-1B	INT4 + LoRA	0.33	0.47	0.83	10.0	42.3	1,092
Gemma-3-1B	INT4 + LoRA	0.19	0.41	0.84	20.3	40.2	1,056
LLaMA-3.2-1B	NF4 + LoRA	0.51	0.59	0.83	11.5	39.3	1,164
Gemma-3-1B	NF4 + LoRA	0.20	0.57	0.80	26.2	41.4	1,113
LLaMA-3.2-1B	Perturbed + LoRA	0.44	0.57	0.82	3.5	33.8	2,859
Gemma-3-1B	Perturbed + LoRA	0.31	0.64	0.73	9.3	34.9	2,484
LLaMA-3.2-1B	Masked + LoRA	0.47	0.61	0.81	3.5	32.8	2,859
Gemma-3-1B	Masked + LoRA	0.23	0.53	0.75	9.3	39.5	2,484
Fine-tuned							
LLaMA-3.2-1B	INT8+Pruned+LoRA	0.72	0.77	0.88	4.9	35.3	1,682
Gemma-3-1B	INT8+Pruned+LoRA	0.21	0.54	0.73	20.0	37.4	1,533
LLaMA-3.2-1B	INT4+Pruned+LoRA	0.37	0.48	0.80	9.8	44.0	1,092
Gemma-3-1B	INT4+Pruned+LoRA	0.22	0.40	0.87	20.0	44.4	1,056

Table 4.5: Inference results for LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct on the AgentInstruct dataset (Fine-tuned with Alpaca dataset).

Accuracy. For the AgentInstruct dataset, the strongest pattern across all evaluations is the dominant performance of INT8-based methods for LLaMA-3.2-1B. The plain `INT8` configuration achieves exceptionally high accuracy (BLEU = 0.93, ROUGE-L = 0.95, SBERT = 0.98), representing the best scores among all tested techniques and datasets. This behavior mirrors the Alpaca-trained results and confirms that LLaMA's architecture benefits substantially from 8-bit quantization, preserving semantic similarity while reducing numerical noise through FP16 dequantized computation.

LLaMA's `INT8+LoRA` variant also performs well (BLEU = 0.86, SBERT = 0.95), although it remains slightly below the pure INT8 model. Other approximation

Model	Approximation Technique	ROUGE-L	SBERT	Inference Time(s)	Model Size(MB)
LLaMA-3.2-1B	INT8	0.95	0.98	0.57	1,681
LLaMA-3.2-1B	INT8 + LoRA	0.88	0.95	1.8	1,682
LLaMA-3.2-1B	INT8 + Pruned + LoRA	0.77	0.88	4.9	1,682
Gemma-3-1B	INT4	0.46	0.87	15.3	1,055
Gemma-3-1B	INT4 + Pruned + LoRA	0.40	0.87	20.0	1,056
Gemma-3-1B	INT4 + LoRA	0.41	0.84	20.3	1,056

Table 4.6: Top-performing approximation techniques for LLaMA-3.2-1B and Gemma-3-1B on the AgentInstruct dataset (Fine-tuned with Alpaca dataset).

techniques including pruning, perturbation, memory masking, and all 4-bit or NF4 methods provide only limited accuracy improvements and often degrade performance. The weakest methods for LLaMA are the INT4-based variants, which fall significantly in both token-level and semantic metrics (BLEU = 0.23–0.37).

Gemma-3-1B again shows a contrasting behavior. As in previous datasets, INT8 remains ineffective, providing only moderate improvements (SBERT = 0.82) and substantially lower BLEU/ROUGE-L than LLaMA. Both `INT4` and `INT4+Pruned+LoRA` achieve SBERT values of 0.86 and 0.87, respectively the highest semantic similarity across all Gemma configurations. This reinforces the observation that Gemma's small-parameter architecture responds better to 4-bit structured quantization than to 8-bit uniform quantization.

Efficiency. The efficiency profile sharply distinguishes the two architectures. For LLaMA-3.2-1B, INT8 delivers substantial latency reductions; the `INT8` model achieves the lowest inference time in all experiments (0.57 s), representing more than a 4× improvement over the baseline. Even with LoRA applied, INT8 remains efficient (1.8 s), significantly outperforming heavier methods such as INT4 (7–10 s) or NF4 (8–12 s). Pruning and perturbation remain lightweight, maintaining near-baseline runtimes (2.4–3.5 s).

Gemma-3-1B again lacks an efficient approximation method. All quantized variants (INT8, INT4, NF4) suffer from severe latency penalties (15–26 s), consistent with the lack of optimized kernels. Only pruning and memory masking preserve baseline-level latency, but they do not offer meaningful accuracy gains in this dataset. Thus, no approximation technique simultaneously improves Gemma's accuracy and efficiency.

Summary. The AgentInstruct evaluations confirm a strong dataset- and model-dependent divide between approximation behaviors. For LLaMA-3.2-1B, INT8

is clearly the optimal choice. It simultaneously achieves the highest accuracy (SBERT = 0.98) and lowest inference latency (0.57 s), providing the best performance–efficiency trade-off among all tested methods. Combining INT8 with LoRA produces slightly lower accuracy but remains highly competitive. Other approximations provide only modest improvements or degrade performance.

Gemma-3-1B, in contrast, achieves its best semantic similarity under 4-bit structured quantization (`INT4` and `INT4+Pruned+LoRA`, SBERT = 0.86–0.87), confirming the model's preference for lower-precision quantization over 8-bit methods. However, these accuracy gains come with significant inference slowdowns, making them impractical for efficiency-oriented scenarios. Overall, the results highlight the importance of architectural compatibility: INT8 is ideal for LLaMA, while Gemma benefits more from 4-bit quantization, albeit at high computational cost.

Chapter 5

Conclusion

The research investigated the integration of approximate computing techniques into instruction-tuned LLMs, focusing on two architectures of comparable scale: LLaMA-3.2-1B-Instruct and Gemma-3-1B-Instruct. The objective was to determine how quantization, stochastic perturbation, pruning, memory masking, and LoRA influence model accuracy, semantic similarity, and computational efficiency across multiple instruction-following datasets. As modern transformer models continue to grow, such approximation strategies offer a viable path toward efficient deployment on constrained hardware.

A modular approximation framework was developed to allow techniques to be applied individually and in combination during fine-tuning. The experimental pipeline incorporated simulated low-precision arithmetic, controlled stochasticity, and structured sparsity, enabling a unified analysis of their effects on numerical stability, representational robustness, and decoding behaviour. Evaluation metrics included token-overlap scores, which are BLEU, ROUGE-L and F1, semantic similarity measured via SBERT cosine similarity, and efficiency metrics such as inference latency, output size, and model footprint.

Experiments were carried out using cross-dataset evaluation schemes rather than training and testing on the same corpus. In the primary configuration, the models were fine-tuned on the Alpaca dataset and evaluated on Dolly-15k, a setting denoted as *Alpaca–Dolly*. This setup intentionally exposes the models to a distribution shift: Alpaca provides highly structured, template-like instruction–response pairs, while Dolly-15k contains more diverse, conversational, and open-ended instructions. As a result, approximation errors that remain hidden under Alpaca-style prompts become more pronounced during evaluation on Dolly, revealing how robust each technique is to linguistic variability.

The alternative configurations *Alpaca–AgentInstruct* and *AgentInstruct–Dolly* follow the same cross-dataset principle. AgentInstruct introduces shorter, more

procedural prompts, which amplify the effect of even small approximation distortions. These cross-dataset setups consistently showed that the impact of reduced precision and stochastic computation is tightly coupled to the mismatch between training and evaluation distributions. Thus, approximation performance cannot be interpreted independently of the dataset pair, as different forms of linguistic structure expose different failure modes of each technique.

Across all experiments, LLaMA-3.2-1B-Instruct demonstrated substantially higher tolerance to quantization, pruning, and stochastic perturbation than Gemma-3-1B-Instruct. LLaMA repeatedly achieved strong improvements under INT8 quantization, often reducing latency and model size while simultaneously increasing both token-overlap and SBERT scores. In contrast, Gemma exhibited pronounced degradation under INT8, inference became an order of magnitude slower, and accuracy stagnated or decreased relative to baseline. These findings align with external observations that Gemma's activation distribution and normalization design make it more sensitive to mid-precision quantizers.

More aggressive low-bit quantizers produced divergent outcomes between the two models. For LLaMA, INT4 and NF4 caused substantial losses in BLEU, ROUGE-L, and SBERT similarity, accompanied by large increases in latency due to simulated dequantization. Gemma, however, improved under INT4 and NF4 relative to its INT8 variant, a behaviour consistent with prior reports that small Gemma models sometimes favour lower-bit quantizers due to reduced systematic quantization bias. Although these improvements did not surpass LLaMA's INT8 performance, they reveal architecture-dependent interactions between quantization level and representational robustness.

Pruning and stochastic perturbation produced interpretable patterns across datasets. On LLaMA, pruning preserved accuracy while reducing latency, and stochastic perturbation provided mild regularization benefits, yielding balanced improvements in both overlap-based and semantic metrics. These techniques, however, had limited or inconsistent benefits for Gemma, which frequently exhibited performance drops under structural modification or noise injection. Memory masking produced minimal changes in either model, suggesting that more adaptive masking strategies may be necessary for meaningful impact.

Hybrid techniques produced the most compelling trade-offs. For LLaMA, combinations such as Pruned + INT8 or INT8 + LoRA consistently provided strong accuracy with moderate latency and compact outputs. On Gemma, hybrid techniques did not fully compensate for INT8 sensitivity, even with LoRA or pruning, accuracy remained lower than baseline LLaMA performance, and inference remained significantly slower. These findings indicate that approximation gains depend on architectural resilience, and that techniques effective for one model family do not generalize universally.

Efficiency patterns further highlighted fundamental differences between the

models. LLaMA's INT8 variants achieved the lowest latencies among all tested configurations, sometimes approaching 0.6 s per inference. In stark contrast, Gemma's INT8 variants displayed the highest latencies of any model–often exceeding 15–20 s per inference, suggesting limited compatibility with simulated 8-bit quantization workflows.

Overall, the results demonstrate that approximate computing techniques can be applied effectively to transformer-based LLMs, but their success is heavily architecture-dependent. LLaMA-3.2-1B-Instruct consistently benefited from INT8 quantization, pruning, and stochastic perturbation, achieving strong accuracy and substantial latency reductions. Gemma-3-1B-Instruct, despite its similar parameter count, showed markedly lower robustness and significantly poorer performance under most approximation strategies. These differences highlight the importance of model–technique compatibility and emphasize that approximation cannot be assumed to generalize across architectures.

Several limitations remain. Experiments were conducted under simulated low-precision arithmetic, without hardware-optimized INT8 or INT4 kernels. Real quantization-aware backends may yield different latency and accuracy behaviours. The analysis was constrained to $\approx$ 1B-parameter models; larger LLaMA variants may exhibit increased redundancy and therefore greater tolerance to approximation. Finally, although SBERT provided an improved measure of semantic fidelity, future work should incorporate additional embedding-based metrics, human evaluation, and energy/throughput profiling to fully characterize approximation effects.

In conclusion, the study provides a systematic analysis of how quantization, pruning, stochasticity, and low-rank adaptation interact with model architecture and task structure in small instruction-tuned LLMs. The findings highlight INT8 and stochastic methods as the most effective techniques for LLaMA-3.2-1B-Instruct, while demonstrating that Gemma-3-1B-Instruct is significantly more fragile under approximation. These insights contribute to a deeper understanding of the trade-offs involved in deploying efficient LLMs and reinforce the potential of approximate computing as a cornerstone of sustainable and scalable AI development.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention is All you Need». In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf (cit. on p. 1).

[2] Tom Brown et al. «Language Models are Few-Shot Learners». In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf (cit. on pp. 1, 2).

[3] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361 (cit. on pp. 1, 2).

[4] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. *A Survey of Quantization Methods for Efficient Neural Network Inference*. 2021. arXiv: 2103.13630 [cs.CV]. URL: https://arxiv.org/abs/2103.13630 (cit. on pp. 1, 2, 8, 10–13, 37, 38).

[5] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV]. URL: https://arxiv.org/abs/1510.00149 (cit. on pp. 1, 2, 14, 16, 17, 22).

[6] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685 (cit. on pp. 1, 2, 18, 19, 40).

[7] Xiang Gao, Jiaxin Zhang, Lalla Mouatadid, and Kamalika Das. *SPUQ: Perturbation-Based Uncertainty Quantification for Large Language Models*. 2024. arXiv: 2403.02509 [cs.CL]. URL: https://arxiv.org/abs/2403.02509 (cit. on pp. 1, 20, 21).

[8] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. *QSGD: Communication-Efficient SGD via Gradient Quantization and Encoding*. 2017. arXiv: 1610.02132 [cs.LG]. URL: https://arxiv.org/abs/1610.02132 (cit. on pp. 1, 2, 23).

[9] An Yang, Kai Liu, Jing Liu, Yajuan Lyu, and Sujian Li. *Adaptations of ROUGE and BLEU to Better Evaluate Machine Reading Comprehension Task*. 2018. arXiv: 1806.03578 [cs.CL]. URL: https://arxiv.org/abs/1806.03578 (cit. on pp. 2, 24, 25).

[10] Gabriel Alon and Michael Kamfonas. *Detecting Language Model Attacks with Perplexity*. 2023. arXiv: 2308.14132 [cs.CL]. URL: https://arxiv.org/abs/2308.14132 (cit. on pp. 2, 24, 26).

[11] Reda Yacouby and Dustin Axman. «Probabilistic Extension of Precision, Recall, and F1 Score for More Thorough Evaluation of Classification Models». In: *Proceedings of the First Workshop on Evaluation and Comparison of NLP Systems*. Ed. by Steffen Eger, Yang Gao, Maxime Peyrard, Wei Zhao, and Eduard Hovy. Online: Association for Computational Linguistics, Nov. 2020, pp. 79–91. DOI: 10.18653/v1/2020.eval4nlp-1.9. URL: https://aclanthology.org/2020.eval4nlp-1.9/ (cit. on pp. 2, 24, 25).

[12] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. 2017. arXiv: 1712.05877 [cs.LG]. URL: https://arxiv.org/abs/1712.05877 (cit. on pp. 2, 5–7, 11, 13, 14, 37).

[13] Song Han, Jeff Pool, John Tran, and William J. Dally. *Learning both Weights and Connections for Efficient Neural Networks*. 2015. arXiv: 1506.02626 [cs.NE]. URL: https://arxiv.org/abs/1506.02626 (cit. on p. 2).

[14] Yuchen Zeng and Kangwook Lee. *The Expressive Power of Low-Rank Adaptation*. 2024. arXiv: 2310.17513 [cs.LG]. URL: https://arxiv.org/abs/2310.17513 (cit. on pp. 2, 18, 19).

[15] Justin Zhao et al. *LoRA Land: 310 Fine-tuned LLMs that Rival GPT-4, A Technical Report*. 2024. arXiv: 2405.00732 [cs.CL]. URL: https://arxiv.org/abs/2405.00732 (cit. on pp. 2, 19).

[16] Yoshua Bengio. *Estimating or Propagating Gradients Through Stochastic Neurons*. 2013. arXiv: 1305.2982 [cs.LG]. URL: https://arxiv.org/abs/1305.2982 (cit. on pp. 2, 8, 11).

[17] Junkai Chen, Zhenhao Li, Xing Hu, and Xin Xia. *NLPerturbator: Studying the Robustness of Code LLMs to Natural Language Variations*. 2024. arXiv: 2406.19783 [cs.SE]. URL: https://arxiv.org/abs/2406.19783 (cit. on p. 2).

[18] Onem Chinova, Victor Stratton, Dominic Kingswell, Evelyn Whitmore, and Zephyr Coleridge. *Stochastic Token Permutation in Large Language Models for Controlled Contextual Perturbation*. OSF Preprints. 2025. URL: https://osf.io/ (cit. on p. 2).

[19] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. 2023. arXiv: 2210.17323 [cs.LG]. URL: https://arxiv.org/abs/2210.17323 (cit. on pp. 7, 9, 12, 38, 40).

[20] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. *PACT: Parameterized Clipping Activation for Quantized Neural Networks*. 2018. arXiv: 1805.06085 [cs.CV]. URL: https://arxiv.org/abs/1805.06085 (cit. on p. 8).

[21] Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. *Learned Step Size Quantization*. 2020. arXiv: 1902.08153 [cs.LG]. URL: https://arxiv.org/abs/1902.08153 (cit. on p. 8).

[22] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. *A White Paper on Neural Network Quantization*. 2021. arXiv: 2106.08295 [cs.LG]. URL: https://arxiv.org/abs/2106.08295 (cit. on p. 9).

[23] Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. *Post-training 4-bit quantization of convolution networks for rapid-deployment*. 2019. arXiv: 1810.05723 [cs.CV]. URL: https://arxiv.org/abs/1810.05723 (cit. on p. 9).

[24] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. *Low-bit Quantization of Neural Networks for Efficient Inference*. 2019. arXiv: 1902.06822 [cs.LG]. URL: https://arxiv.org/abs/1902.06822 (cit. on p. 9).

[25] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhiru Zhang. *Improving Neural Network Quantization without Retraining using Outlier Channel Splitting*. 2019. arXiv: 1901.09504 [cs.LG]. URL: https://arxiv.org/abs/1901.09504 (cit. on p. 9).

[26] Zhen Dong, Zhewei Yao, Yaohui Cai, Daiyaan Arfeen, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. *HAWQ-V2: Hessian Aware trace-Weighted Quantization of Neural Networks*. 2019. arXiv: 1911.03852 [cs.CV]. URL: https://arxiv.org/abs/1911.03852 (cit. on p. 14).

[27] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: 1503.02531 [stat.ML]. URL: https://arxiv.org/abs/1503.02531 (cit. on p. 14).

[28] Can Chen, Xi Chen, Chen Ma, Zixuan Liu, and Xue Liu. *Gradient-based Bi-level Optimization for Deep Learning: A Survey*. 2023. arXiv: 2207.11719 [cs.LG]. URL: https://arxiv.org/abs/2207.11719 (cit. on p. 14).

[29] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: 2305.14314 [cs.LG]. URL: https://arxiv.org/abs/2305.14314 (cit. on pp. 15, 38, 39).

[30] Yann LeCun, John Denker, and Sara Solla. «Optimal Brain Damage». In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky. Vol. 2. Morgan-Kaufmann, 1989. URL: https://proceedings.neurips.cc/paper_files/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf (cit. on p. 16).

[31] Paul Michel, Omer Levy, and Graham Neubig. *Are Sixteen Heads Really Better than One?* 2019. arXiv: 1905.10650 [cs.CL]. URL: https://arxiv.org/abs/1905.10650 (cit. on p. 16).

[32] Kyuhong Shim, Iksoo Choi, Wonyong Sung, and Jungwook Choi. *Layer-wise Pruning of Transformer Attention Heads for Efficient Language Modeling*. 2021. arXiv: 2110.03252 [cs.CL]. URL: https://arxiv.org/abs/2110.03252 (cit. on p. 17).

[33] Jonathan Frankle and Michael Carbin. *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks*. 2019. arXiv: 1803.03635 [cs.LG]. URL: https://arxiv.org/abs/1803.03635 (cit. on p. 17).

[34] Jiajun Wang. «Research on pruning optimization techniques for neural networks». In: *Applied and Computational Engineering* 19 (Oct. 2023), pp. 152–158. DOI: 10.54254/2755-2721/19/20231025 (cit. on p. 17).

[35] Yue Zhang, Leyang Cui, Wei Bi, and Shuming Shi. *Alleviating Hallucinations of Large Language Models through Induced Hallucinations*. 2024. arXiv: 2312.15710 [cs.CL]. URL: https://arxiv.org/abs/2312.15710 (cit. on p. 20).

[36] Eyke Hüllermeier and Willem Waegeman. «Aleatoric and epistemic uncertainty in machine learning: an introduction to concepts and methods». In: *Machine Learning* 110.3 (Mar. 2021), pp. 457–506. ISSN: 1573-0565. DOI: 10.1007/s10994-021-05946-3. URL: http://dx.doi.org/10.1007/s10994-021-05946-3 (cit. on p. 20).

[37] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. «Dropout: A Simple Way to Prevent Neural Networks from Overfitting». In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html (cit. on p. 22).

[38] Yarin Gal and Zoubin Ghahramani. «Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning». In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1050–1059. URL: https://proceedings.mlr.press/v48/gal16.html (cit. on p. 22).

[39] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. *LLM in a flash: Efficient Large Language Model Inference with Limited Memory*. 2024. arXiv: 2312.11514 [cs.CL]. URL: https://arxiv.org/abs/2312.11514 (cit. on p. 23).

[40] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. *Training Deep Nets with Sublinear Memory Cost*. 2016. arXiv: 1604.06174 [cs.LG]. URL: https://arxiv.org/abs/1604.06174 (cit. on p. 23).

[41] Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. *The Reversible Residual Network: Backpropagation Without Storing Activations*. 2017. arXiv: 1707.04585 [cs.CV]. URL: https://arxiv.org/abs/1707.04585 (cit. on p. 23).

[42] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. «Regularization of Neural Networks using DropConnect». In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 1058–1066. URL: https://proceedings.mlr.press/v28/wan13.html (cit. on p. 23).

[43] Nils Reimers and Iryna Gurevych. *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. 2019. arXiv: 1908.10084 [cs.CL]. URL: https://arxiv.org/abs/1908.10084 (cit. on p. 25).

[44] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. *Stanford Alpaca: An Instruction-following LLaMA model*. https://github.com/tatsu-lab/stanford_alpaca. 2023 (cit. on p. 31).

[45] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. *Self-Instruct: Aligning Language Models with Self-Generated Instructions*. 2023. arXiv: 2212.10560 [cs.CL]. URL: https://arxiv.org/abs/2212.10560 (cit. on p. 31).

[46] Mike Conover et al. *Free Dolly: Introducing the World's First Truly Open Instruction-Tuned LLM*. 2023. URL: https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm (visited on 06/30/2023) (cit. on p. 32).

[47] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL]. URL: https://arxiv.org/abs/2203.02155 (cit. on pp. 32, 35).

[48] Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. *AgentTuning: Enabling Generalized Agent Abilities for LLMs*. 2023. arXiv: 2310.12823 [cs.CL] (cit. on p. 33).

[49] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: https://arxiv.org/abs/2302.13971 (cit. on p. 34).

[50] Noam Shazeer. *Fast Transformer Decoding: One Write-Head is All You Need*. 2019. arXiv: 1911.02150 [cs.NE]. URL: https://arxiv.org/abs/1911.02150 (cit. on p. 34).

[51] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: https://arxiv.org/abs/2104.09864 (cit. on p. 35).

[52] Noam Shazeer. *GLU Variants Improve Transformer*. 2020. arXiv: 2002.05202 [cs.LG]. URL: https://arxiv.org/abs/2002.05202 (cit. on p. 35).

[53] Gemma Team et al. *Gemma 3 Technical Report*. 2025. arXiv: 2503.19786 [cs.CL]. URL: https://arxiv.org/abs/2503.19786 (cit. on pp. 35, 36).