Optimizing Large Language Models: Algorithmic Advancements and Model Design Strategies

FENGXIANG BIE

Master of Philosophy(Engineering)

Supervisor: Associate Professor Shuaiwen Leon Song
Associate Supervisor: Associate Professor Qiang Tang

A thesis submitted in fulfilment of
the requirements for the degree of
Master of Philosophy

School of Information Technology
Faculty of Engineering
The University of Sydney
Australia

11 May 2026

Abstract

Large Language models have achieved remarkable performance across diverse tasks, but face two critical deployment challenges: (1) the key-value (KV) cache memory bottleneck that limits model deployment in resource-constrained environments, and (2) the sequential autoregressive generation latency that reduces inference throughput and user experience.

This thesis presents two complementary contributions addressing these distinct challenges. First, CARE (Covariance-Aware and Rank-Enhanced) tackles the KV-cache memory bottleneck by converting pretrained Grouped Query Attention (GQA) models into memory-efficient Multi-Head Latent Attention (MLA) architectures. Unlike naive SVD approaches that ignore activation patterns, CARE introduces activation-preserving factorization using covariance-weighted SVD and adaptive rank allocation via water-filling algorithms. Second, Infinigram-based speculative decoding addresses inference latency by leveraging large-scale n-gram statistics to predict multiple tokens in parallel, achieving significant speedup through CPU-optimized data structures and confidence-based acceptance strategies.

Experimental results on Llama-3.1-8B demonstrate that CARE achieves up to 331% relative improvement in zero-shot accuracy over baseline conversion methods while maintaining identical KV-cache footprint. Post-conversion healing fully recovers original model performance with minimal fine-tuning. Infinigram delivers significant inference speedups across various sequence lengths and batch sizes, with acceptance rates improving for longer context matches and higher-frequency patterns.

This work contributes novel methodologies combining model design strategies and algorithmic advancements for efficient large generative model deployment, providing practical solutions to key memory and computational challenges without compromising model capabilities.

Acknowledgements

Statement of Originality

This is to certify that the content of this thesis is my own work. This thesis has not been submitted for any other degree or purpose.
I certify that the intellectual content of this thesis is the product of my own work, and that all assistance received in preparing this thesis and all sources have been acknowledged.

Author Attribution Statement

Chapter 2 contains material previously published in my written survey [1]. This material comprises Section 1.1, Section 2.1, and Table 2.1. I am the first author of this paper, and I contributed to the writing of the whole paper.
Chapter 3 of this thesis contains material published at the International Conference on Learning Representations (ICLR) 2026 [2]. For this work, I co-designed the study and contributed to the motivating experiments: I identified the performance degradation issue caused by uniform rank allocation through uniform-SVD experiments on DeepSeek-V2-Lite, reported in Figure 3.1. I also co-designed the CARE methodology, including the covariance computation implementation and the water-filling rank selection strategy; the resulting rank-allocation behavior and its effect are analyzed in Table 3.5. I ran the pre-healing experiments reported in Table 3.1 and Table 3.2, and I conducted the healing experiments for the CARE method reported in Table 3.3.
Chapter 4 of this thesis is my own work. I independently designed the study, implemented the methodology, conducted the experiments, and analyzed the results presented in this chapter.

In addition to the authorship attribution statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

Use of Generative AI

During the preparation of this thesis, the author used Claude for the purposes of text enhancement. The use of this generative AI tool includes paraphrasing, writing, and sentence structure improvement. The author confirms that where text was modified by generative AI, the content was reviewed for possible errors, inaccuracies, and bias.

The author takes full responsibility for the submitted thesis and ensures the work is their own and has used generative AI within the parameters of use.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 AI Generated Content and Large Language Models

In recent years, AI Generated Content (AIGC) has emerged as a transformative technology that fundamentally reshapes how we create, process, and interact with information. AIGC encompasses the use of artificial intelligence to generate human-like content across multiple modalities, with large language models (LLMs) serving as the cornerstone of this revolution. The remarkable capabilities of modern models such as ChatGPT [3, 4], GPT-4 [5], Claude [6], and Llama [7, 8] demonstrate unprecedented proficiency in understanding complex queries and generating coherent, contextually appropriate responses across diverse domains.

The impact of LLMs extends beyond text generation, influencing applications including conversational systems [9], code generation [10], scientific writing [11], tutoring [12], and professional content creation. These models exhibit emergent abilities such as few-shot learning [13], chain-of-thought reasoning [14], and multi-step problem solving. With the combination of transformer architectures [15] and massive training datasets, LLMs are capable of zero-shot and few-shot generalization, enabling adaptation to novel tasks without explicit retraining.

A variety of architectural paradigms exist, including encoder-only models such as BERT [16], encoder–decoder frameworks like T5 [17], and decoder-only autoregressive models like GPT [18]. Scaling trends across these architectures have resulted in exponential growth in model size, from millions to hundreds of billions of parameters [19]. This scaling underpins emergent capabilities [20] and has positioned LLMs as foundational components of modern

AI systems, driving advances across industries ranging from automated assistants [21] to scientific research acceleration [22].

Despite their successes, current LLMs face critical barriers to practical deployment. Their massive parameter counts and reliance on large key-value (KV) caches impose prohibitive memory requirements, particularly in resource-constrained environments [23]. Moreover, the sequential nature of autoregressive text generation introduces inference latency that limits throughput and user experience [24]. Addressing these challenges requires innovations in both architectural compression and inference acceleration. This thesis contributes two complementary approaches: CARE, a method for KV-cache–efficient attention conversion, and Infinigram-based speculative decoding for parallelized inference. Together, these methods highlight a path forward for improving memory efficiency and inference speed without sacrificing model performance.

CHAPTER 2

Literature review

2.1 Transformer models

Transformer models represent a powerful class of deep learning models designed to process and generate sequential data, such as text tokens [15] and image tokens [25]. Transformer architecture tackles the shortcomings of earlier recurrent neural networks [26] and convolutional networks [27] which had difficulty retaining information across large context windows. This architecture excels in capturing long-range dependencies and contextual information [28], making it particularly effective in tasks that require understanding and generating complex sequences.

The attention mechanism [29], which works as a core algorithm for transformer models, calculates the attention weights between each pairs of elements in the input sequence. It achieves this by utilizing three representations of the input: query, key and a value, represented by Q,K and V. The attention weights are calculated as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{2.1}$$

Where d_k denotes the dimension of the key vectors and serves as a scaling factor to prevent excessive large value of the dot products. The computed attention values are then employed to compute a weighted sum of the value vectors [15]. This allows transformers to concentrate on the most important parts of the input sequence. The process is repeated for each attention head in multi-head attention [30], with the final outputs passed to a feedforward neural network [30].

Transformer-based models can be broadly categorized into two approaches: autoregressive (AR) [31] and non-autoregressive (NAR) [32] methods. Autoregressive text generation is the most common approach used in large language models [13]. In this method, the model generates text sequentially, predicting one token at a time with the information of previously generated tokens. The probability of the entire sequence is defined as the product of conditional probabilities:

$$p(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} p(x_t | x_1, x_2, \ldots, x_{t-1}), \tag{2.2}$$

where $x_1, x_2, \ldots, x_T$ are the tokens in the generated sequence.

NAR models, including BERT [33] and its variants [34, 35], typically employ masking and iterative refinement strategies [36] to generate entire sequences in parallel. Although this paradigm yields substantial speedups over autoregressive approaches, NAR models often struggle to maintain coherence and fluency in generated text [37]. This limitation stems from the conditional independence assumption underlying NAR generation, which can introduce inconsistencies across output positions [38].

2.2 Variation of Large Language Models

2.2.1 Llama Models

Meta AI's Llama series, introduced in February 2023, has emerged as a pivotal force in democratizing large-scale language modeling [7]. By making high-performance architectures publicly accessible, the Llama family has substantially narrowed the capability gap between open-source and proprietary systems [8].

The series has evolved considerably across multiple releases. Model capacities now span from 1 billion to 2 trillion parameters, with recent iterations adopting mixture-of-experts designs to improve computational efficiency. Whereas the original release offered only pretrained foundation models, Llama 2 marked a strategic shift by simultaneously distributing instruction-tuned variants optimized for downstream applications [8]. The most recent release,

TABLE 2.1: Comparison of prominent large language models and vision models. Models are organized by type and listed chronologically within each category. The rapid growth in model parameters demonstrates the trend toward larger architectures, though practical deployment constraints often favor smaller, more efficient models.

Model	Parameters	Year	Architecture Type
Vision Models			
ViT [39]	86M–632M	2020	Vision Transformer
CLIP [40]	151M–428M	2021	Vision-Language Contrastive
Florence [41]	893M	2021	Vision-Language Contrastive
CoAtNet-7 [42]	2.4B	2021	Conv-Attention Hybrid
Encoder-only Language Models			
BERT [16]	110M–340M	2018	Bidirectional Encoder
Encoder-Decoder Language Models			
T5 [17]	60M–11B	2019	Text-to-Text Transformer
Megatron-LM [43]	8.3B	2020	Large-scale Transformer
Flan-T5 [44]	60M–11B	2022	Instruction-tuned T5
Decoder-only Language Models			
GPT-3 [18]	125M–175B	2020	Autoregressive Decoder
OPT [45]	125M–175B	2022	Open GPT-3 Alternative
Llama Family Models			
Llama 1 [7]	7B, 13B, 30B, 65B	2023	RMSNorm + SwiGLU
Llama 2 [8]	7B, 13B, 70B	2023	Improved Context Length
Llama 2-Chat [8]	7B, 13B, 70B	2023	RLHF Fine-tuned
Code Llama[46]	7B, 13B, 34B	2023	Code-specialized
Llama 3[47]	8B, 70B	2024	GQA + Enhanced Training
Llama 3.1[48]	8B, 70B, 405B	2024	Extended Context (128K)
Llama 3.2[48]	1B, 3B, 11B, 90B	2024	Multimodal Capabilities
DeepSeek Models			
DeepSeek-LLM[49]	7B, 67B	2024	Base Language Model
DeepSeek-Coder[50]	1.3B–33B	2024	Code-specialized
DeepSeek-MoE[51]	16B (2.8B active)	2024	Mixture of Experts
DeepSeek-V2[52]	236B (21B active)	2024	MLA + DeepSeekMoE
DeepSeek-V3[53]	671B (37B active)	2024	MLA + DeepSeekMoE
Other Large-scale Models			
Megatron-Turing NLG [54]	530B	2022	Largest Dense Model
GPT-4 [4]	Undisclosed	2023	Multimodal

Llama 3 [47], continues this trajectory of iterative refinement—each generation extending architectural innovations and broadening the practical scope of open-weight language models.

2.2.1.1 Llama Architecture Development

The first model of this family, Llama 1, was built based on the encoder-decoder transformer architecture developed by Vaswani et al. in 2017 [30]. Llama 1 introduced several enhancements to the original Transformer architecture [7], combining it with several improvements that would become the foundation for subsequent iterations.

Pre-Normalization: Inspired by the improvement of training stability implemented in the architecture of GPT-3 [18], Llama 1 normalizes the input of each transformer sub-layer rather than only the output [7]. This approach facilitates more efficient gradient flow during the backpropagation process by allowing error gradients to flow directly from top to bottom layers without passing through the normalization operations.

RMSNorm: They replaced the traditional LayerNorm function with RMSNorm (Root Mean Square Norm) [55], which is more computationally efficient while preserving training stability and increasing model convergence. RMSNorm retains the re-scaling invariance property while simplifying the computation compared to LayerNorm [7].

SwiGLU Activation Function: Regarding the activation function, Llama 1 replaced the well-known ReLU with the SwiGLU function [56], which has been shown to improve model performance. SwiGLU combines Swish and GLU activations and uses a gating mechanism that selectively activates neurons based on the received input [7].

Rotary Positional Embeddings (RoPE): Llama models employ Rotary Positional Embeddings [57], which unify the advantages of absolute and relative positional encoding schemes. RoPE encodes positional information by applying position-dependent rotations to paired dimensions within query and key vectors, enabling the attention mechanism to capture token positions implicitly through the resulting inner product structure [7].

Meta released Llama 2 in July 2023 in partnership with Microsoft, marking the second generation of the Llama family [8]. While the core architecture remains largely consistent with its predecessor, Llama 2 introduces notable enhancements in training data scale, context length capacity, and attention mechanism design.

Enhanced Training Data: The Llama 2 foundation models were trained on a corpus comprising 2 trillion tokens, representing a 40 percent increase over the original Llama dataset [8]. The curation process filtered websites prone to disclosing personal information while up-sampling sources deemed reliable.

Extended Context Length: Llama 2 doubles the maximum context window from 2048 to 4096 tokens [8], enabling the model to accommodate longer input sequences. This expansion improves performance on tasks requiring extended textual reasoning, including question answering for long context, summarization, and multi-turn dialogue.

Grouped Query Attention (GQA): The larger 34B and 70B variants employ Grouped Query Attention [58], an interpolation between Multi-Head Attention and Multi-Query Attention. Rather than maintaining independent key-value projections for each query head, GQA allows multiple query heads to reference a shared set of key and value representations. This parameter sharing substantially reduces KV cache memory requirements and accelerates inference throughput—critical considerations for deploying models at scale—while incurring only marginal degradation in output quality [8].

Reinforcement Learning from Human Feedback (RLHF): A notable departure from its predecessor, Llama 2 incorporates reinforcement learning from human feedback into its training pipeline [21]. By optimizing against human preference signals, the model exhibits markedly improved conversational utility compared to the original Llama release [8].

Llama 3 features the transformer-based architecture[15] with parameter sizes from 8 billion to 70 billion, while scaling its model size, it also utilizes enhanced attention mechanisms and significantly improved training data [48].

Universal Grouped Query Attention: While Llama 3 preserves the broad architectural foundations of its predecessor, it extends Grouped Query Attention across all model sizes—a technique previously limited to the larger 34B and 70B configurations [48]. By compressing key and value representations within the attention mechanism, GQA yields substantial reductions in KV cache memory consumption at inference time [58].

Enhanced Context Window: Llama 3 extends the context window to 8,192 tokens, doubling the 4,096-token limit of Llama 2 and quadrupling the original 2,048-token capacity of Llama 1 [48]. Although this remains modest compared to models such as GPT-4 [4], it represents a continued progression within the Llama series.

Improved Tokenizer: Llama 3 introduced a more efficient tokenizer and expanded the vocabulary size to 128,256 tokens [48], enhancing the model's ability to process diverse text inputs more effectively.

Massive Training Dataset: The training dataset size for Llama 3 variants has scaled dramatically to 15 trillion tokens, a substantial increase from the 2 trillion tokens used for Llama 2 [48], significantly improving the model's performance under various content scenarios.

2.2.1.2 Llama Model Variants

The Llama family includes multiple model variants, each designed for specific use cases:

Llama 3.1: Released in 2024, Llama 3.1 extends the capabilities of Llama 3 with models ranging from 8B to 405B parameters [47]. The flagship 405B model represents Meta's largest open-source model to date. Key improvements include an extended context window of 128K tokens and enhanced multilingual support.

Llama 3.2: Also released in 2024, Llama 3.2 introduces lightweight models (1B and 3B parameters) optimized for edge deployment and mobile devices [47]. Additionally, it includes 11B and 90B multimodal variants that can process both text and images, marking Meta's entry into open-source multimodal language models.

Code Llama: Released in 2023, Code Llama is a specialized variant fine-tuned on code datasets, available in 7B, 13B, and 34B parameter sizes [46]. It supports code completion, generation, and infilling capabilities with extended context length for handling longer code snippets.

Llama 2-Chat: The conversational variants of Llama 2 (7B, 13B, 70B) are fine-tuned using RLHF for improved dialogue capabilities [8]. These models demonstrate significantly better performance in multi-turn conversations compared to the base models.

2.2.2 Deepseek Model

DeepSeek-R1 represents a groundbreaking advancement in reasoning-focused large language models, introducing the first-generation reasoning models DeepSeek-R1-Zero and DeepSeek-R1 [59]. These models demonstrate that reasoning capabilities can be incentivized purely through reinforcement learning without relying on supervised fine-tuning as a preliminary step. The DeepSeek-R1 series achieves performance comparable to other state-of-the-art generative models on reasoning tasks, marking a significant milestone in open-source AI reasoning capabilities [59].

The development of DeepSeek-R1 follows the inference-time scaling paradigm pioneered by OpenAI's o1 series [60], which extends Chain-of-Thought reasoning traces to achieve substantial gains on tasks requiring mathematical, programming, and scientific reasoning. However, DeepSeek-R1 departs from prior methods that depend heavily on curated supervised data. Instead, the work demonstrates that large-scale reinforcement learning alone can elicit strong reasoning capabilities—bypassing the supervised fine-tuning stage typically used for initialization [59].

2.2.2.1 Deepseek Base Architecture

DeepSeek-R1 and DeepSeek-R1-Zero are built upon the DeepSeek-V3-Base architecture, which utilizes a Mixture of Experts (MoE) design [53]. DeepSeek-V3 is a strong Mixture-of-Experts (MoE) language model with 671B total parameters with 37B activated for each token [53]. This architecture enables these models to have a massive parameter count while maintaining computational efficiency during inference.

The underlying architecture incorporates several key innovations: Mixture of Experts (MoE) Framework: DeepSeek-R1 incorporates a cutting-edge Mixture of Experts (MoE) framework

that allows the model to have a large total parameter count (671B) while only activating a small portion (37B) during each forward pass [53]. DeepSeekMoE improves upon the standard MoE architecture by using a larger number of smaller experts (Fine-Grained Expert Segmentation) and isolating some experts as shared ones [61].

Multi-Head Latent Attention (MLA): DeepSeek-V3 achieves efficient inference and low-cost training by adopting Multi-Head Latent Attention alongside the DeepSeekMoE architecture, both validated extensively in DeepSeek-V2 [52]. MLA introduces low-rank joint compression of attention keys and values, substantially reducing memory overhead during inference while preserving attention quality [53].

Advanced Transformer Components: DeepSeek-R1 builds upon enhanced transformer layers that incorporate sparse attention mechanisms and optimized tokenization strategies to model contextual dependencies more effectively [59]. Additionally, the architecture employs hybrid attention schemes that dynamically modulate attention weight distributions, enabling strong performance across both short-context and long-context regimes [53].

2.2.2.2 Deepseek Innovations

Load Balancing Strategy: DeepSeek-V3 introduces an auxiliary-loss-free approach to expert load balancing, together with multi-token prediction training objective to strengthen overall performance [53]. Rather than incorporating additional loss terms, this strategy adjusts a per-expert bias during Top-K expert selection: the bias for each expert is incremented or decremented by a fixed factor depending on whether that expert is currently underutilized or overloaded [53].

Multi-Token Prediction (MTP): DeepSeek-V3 introduces the multi-token prediction objective that enables the model to generate several tokens simultaneously during training [53]. Beyond improving training efficiency, this formulation naturally supports speculative decoding[62] at inference time, offering a pathway to accelerated generation [63].

2.3 Optimization Techniques for Large Generative Models

The deployment of large language models in production environments faces significant computational and memory constraints. While models with billions of parameters achieve state-of-the-art performance, their practical application requires sophisticated optimization techniques to reduce inference latency, memory footprint, and computational costs. This section examines two fundamental approaches to model optimization: low-rank approximation methods that compress model parameters while preserving performance, and speculative decoding techniques that accelerate inference through parallel token generation.

2.3.1 Low-Rank Approximation and Parameter-Efficient Fine-Tuning

Low-rank approximation techniques exploit the observation that weight matrices in neural networks often exhibit low intrinsic dimensionality [64]. This property enables significant compression and efficient adaptation of large models through decomposition methods that reduce the effective number of parameters while maintaining model expressiveness.

2.3.1.1 Singular Value Decomposition for Model Compression

Singular Value Decomposition (SVD) provides a foundational matrix factorization technique widely applied in neural network compression [65]. Given a weight matrix $W \in \mathbb{R}^{m \times n}$, SVD factorizes it into three components:

$$W = U\Sigma V^T \tag{2.3}$$

Here, $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices whose columns correspond to the left and right singular vectors, respectively, while $\Sigma \in \mathbb{R}^{m \times n}$ is a diagonal matrix containing singular values in descending order.

The low-rank approximation is obtained by retaining only the top k singular values:

$$W_k = U_k \Sigma_k V_k^T \tag{2.4}$$

where $U_k \in \mathbb{R}^{m \times k}$, $\Sigma_k \in \mathbb{R}^{k \times k}$, and $V_k \in \mathbb{R}^{n \times k}$ represent truncated matrices. This approximation minimizes the Frobenius norm of the reconstruction error:

$$\|W - W_k\|_F^2 = \sum_{i=k+1}^{\min(m,n)} \sigma_i^2 \tag{2.5}$$

The compression ratio achieved through SVD depends on the rank k and original dimensions, with the compressed representation requiring $k(m + n + 1)$ parameters compared to the original mn parameters [66].

2.3.1.2 Low-Rank Adaptation (LoRA)

LoRA introduces a parameter-efficient fine-tuning paradigm that freezes the weights of the base model and injects trainable rank decomposition matrices [67]. For a pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$ in the base model, LoRA [67] represents the adapted weights as:

$$W = W_0 + \Delta W = W_0 + BA \tag{2.6}$$

where $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$ are low-rank matrices with rank $r \ll \min(d, k)$. During training, only A and B are updated while W_0 remains frozen, reducing the number of trainable parameters from dk to $r(d + k)$.

The forward pass with LoRA becomes:

$$h = W_0 x + \frac{\alpha}{r} BAx \tag{2.7}$$

where α is a scaling factor that controls the magnitude of the low-rank adaptation. The initialization scheme sets A with a random Gaussian distribution and B to zero, ensuring that the weight difference ΔW is 0 at the start of training [67].

LoRA offers several advantages over full fine-tuning:

- **Memory Efficiency**: Reduces GPU memory requirements by maintaining only small trainable matrices
- **Storage Efficiency**: Multiple task-specific adaptations can be finetuned and stored as compact LoRA modules (typically <1% of model size)
- **Inference Flexibility**: LoRA weights can be merged into base weights for zero additional inference latency
- **Composition**: Multiple LoRA modules can be combined for multi-task learning [68]

2.3.1.3 Variants and Extensions

Recent work has extended the LoRA framework in several directions:

QLoRA combines quantization with LoRA, enabling fine-tuning of quantized models with minimal memory overhead [69]. By storing the base model in 4-bit precision and computing LoRA updates in higher precision, QLoRA reduces memory requirements by up to 75% while maintaining comparable performance.

AdaLoRA introduces adaptive rank allocation across different weight matrices [70]. Rather than using a fixed rank for all layers, AdaLoRA dynamically adjusts the rank based on the importance of each weight matrix, measured through singular value decomposition during training.

2.3.2 Key-Value Cache Compression

The key-value (KV) cache represents a critical memory bottleneck in transformer inference, storing past attention states to avoid redundant computation during autoregressive generation

[71]. For a model with L layers, H attention heads, sequence length n, and head dimension d_h, the KV cache requires $O(L \cdot H \cdot n \cdot d_h)$ memory per batch element, which grows linearly with sequence length and can exceed the model parameters for long contexts.

2.3.2.1 KV Cache Fundamentals

During autoregressive generation, transformers cache key and value tensors from previous time steps to compute attention scores efficiently. For each attention head at position t, the attention computation becomes:

$$\text{Attention}(Q_t, K_{1:t}, V_{1:t}) = \text{softmax}\left(\frac{Q_t K_{1:t}^T}{\sqrt{d_k}}\right) V_{1:t} \tag{2.8}$$

where $K_{1:t}$ and $V_{1:t}$ represent cached keys and values from positions 1 to t. Without caching, regenerating these tensors would require $O(t^2)$ operations for generating t tokens, compared to $O(t)$ with caching [71].

The memory footprint of the KV cache for a single sequence is:

$$M_{\text{KV}} = 2 \cdot L \cdot H \cdot n \cdot d_h \cdot \text{sizeof(dtype)} \tag{2.9}$$

For modern large language models, this can reach several gigabytes per sequence. For instance, a 70B parameter model with 80 layers, 64 attention heads, and 128-dimensional heads requires approximately 5GB for a 32K context length in FP16 precision [72].

2.3.2.2 Compression Techniques

Multi-Query and Grouped-Query Attention Multi-Query Attention (MQA) shares a single key-value head across all query heads, reducing the KV cache by a factor of H [73]:

$$M_{\text{MQA}} = \frac{2 \cdot L \cdot n \cdot d_h \cdot \text{sizeof(dtype)}}{H} \tag{2.10}$$

Grouped-Query Attention (GQA) provides a middle ground, sharing key-value heads among groups of query heads [58]. With G groups, the cache reduction factor is H/G, balancing memory efficiency with model quality.

Token Eviction and Pruning Several methods selectively retain important tokens in the cache based on attention patterns:

H2O (Heavy-Hitter Oracle) identifies and retains tokens with consistently high attention scores across layers and heads [74]. The algorithm maintains a priority queue of tokens based on cumulative attention weights:

$$s_i = \sum_{l=1}^{L} \sum_{h=1}^{H} \sum_{j>i} \alpha_{l,h,j,i} \tag{2.11}$$

where $\alpha_{l,h,j,i}$ represents the attention weight from position j to position i in layer l, head h.

StreamingLLM maintains attention sinks—initial tokens that accumulate high attention scores—alongside a sliding window of recent tokens [75]. This approach enables stable generation for arbitrarily long sequences with bounded memory:

$$\text{Cache} = \{K_{1:s}, V_{1:s}\} \cup \{K_{t-w:t}, V_{t-w:t}\} \tag{2.12}$$

where s represents attention sink tokens and w is the window size.

Quantization and Low-Rank Approximation KV cache quantization reduces memory through lower-precision representations. KIVI demonstrates that 2-bit quantization of keys and values achieves minimal quality degradation [76]:

$$\tilde{K} = Q_k(K), \quad \tilde{V} = Q_v(V) \tag{2.13}$$

where Q_k and Q_v are quantization functions mapping to 2-bit representations.

Low-rank decomposition approximates the KV cache through factorization [77]:

$$K \approx U_K V_K^T, \quad V \approx U_V V_V^T \tag{2.14}$$

where $U_K, U_V \in \mathbb{R}^{n \times r}$ and $V_K, V_V \in \mathbb{R}^{d_h \times r}$ with rank $r \ll \min(n, d_h)$.

Dynamic Sparse Attention Sparse attention mechanisms compute attention only for relevant token pairs, implicitly compressing the effective cache [78]. Patterns include:

- **Sliding Window**: Attend to recent w tokens
- **Global Tokens**: Designated tokens attend to all positions
- **Random Attention**: Stochastic sampling of attention connections
- **Learned Patterns**: Data-driven sparse structures [79]

2.3.2.3 System-Level Optimizations

PagedAttention manages KV cache memory using virtual memory and paging techniques [72]. By storing the cache in non-consecutive memory blocks and using a page table for address translation, PagedAttention achieves:

- Near-zero memory waste through fine-grained allocation
- Efficient memory sharing across parallel requests
- Dynamic memory management without pre-allocation

Flash-Attention optimizes memory access patterns for KV cache operations [80]. By partitioning computation across the sequence dimension and leveraging parallelism across attention heads, Flash- achieves up to 8× speedup in long-context generation while maintaining the full KV cache.

Cascade Inference uses a small model to generate and cache KV states, which are then refined by a larger model [81]. This hierarchical approach reduces overall memory requirements while maintaining generation quality.

2.3.3 Speculative Decoding

The computational demands of autoregressive language model inference present significant bottlenecks for practical deployment. Each token generation requires a complete forward pass through the entire model, creating sequential dependencies that prevent parallel token generation. This fundamental constraint has motivated the development of speculative decoding [62, 82], a technique that leverages smaller auxiliary models to propose multiple tokens simultaneously, which are then verified by the target model in a single evaluation step.

The theoretical elegance of speculative decoding lies in its distribution-preserving property: the final output maintains identical statistical characteristics to direct sampling from the target model. This guarantee enables acceleration without compromising generation quality, distinguishing speculative decoding from approximation-based speedup methods that may introduce distributional shifts [62, 82].

2.3.3.1 Mathematical Framework

Problem Formulation

Consider a target language model M_t with parameters θ_t that defines a probability distribution over token sequences. For an input context $x_{1:n}$, standard autoregressive generation produces the next token by sampling from:

$$x_{n+1} \sim p_{M_t}(x_{n+1}|x_{1:n}) \tag{2.15}$$

The computational cost scales linearly with sequence length, as each token requires an independent forward pass. Speculative decoding addresses this limitation by introducing a draft model M_d with parameters θ_d, where $|\theta_d| \ll |\theta_t|$, enabling faster inference at the cost of reduced accuracy [83].

Distribution Preservation Theorem

The core theoretical contribution of speculative decoding establishes that the output distribution remains mathematically equivalent to direct target model sampling. This fundamental property guarantees that speculative decoding produces identical outputs distribution to autoregressive generation only with target models [83].

Theorem: For any distributions $p(x)$ and $q(x)$, tokens sampled via speculative sampling from $p(x)$ using $q(x)$ as the draft distribution are distributed identically to those sampled from $p(x)$ alone [62, 82].

Proof: Let β be the acceptance probability for a token x' sampled from $q(x)$:

$$\beta = \min\left(1, \frac{p(x')}{q(x')}\right) \tag{2.16}$$

The adjusted residual distribution $p'(x)$ used when rejection occurs has normalizing constant:

$$p'(x) = \frac{\max(0, p(x) - q(x))}{1 - \sum_z \min(p(z), q(z))} \tag{2.17}$$

Since $\sum_z \min(p(z), q(z)) = \beta$ by the definition of acceptance probability, the normalizing constant becomes $1 - \beta$.

The total probability of sampling token x' through speculative decoding is [62]:

$$\begin{aligned}
P(x = x') &= P(\text{accepted}, x = x') + P(\text{rejected}, x = x') \\
&= q(x') \min\left(1, \frac{p(x')}{q(x')}\right) + (1 - \beta)p'(x') \\
&= \min(q(x'), p(x')) + \frac{p(x') - \min(q(x'), p(x'))}{1 - \beta} \cdot (1 - \beta) \\
&= \min(p(x'), q(x')) + p(x') - \min(p(x'), q(x')) \\
&= p(x')
\end{aligned} \tag{2.18}$$

Therefore, the final sampling distribution exactly matches the target distribution $p(x)$, regardless of the quality of the draft distribution $q(x)$. This distribution preservation property

ensures that speculative decoding maintains identical statistical characteristics to direct target model sampling while potentially achieving significant speedup.

The fundamental speculative sampling procedure, detailed in Algorithm 1, operates through three distinct phases: draft generation, parallel verification, and probabilistic acceptance. The draft model generates k speculative tokens sequentially, creating a candidate continuation of length k. The target model then evaluates all proposed tokens simultaneously, computing probability distributions for each position.

The acceptance decision for each speculative token $\tilde{x}_i$ depends on the ratio $r_i = \frac{p_t(\tilde{x}_i)}{p_d(\tilde{x}_i)}$. When $r_i \geq 1$, the target model assigns higher probability than the draft model, leading to automatic acceptance. For $r_i < 1$, acceptance occurs with probability r_i, maintaining the target distribution through rejection sampling.

Upon rejection at position i, the algorithm samples from a residual distribution that accounts for the probability mass difference between target and draft models:

$$p_{\text{residual}}(y) = \frac{\max(0, p_t(y) - p_d(y))}{\sum_z \max(0, p_t(z) - p_d(z))} \tag{2.19}$$

This residual sampling ensures that the final token distribution matches the target model exactly, even when speculation fails [83].

2.3.3.2 Neural Speculative Decoding Approaches

Modern speculative decoding predominantly relies on neural draft models that approximate the target model's behavior while requiring significantly less computation. Several state-of-the-art approaches have emerged:

Tree-based Speculation: SpecInfer [84] extends speculative decoding to tree structures, exploring multiple speculation paths simultaneously. These methods increase the likelihood of finding acceptable token sequences but require more complex verification procedures and memory management.

Algorithm 1 Speculative Sampling Algorithm

Require: Draft model M_d, Target model M_t, input sequence $x_{1:n}$, speculation length k
Ensure: Next token(s) sampled from target distribution p_{M_t}

Draft Phase:
for $i = 1$ to k **do**
  $p_d^{(i)} \leftarrow M_d(x_{1:n+i-1})$ {Draft model forward pass}
  $\tilde{x}_{n+i} \sim p_d^{(i)}$ {Sample speculative token}
end for
Verification Phase:
$\{p_t^{(1)}, p_t^{(2)}, \ldots, p_t^{(k+1)}\} \leftarrow M_t(x_{1:n}, \tilde{x}_{n+1}, \ldots, \tilde{x}_{n+k})$ {Target model parallel evaluation}
Acceptance Phase:
for $i = 1$ to k **do**
  $\alpha_i \leftarrow \min\left(1, \frac{p_t^{(i)}(\tilde{x}_{n+i})}{p_d^{(i)}(\tilde{x}_{n+i})}\right)$ {Acceptance probability}
  $u_i \sim \text{Uniform}(0, 1)$
  if $u_i \leq \alpha_i$ **then**
    Accept $\tilde{x}_{n+i}$, set $x_{n+i} = \tilde{x}_{n+i}$
  else
    Rejection Sampling:
    $p_{\text{resid}}^{(i)}(y) \leftarrow \max\left(0, \frac{p_t^{(i)}(y) - p_d^{(i)}(y)}{\sum_z \max(0, p_t^{(i)}(z) - p_d^{(i)}(z))}\right)$
    $x_{n+i} \sim p_{\text{resid}}^{(i)}$ {Sample from residual distribution}
    return $x_{1:n+i}$ {Stop speculation, return sequence}
  end if
end for
Final Token Generation:
if All k tokens accepted **then**
  $x_{n+k+1} \sim p_t^{(k+1)}$ {Sample additional token from target}
  return $x_{1:n+k+1}$
end if

Architecture-Optimized Draft Models: Recent advances introduce draft models with architectures tailored for efficient inference.

Medusa [85], which accelerates decoding by attaching a set of lightweight non-autoregressive *prediction heads* to intermediate layers of the target model. Instead of training a separate draft model, Medusa reuses the hidden states of the target model and equips each head with the ability to predict tokens several steps ahead. During inference, these parallel heads generate candidate continuations that can be verified in a single forward pass of the target model. This design eliminates the need for an external speculator and tightly integrates speculative

decoding within the base model itself, achieving speedups with minimal additional parameters and training cost.

Eagle [86–88] exemplifies this trend through *hidden-state conditioning* with llama-based lightweight autoregressive speculative, a design in which the draft model is trained to directly predict the target model's next hidden representations based on previous target model hidden states rather than only token distributions. At training time, Eagle learns to map the current hidden state of the target model, concatenated with the token embedding, to the next-step hidden state and corresponding token. Compared to Medusa[85], Eagle model's autoregressive approach generate more accurate tokens that are highly aligned with early tokens and is able to propose more candidate tokens. Besides, it also absorbs target model's information to the draft model and alleviates the train–inference mismatch that plagues earlier neural speculator, enabling Eagle to propose longer token blocks with higher acceptance rates.

Limitations of Neural Approaches: Despite their effectiveness, neural draft models face inherent challenges:

- Substantial training costs requiring GPU resources and large datasets, often taking days converge
- Model-specific optimization limiting transferability across different target models and domains
- Sequential token generation creating speculation bottlenecks even with optimized architectures
- GPU memory requirements restricting deployment scenarios, particularly in edge computing environments
- Difficulty in adapting to domain-specific patterns without extensive retraining

2.3.3.3 Non-Neural Speculative Decoding Approaches

N-gram models represent one of the earliest approaches to statistical language modeling [89, 90]. Despite their simplicity, they remain remarkably effective for capturing local dependencies in text [91]. An n-gram model estimates:

$$P(w_t|w_{t-n+1},\ldots,w_{t-1}) = \frac{\text{count}(w_{t-n+1},\ldots,w_{t-1},w_t)}{\text{count}(w_{t-n+1},\ldots,w_{t-1})} \tag{2.20}$$

This approach can generate full next-token distributions efficiently, making it suitable for speculative decoding.

Infinigram [92] fundamentally reimagines n-gram language modeling by removing traditional constraints on context length. Unlike classical n-gram models that are limited to fixed context windows, Infinigram enables unbounded context matching through sophisticated indexing structures.

Key innovations of the original Infinigram system include:

- **Suffix Array Architecture**: Infinigram employs suffix arrays and FM-index structures to enable $O(\log n)$ lookup time for arbitrary-length n-grams. This allows the system to search through trillion-token corpora in microseconds, finding the longest matching context without pre-specifying n-gram length.
- **Dynamic Context Selection**: Rather than using fixed n-gram orders, Infinigram dynamically determines the optimal context length based on available matches in the corpus. If a 50-token exact match exists in the training data, the system can leverage this full context for prediction.
- **Efficient Compression**: The system uses sophisticated compression techniques including run-length encoding and delta compression to store massive text corpora efficiently. A 1.4 trillion token corpus can be indexed in approximately 7TB of storage.

N-gram models face several limitations when applied for speculative decoding:

- **Sparsity:** As n increases, traditional n-grams appear rarely or never in training data
- **Context limitation:** Fixed context window cannot capture long-range dependencies
- **Unexplored deployment** Although Infinigram provides solutions to traditional ngram limitations, it's performance as a speculative decoding model remains unexplored.

SuffixDecoding [93] eliminates the need for a separate draft model by leveraging efficient suffix trees to cache token sequences from both prior outputs (global tree) and the current request (per-request tree). Given a recently generated suffix, the algorithm rapidly locates matching continuations in these trees and greedily expands a *speculation tree*, guided by frequency-based statistics to prioritize likely tokens. To avoid wasted verification, the speculation length is set adaptively: longer pattern matches trigger longer speculations, while shorter matches result in more conservative drafts. Among multiple candidates, the speculation tree with the highest expected acceptance score is selected for verification. Because suffix trees operate entirely in CPU memory with negligible overhead, SuffixDecoding achieves draft token generation on the order of tens of microseconds per token. Moreover, it supports a hybrid mode, dynamically falling back to model-based methods (e.g., Eagle-3) when repetition is limited, thus combining the efficiency of model-free speculation with the robustness of model-based approaches.

Despite its efficiency, SuffixDecoding faces scalability challenges.

- **Memoroy Expensive:** Constructing and maintaining large suffix trees can be memory-intensive: although the data structure is linear in size, storing millions of tokens across many requests can consume significant CPU memory, and its tree structure makes it challenging to save to local disk with decent memory comsuption, raising concerns for very large-scale deployments.
- **Scalability:** To mitigate unbounded growth, SuffixDecoding relies on token eviction strategies, which prevent it from storing arbitrarily long histories.

2.4 Research Gaps and Opportunities

2.4.1 Limitations of Current Approaches

We summarize the limitations of above techniques into two main categories:

2.4.1.1 Activation-Agnostic Model Compression

Existing low-rank approximation methods, particularly naive SVD approaches, primarily focus on minimizing weight-space reconstruction error without considering how models actually process input activations during inference. This fundamental limitation leads to:

- **Uniform rank allocation**: Current SVD compression methods apply uniform compression across layers, ignoring heterogeneous spectral properties
- **Suboptimal compression quality**: Weight-space optimization fails to preserve activation patterns that are crucial for model performance
- **Poor zero-shot performance**: Compression methods that ignore activation distributions suffer significant accuracy degradation

2.4.1.2 Inefficient Speculative Decoding Mechanisms

Current speculative decoding approaches rely heavily on neural draft models. Despite non-neural models show promising research directions, they still have several inherent limitations in practice. The key limitations of them include:

- **Neural High computational overhead**: Neural draft models require substantial GPU resources and training time
- **Neural Sequential generation bottleneck**: Even small draft models must generate tokens sequentially, limiting speedup potential
- **Neural Limited domain adaptability**: Neural approaches struggle to exploit domain-specific patterns in structured tasks like code generation and capture common repeated patterns
- **Non-Neural Scalability**: Complicated data structures like suffix trees[93] can become memory-intensive and hard to manage at scale, other promising implementations like Infinigram[92] still lack statistical evaluation and deployment as a speculator.

- **Non-Neural Limited context modeling**: Performance degradation on speed when searching on large corpus and long prefix context to retrieve full next-token distributions or next k tokens for verification

2.4.2 Identified Research Opportunities

These limitations reveal two complementary opportunities for advancing efficient large model deployment:

2.4.2.1 Opportunity 1: Activation-Aware Model Conversion

The gap between weight-space and activation-space optimization presents an opportunity to develop conversion methods that:

- Preserve activation patterns through covariance-weighted decomposition
- Adaptively allocate compression resources based on layer-specific spectral properties
- Maintain model capabilities while achieving aggressive memory reduction

This opportunity motivates our CARE method (Chapter 3), which introduces activation-preserving factorization for converting GQA models to efficient MLA architectures.

2.4.2.2 Opportunity 2: Statistical Foundation for Speculative Decoding

The limitations of current neural and non-neural draft models suggest exploring alternative approaches:

- Leverage classical statistical methods with negligible computational overhead even with large datasets
- Be able to search on variable length prefix rather than fixed-length n-grams to capture long-range dependencies
- Build memory-efficient and distributed large non-neural models to fully approximate target model output on complicated text patterns
- Adapt to target model distributions through direct training of target model outputs

- Exploit domain-specific patterns for enhanced performance

This opportunity leads to our Infinigram-based speculative decoding (Chapter 4), which uses n-gram statistics to achieve efficient speculation with CPU-only resources.

2.4.3 Transition to Proposed Solutions

The identified research gaps establish clear objectives for this thesis:

(1) **Chapter 3 - CARE**: Address the KV-cache memory bottleneck through activation-aware model conversion that preserves performance while reducing memory footprint
(2) **Chapter 4 - Infinigram Speculator**: Tackle inference latency through statistical speculative decoding that achieves significant speedup without neural overhead

These complementary approaches target distinct aspects of the deployment challenge—memory efficiency and computational speed—providing a comprehensive solution for practical large model deployment. By addressing both the structural (model architecture) and algorithmic (inference strategy) dimensions, this work contributes to making large generative models more accessible and deployable across diverse computational environments.

CHAPTER 3

Covariance-Aware and Rank-Enhanced Decomposition for Latent Attention

---

This chapter presents CARE (Covariance-Aware and Rank-Enhanced), a post-hoc conversion method for transforming pretrained Grouped Query Attention (GQA) models into Multi-Head Latent Attention (MLA) under fixed KV-cache budgets. CARE addresses key limitations of naive SVD approaches by introducing activation-preserving factorization and adjusted-rank allocation across layers, achieving superior performance while maintaining KV-cache efficiency.

3.1 Introduction and Motivation

Large Language Models (LLMs) deliver impressive capabilities but at high inference cost, with the key-value (KV) cache in self-attention emerging as a primary memory and bandwidth bottleneck [24]. While Grouped Query Attention (GQA) [58] reduces KV cache size by sharing keys/values within head groups, Multi-Head Latent Attention (MLA) [94] offers a more efficient approach by compressing keys and values into low-dimensional latent vectors, dramatically reducing KV size while preserving or improving task accuracy.

However, the ecosystem is dominated by pretrained GQA checkpoints, and retraining from scratch under MLA is expensive. Recent conversion methods demonstrate feasibility but typically apply naive singular value decomposition (SVD) that focuses on minimizing weight-space error rather than activation-space error or enforces uniform rank across layers [95]. Activation-aware variants such as ASVD [96] partially mitigate this by rescaling weights by per-channel activation magnitudes before SVD, but they still optimize a weight-space

objective and do not address layer-wise rank heterogeneity. CARE addresses these limitations by introducing a covariance-aware factorization pipeline that aligns approximation with actual input activations and adaptively allocates ranks across layers based on spectral complexity.

3.2 Theoretical Foundation

3.2.1 Problem Formulation: GQA to MLA Conversion

Converting pretrained Grouped Query Attention (GQA) into Multi-Head Latent Attention (MLA) involves reparameterizing the key and value projections while maintaining equivalent KV-cache size. Given a GQA layer with n_h heads of size d_h split into g_h groups, we have grouped weight matrices $W_K^{(g)}, W_V^{(g)} \in \mathbb{R}^{D\times(g_h d_h)}$ where $D = n_h d_h$.

MLA reparameterizes these as:

$$K = (XW_a^K)W_b^K, \quad V = (XW_a^V)W_b^V \tag{3.1}$$

$$W_a^{\{K,V\}} \in \mathbb{R}^{D\times r}, \quad W_b^{\{K,V\}} \in \mathbb{R}^{r\times(n_h d_h)} \tag{3.2}$$

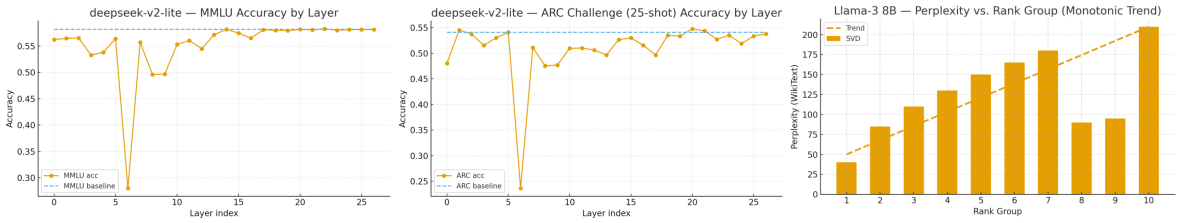
where only the latent $XW_a^{\{K,V\}} \in \mathbb{R}^{T\times r}$ is cached. To maintain KV-parity with the original GQA model, we enforce $r = g_h d_h$.

3.2.2 Limitations of Naive SVD

Direct SVD initialization minimizes weight-space error $\|W - \hat{W}\|_F$ rather than activation-space error $\|XW - X\hat{W}\|_F$, ignoring how the projection operates during inference. This fundamental mismatch creates several critical issues that undermine the effectiveness of naive conversion approaches.

Figure 3.1(a)(b) demonstrates the heterogeneous sensitivity of different layers to rank reduction, where randomly reducing each layer's rank by 50% shows that some layers suffer significant accuracy drops while others remain near baseline performance. This layer-wise

variation highlights the suboptimality of uniform rank allocation. Furthermore, Figure 3.1(c) reveals that SVD's singular values serve as imperfect proxies for MLA conversion importance—smaller singular value groups do not always correspond to smaller performance impacts, exhibiting non-monotonic sensitivity patterns that challenge the fundamental assumption of spectral truncation methods.



(a) (b) (c)

FIGURE 3.1: Sensitivity analysis of neural network layers to rank compression. Panels (a)(b) illustrate the performance impact when layer ranks are halved in the deepseek-v2-lite model, measuring ARC Challenge (25-shot) and MMLU accuracy against layer position. Baseline performance levels are indicated by dotted reference lines (ARC: 54.09%, MMLU: 58.16%). Results reveal non-uniform degradation patterns across the network depth, with certain layers exhibiting substantial performance drops while others maintain near-baseline accuracy. Panel (c) demonstrates systematic singular value group analysis on Llama-3-8B MLA components (layers 30–32), measuring WikiText perplexity changes. Singular values are organized into ten descending magnitude clusters, with each cluster's impact assessed through controlled truncation experiments averaged across the three target layers. The ideal monotonic relationship between singular value magnitude and performance impact (dotted trend) contrasts with observed behavior, where mid-range clusters (8–9) produce less degradation than expected, challenging the assumption that smaller singular values have proportionally smaller influence on MLA architectures.

3.2.2.1 Activation-Weight Mismatch

The core limitation of naive SVD lies in its focus on weight approximation rather than activation preservation. When decomposing $W \approx UV^T$, standard SVD minimizes $\|W - UV^T\|_F^2$, but the actual inference objective requires minimizing $\|XW - XUV^T\|_F^2$ for input activations X. This discrepancy becomes problematic when:

- Input activations have non-uniform variance across dimensions

- Weight matrices exhibit different spectral properties than their activation-weighted counterparts
- The approximation quality varies significantly across different input patterns

This mismatch induces attention-logit drift even when weight approximation appears accurate, leading to degraded downstream performance despite seemingly good reconstruction metrics.

3.2.2.2 Uniform Rank Allocation Problems

Enforcing uniform rank allocation across layers ignores the heterogeneous spectral complexity of different transformer components. This leads to:

- **Over-compression of complex layers**: Deep layers with rich spectral structure are allocated insufficient rank, causing significant information loss
- **Under-compression of simple layers**: Early layers with intrinsically low-rank structure receive excessive capacity allocation
- **Suboptimal resource utilization**: Limited rank budget is distributed inefficiently, failing to maximize overall approximation quality

The uniform approach treats all layers equivalently despite their varying roles in the model's representational hierarchy, resulting in conversion quality that scales poorly with compression aggressiveness.

3.2.3 Activation-Preserving Factorization

To address the activation-weight mismatch, CARE minimizes empirical activation error rather than weight error. Given activation batches $\{X_b\}_{b=1}^{N}$ from a calibration set, we define the covariance matrix:

$$C = \frac{1}{N} \sum_{b=1}^{N} X_b^T X_b \tag{3.3}$$

The activation-preserving objective becomes:

$$\min_{\text{rank}(\hat{W})\leq r} \frac{1}{N}\sum_{b=1}^{N} \|X_b(W-\hat{W})\|_F^2 \tag{3.4}$$

This can be rewritten as $\|\sqrt{C}(W-\hat{W})\|_F^2$. The optimal rank-$r$ solution is obtained by SVD on the covariance-weighted matrix:

$$\sqrt{C}\,W = U\Sigma V^T, \quad \text{then} \quad \hat{W} = \sqrt{C}^{-1}U_r\Sigma_r V_r^T, \tag{3.5}$$

where U_r, Σ_r, V_r retain the top-r components of U, Σ, V. In practice we use the shrinkage-regularized $\sqrt{C}_\lambda = (1-\alpha)\sqrt{C} + \alpha\lambda I$ to ensure invertibility. This covariance-weighted SVD preserves dominant activation directions more faithfully than naive weight-space SVD [97].

3.2.4 Adjusted-Rank Allocation

Due to heterogeneous spectral properties across attention layers, CARE adaptively distributes a fixed total rank budget R_{total} to maximize retained energy. For each layer ℓ, we compute the covariance-weighted singular values $\{\sigma_{\ell,K,m}\}$ and $\{\sigma_{\ell,V,m}\}$ of $\sqrt{C^{(\ell)}}\,W_K^{(\ell)}$ and $\sqrt{C^{(\ell)}}\,W_V^{(\ell)}$ respectively (the same activation-weighted matrices factored in Sec. 3.2.3).

The optimal rank allocation solves:

$$\max_{\{r_{\ell,K}, r_{\ell,V}\}} \sum_{\ell=1}^{L}\left(\sum_{m=1}^{r_{\ell,K}}\sigma_{\ell,K,m} + \sum_{m=1}^{r_{\ell,V}}\sigma_{\ell,V,m}\right) \tag{3.6}$$

subject to $\sum_{\ell=1}^{L}(r_{\ell,K}+r_{\ell,V}) = R_{total}$ and $r_{\ell,V} \leq R_{\ell,V}, r_{\ell,K} \leq R_{\ell,K}$.

This is solved greedily using a water-filling algorithm [98]: repeatedly increment the rank of the matrix that yields the largest increase in singular value sum. This approach allocates higher ranks to spectrally complex matrices while reclaiming capacity from intrinsically low-rank ones.

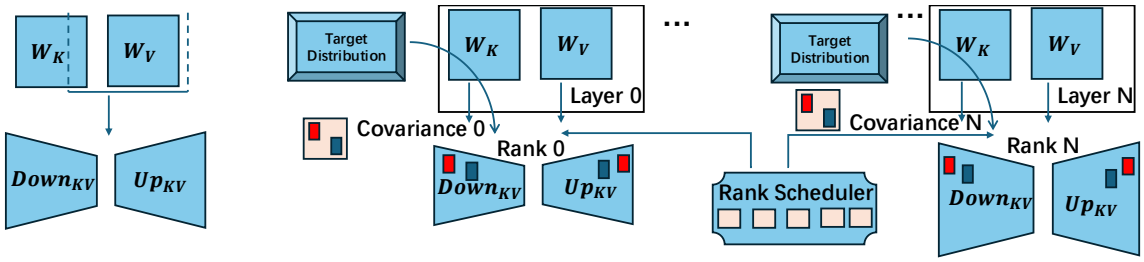


FIGURE 3.2: Overview of the CARE methodology for converting Group Query Attention models to Multi-Head Latent Attention architectures. The transformation pipeline consists of three sequential phases: first, statistical estimation of activation covariance matrices from representative calibration samples; second, spectral decomposition of key and value projection matrices using covariance-informed SVD; and third, optimal rank distribution via water-filling optimization that accounts for layer-specific spectral characteristics. This systematic approach maintains computational efficiency in the key-value cache while preserving the fidelity of neural activations throughout the conversion process.

3.3 CARE Implementation Pipeline

The CARE conversion process follows a systematic four-stage pipeline that transforms pretrained GQA models into efficient MLA architectures while preserving activation patterns and optimizing rank distribution. Each stage addresses specific challenges in the conversion process.

Design rationale. The order of the stages is not incidental but reflects a chain of dependencies between the empirical and algorithmic components of CARE. Stage 1 collects calibration activations because the inference-relevant objective is activation-space error $\|XW - X\hat{W}\|_F^2$, not weight-space error—this objective cannot be evaluated, let alone minimized, without first observing how the model actually exercises each layer. Stage 2 condenses those activations into per-layer covariance matrices $C^{(\ell)}$ with shrinkage regularization; the covariance is the minimal sufficient statistic for the activation objective and the regularization keeps $\sqrt{C^{(\ell)}}$ invertible for the subsequent steps. Stage 3 then reduces to a covariance-weighted SVD on $\sqrt{C^{(\ell)}}\, W^{(\ell)}_{\{K,V\}}$, which yields singular values that already encode activation importance—making them directly comparable across layers and across K/V projections. Only with

comparable spectra in hand does Stage 4's water-filling allocation become well-posed: it can assign each additional rank to the projection where the next singular value contributes most to retained energy, rather than distributing budget uniformly. Finally, Stage 5 (initialization) and Stage 6 (healing) close the residual gap between the rank-truncated reparameterization and the original GQA layer, with the activation-preserving initialization guaranteeing that healing starts from a strong starting point and converges in a small number of steps. Reordering or skipping any stage breaks this chain: skipping calibration regresses to weight-space SVD, skipping covariance weighting leaves Stage 4's spectra incomparable, and skipping healing leaves a small but non-trivial residual that compounds across layers at low ranks.

3.3.1 Stage 1: Calibration Data Collection and Activation Capture

The first stage establishes the empirical foundation for activation-preserving factorization:

3.3.1.1 Calibration Dataset Preparation

CARE requires a representative calibration dataset to capture the input activation statistics of the target model. The calibration process involves:

- **Dataset Selection**: Choose 256 sequences from diverse sources (C4, WikiText2, PTB, Alpaca) to ensure representative coverage
- **Sequence Length**: The pipeline supports arbitrary lengths up to the host-memory limit of the covariance accumulator (about 512 tokens on H100 80GB; longer sequences trigger OOM during covariance estimation, see Sec. 3.4.5). Our experimental defaults are 32 tokens for zero-shot evaluation and 256 tokens when calibration is paired with healing; the sample-size and sequence-length sensitivity sweep is shown in Fig. 3.4
- **Domain Balance**: Include both general text and domain-specific content to match expected deployment scenarios
- **Preprocessing**: Apply the same tokenization and normalization as the original model training

3.3.1.2 Forward Pass Execution

For each calibration sequence, we perform forward passes through the original GQA model to collect layer-wise activations:

- **Activation Extraction**: Capture input activations $X^{(\ell)} \in \mathbb{R}^{T \times D}$ before each attention layer ℓ
- **Memory Management**: Store activations efficiently using mixed-precision (FP16) to reduce memory overhead
- **Batch Processing**: Process calibration data in batches to balance memory usage and computational efficiency
- **Gradient Computation**: Disable gradient computation during calibration to minimize memory requirements

3.3.2 Stage 2: Covariance Matrix Estimation and Regularization

The second stage computes activation covariance matrices and applies regularization for numerical stability:

3.3.2.1 Empirical Covariance Computation

For each layer ℓ, we compute the empirical covariance matrix:

$$C^{(\ell)} = \frac{1}{N} \sum_{b=1}^{N} (X_b^{(\ell)})^T X_b^{(\ell)} \tag{3.7}$$

where N is the number of calibration batches and $X_b^{(\ell)}$ represents the activations for batch b at layer ℓ.

3.3.2.2 Numerical Stabilization

To ensure numerical stability and invertibility, CARE applies shrinkage regularization [99]:

$$C_{\lambda}^{(\ell)} = (1 - \alpha)C^{(\ell)} + \alpha\lambda I \tag{3.8}$$

where $\alpha \in [0.01, 0.1]$ controls the shrinkage strength and λ is chosen as a small fraction of the largest eigenvalue of $C^{(\ell)}$. This prevents ill-conditioning while preserving the dominant covariance structure.

3.3.2.3 Covariance Matrix Properties

The regularized covariance matrices exhibit several important properties:

- **Positive Definiteness**: Guaranteed invertibility for downstream computations
- **Spectral Structure**: Preserves the relative magnitudes of principal components
- **Numerical Stability**: Condition numbers remain within acceptable ranges ($< 10^{12}$)
- **Computational Efficiency**: Sparse storage when appropriate for high-dimensional cases

3.3.3 Stage 3: Covariance-Weighted SVD Decomposition

The third stage performs the core factorization using activation-preserving SVD:

3.3.3.1 Matrix Preparation

For each layer ℓ and projection type $\{K, V\}$, we compute the covariance-weighted matrix:

$$M_{\{K,V\}}^{(\ell)} = \sqrt{C_{\lambda}^{(\ell)}}\, W_{\{K,V\}}^{(\ell)} \tag{3.9}$$

This weighting ensures that the SVD decomposition prioritizes directions with high activation variance, and using the shrinkage-regularized $\sqrt{C_{\lambda}^{(\ell)}}$ keeps the subsequent inverse $\sqrt{C_{\lambda}^{(\ell)}}^{-1}$ well-conditioned.

3.3.3.2 Randomized SVD Implementation

To handle large matrices efficiently, CARE employs randomized SVD [97] with the following optimizations:

- **Sketching**: Use Gaussian random matrices of size $D \times (r + p)$ where $p \approx 10$ is the oversampling parameter
- **Power Iteration**: Apply 2-3 power iterations to improve accuracy for matrices with slow spectral decay
- **Block Processing**: Process matrices in blocks to reduce peak memory usage
- **Numerical Precision**: Use FP32 precision for SVD computations to maintain accuracy

The decomposition yields:

$$M^{(\ell)}_{\{K,V\}} = U^{(\ell)}_{\{K,V\}} \Sigma^{(\ell)}_{\{K,V\}} (V^{(\ell)}_{\{K,V\}})^T \tag{3.10}$$

where $\Sigma^{(\ell)}_{\{K,V\}}$ contains the covariance-weighted singular values sorted in descending order.

3.3.4 Stage 4: Water-Filling Rank Allocation Algorithm

Algorithm 2 Water-Filling Rank Allocation

Require: Covariance matrices $\{C^{(\ell)}\}_{\ell=1}^{L}$, weight matrices $\{W_K^{(\ell)}, W_V^{(\ell)}\}_{\ell=1}^{L}$, total budget R_{tot}
Ensure: Optimal rank assignments $\{r_{\ell,K}, r_{\ell,V}\}_{\ell=1}^{L}$

1: Compute covariance-weighted SVD: $\sqrt{C_\lambda^{(\ell)}}\, W^{(\ell)}_{\{K,V\}} = U\Sigma V^T$ for all layers
2: Initialize all ranks to 1
3: Compute remaining budget: $B = R_{tot} - 2L$
4: **while** $B > 0$ **do**
5: Find matrix $(\ell, \{K, V\})$ with largest next singular value
6: Increment corresponding rank: $r_{\ell,\{K,V\}} \leftarrow r_{\ell,\{K,V\}} + 1$
7: Decrement budget: $B \leftarrow B - 1$
8: **end while**
9: **return** $\{r_{\ell,K}, r_{\ell,V}\}_{\ell=1}^{L}$

3.3.5 Stage 5: MLA Parameter Initialization

After determining optimal ranks, we initialize the MLA parameters using the covariance-weighted SVD factors with careful attention to numerical stability and initialization quality.

3.3.5.1 Low-Rank Factor Construction

For each layer ℓ and projection type $\{K, V\}$, we construct the MLA parameters:

$$W_a^{(\ell),\{K,V\}} \leftarrow \sqrt{C_\lambda^{(\ell)}}^{-1} U_{r_{\ell,\{K,V\}}}^{(\ell),\{K,V\}} \Sigma_{r_{\ell,\{K,V\}}}^{(\ell),\{K,V\}} \tag{3.11}$$

$$W_b^{(\ell),\{K,V\}} \leftarrow (V_{r_{\ell,\{K,V\}}}^{(\ell),\{K,V\}})^T \tag{3.12}$$

where $r_{\ell,\{K,V\}}$ is the allocated rank from the water-filling algorithm.

3.3.5.2 Initialization Quality Verification

To ensure high-quality initialization, CARE performs several verification steps:

- **Reconstruction Error Check**: Verify that $\|W_{\{K,V\}}^{(\ell)} - W_a^{(\ell),\{K,V\}} W_b^{(\ell),\{K,V\}}\|_F < \epsilon$ for appropriate tolerance ϵ
- **Activation Preservation**: Validate that $\|\sqrt{C^{(\ell)}}(W_{\{K,V\}}^{(\ell)} - W_a^{(\ell),\{K,V\}} W_b^{(\ell),\{K,V\}})\|_F$ is minimized
- **Numerical Stability**: Check condition numbers of the resulting matrices to ensure stable training
- **Parameter Scaling**: Apply appropriate initialization scaling to match the original parameter magnitudes

3.3.5.3 Memory Layout Optimization

The MLA parameters are organized for efficient inference:

- **Contiguous Storage**: Store W_a and W_b matrices in contiguous memory blocks
- **Cache-Friendly Layout**: Optimize data layout for efficient matrix multiplications during inference
- **Precision Selection**: Use mixed precision (FP16/BF16) where appropriate without compromising accuracy
- **Quantization Compatibility**: Ensure parameters are amenable to post-training quantization if required

3.3.6 Stage 6: Post-Conversion Healing

After initialization, CARE applies brief "healing" fine-tuning to close residual gaps between the original and converted models. This stage is crucial for recovering performance losses from the low-rank approximation.

Compatibility with 100% MLA restoration. The healing recipe described below operates on the latent K/V factors W^a, W^b produced by Stage 5 and is agnostic to whether positional information is supplied by re-using the original RoPE on the recovered keys or by adding a small decoupled RoPE channel for full *100% MLA restoration* as in TransMLA [100] to match DeepSeek architecture [53]. In the latter case, CARE simply replaces TransMLA's weight-space SVD initialization with the covariance-weighted $\sqrt{C}\,W$ SVD of Sec. 3.2.3 with rank scheduling, while keeping TransMLA's restoration and healing pipeline unchanged.

3.3.6.1 Healing Objective and Loss Functions

The healing process employs a multi-component loss function that combines cross-entropy and knowledge distillation [101]:

$$L_{CE} = -\frac{1}{T}\sum_{t=1}^{T} \log p^S(x_{t+1}|x_{\leq t}) \tag{3.13}$$

$$L_{KD} = \frac{1}{T}\sum_{t=1}^{T} \text{KL}(\text{softmax}(z_t^T/\tau)\|\text{softmax}(z_t^S/\tau)) \tag{3.14}$$

$$L = L_{CE} + \beta\tau^2 L_{KD} \tag{3.15}$$

where z_t^T and z_t^S are teacher (original) and student (converted) logits, $\tau = 4.0$ is the distillation temperature [101], and $\beta = 0.5$ balances the loss components.

3.3.6.2 Training Configuration and Hyperparameters

The healing fine-tuning uses carefully tuned hyperparameters to ensure stable and efficient recovery:

- **Learning Rate**: Start with 2×10^{-5} and use cosine annealing with warmup
- **Batch Size**: Use gradient accumulation to achieve effective batch size of 64 sequences
- **Optimizer**: AdamW with $\beta_1 = 0.9$, $\beta_2 = 0.95$, weight decay $= 0.1$, monitored via validation perplexity
- **Gradient Clipping**: Apply gradient norm clipping with threshold 1.0

3.3.6.3 Convergence Monitoring

The healing process includes comprehensive monitoring to ensure quality recovery:

- **Perplexity Tracking**: Monitor validation perplexity on held-out calibration data
- **KV-Cache Validation**: Verify that KV-cache outputs remain numerically consistent
- **Attention Pattern Analysis**: Compare attention patterns between original and converted models

3.3.7 Implementation Considerations and Computational Complexity

3.3.7.1 Computational Complexity Analysis

CARE's conversion process involves several computational stages with distinct complexity characteristics:

- **Calibration Data Processing**: $O(NTD)$ for forward passes through N sequences of length T with hidden dimension D
- **Covariance Estimation**: $O(N \cdot T \cdot D^2)$ for computing empirical covariance matrices across layers
- **Covariance-Weighted SVD**: $O(D \cdot n_h d_h \cdot r)$ per layer using randomized SVD with rank r
- **Water-Filling Allocation**: $O(L \cdot R_{tot} \cdot \log(R_{tot}))$ for optimal rank distribution across L layers
- **Parameter Initialization**: $O(L \cdot D \cdot r)$ for constructing MLA factors
- **Healing Fine-tuning**: $O(S \cdot T \cdot D \cdot r)$ for S training steps with reduced parameter count

The total conversion time scales approximately as $O(L \cdot D^2 \cdot N + L \cdot R_{tot}^2)$, making it practical for large models.

3.3.7.2 Memory Requirements

Memory usage during conversion involves several components:

- **Activation Storage**: $O(N \cdot T \cdot D)$ for calibration activations (can be processed in batches)
- **Covariance Matrices**: $O(L \cdot D^2)$ for storing layer-wise covariance matrices
- **SVD Intermediates**: $O(D \cdot r)$ for SVD computation workspace (temporary)
- **Original + Converted Parameters**: $2 \times O(L \cdot D \cdot n_h d_h)$ during transition period

3.3.7.3 Inference Complexity and Benefits

At inference time, MLA provides significant advantages:

- **KV-Cache Reduction**: Width reduces from $g_h d_h$ (GQA) to $r = g_h d_h$ (maintaining KV-parity)
- **Memory Bandwidth**: Substantially reduced due to smaller KV-cache footprint
- **Net Effect**: Memory bandwidth reduction typically outweighs FLOP increase for large models

3.3.7.4 Practical Implementation Tips

For successful CARE deployment, several practical considerations are important:

- **Calibration Data Quality**: Ensure calibration data matches the expected inference distribution
- **Numerical Stability**: Monitor condition numbers and apply appropriate regularization
- **Checkpointing**: Save intermediate results (covariance matrices, SVD factors) for debugging
- **Validation**: Always validate converted models on representative tasks before deployment
- **Hardware Optimization**: Leverage tensor cores and optimized BLAS libraries for matrix operations

3.4 Experimental Results

3.4.1 Experimental Setup

We evaluate CARE on Llama-3.1-8B conversion from GQA to MLA under KV-parity constraints. Our experimental configuration includes:

- **Model**: Llama-3.1-8B with GQA architecture (cross-model results on Qwen3-4B-Instruct-2507 are reported in Sec. 3.4.3)
- **Baselines**: Direct-SVD, Activation-aware SVD(ASVD) [96], TransMLA [100](MLA restoration healing comparison)
- **Tasks**: WikiText2 (perplexity), ARC-Challenge, ARC-Easy, HellaSwag, PIQA, MMLU, OpenBookQA, RACE, WinoGrande
- **Metrics**: Zero-shot accuracy (higher better) and perplexity (lower better)
- **Calibration**: 256 sequences from C4, WikiText2, PTB, and Alpaca datasets

Hardware, environment, and reproducibility. CARE has two phases with different resource profiles. Covariance computation, SVD factorisation, and all pre-healing experiments run on a single NVIDIA H100 80GB GPU paired with 2 TB of system RAM, with SVD/covariance routines kept in FP32 to preserve numerical accuracy. Post-conversion healing fine-tuning uses 8×H100 80GB GPUs with the Axolotl distributed training framework [102] in mixed precision (BF16). Zero-shot evaluation uses 256 calibration samples at sequence length 32 by default; healing uses 256 samples at sequence length 256. Healing token budgets sweep $\{1\text{B}, 3\text{B}\}$ to characterise recovery under matched supervision, with a per-step effective batch size of 64 sequences via gradient accumulation (Sec. 3.4.5).

3.4.2 Zero-Shot Performance Analysis

Table 3.1 shows zero-shot performance of CARE variants against baselines across different KV-cache compression ratios. We compare against naive weight-space SVD and the activation-aware ASVD baseline [96], which scales weights by per-channel activation magnitudes before SVD. CARE consistently outperforms both baselines, with improvements most pronounced at aggressive compression ratios.

Rank	KV Save	Method	Wiki(↓)	ARC(↑)	ARE(↑)	HS(↑)	PIQA(↑)	MMLU(↑)	OBQA(↑)	RA(↑)	WG(↑)	AVG(↑)
64	93.75	GQA (Original)	6.82	50.34	80.18	60.15	79.65	48.05	34.80	40.10	72.69	58.25
		SVD	1943.91	20.31	27.95	14.13	55.98	24.17	12.40	20.96	51.38	28.41
		ASVD [96]	2525.33	23.81	26.68	26.68	52.18	22.97	27.80	20.86	50.99	31.50
		CARE-E (Ours)	**576.21**	**19.45**	**32.95**	**26.35**	**57.02**	**23.60**	**27.40**	**21.53**	**50.20**	**32.31**
128	87.50	SVD	1486.43	20.48	27.36	16.07	54.19	21.53	14.00	21.34	50.36	28.17
		ASVD [96]	1675.54	25.00	27.99	26.61	52.39	23.04	26.60	21.82	48.46	31.49
		CARE-E (Ours)	**214.17**	**20.90**	**42.30**	**28.57**	**61.70**	**23.75**	**28.20**	**24.31**	**52.41**	**35.27**
256	75.00	SVD	626.93	18.94	30.77	15.54	56.04	23.38	12.40	22.39	49.64	28.64
		ASVD [96]	312.86	21.76	32.03	29.52	55.22	23.05	25.20	24.59	52.64	33.00
		CARE-E (Ours)	**39.57**	**30.29**	**60.14**	**39.05**	**71.16**	**34.61**	**33.60**	**30.53**	**61.17**	**45.07**
512	50.00	SVD	131.05	23.55	45.45	22.13	61.21	23.59	17.40	26.03	53.04	34.05
		ASVD [96]	12.02	46.33	69.11	70.75	76.17	41.80	36.60	33.97	66.85	55.20
		CARE-E (Ours)	**9.45**	**42.24**	**74.16**	**68.96**	**77.53**	**44.33**	**37.00**	**38.66**	**71.03**	**56.74**

TABLE 3.1: Zero-shot performance comparison on Llama-3.1-8B against naive weight-space SVD and the activation-aware ASVD [96] baseline. CARE-E delivers consistent improvements over both baselines at aggressive compression (rank 64/128/256). At rank 512, ASVD becomes competitive on some metrics, but CARE-E retains the lowest perplexity and the best aggregate ranking on most tasks.

3.4.3 Cross-Model Generalization on Qwen3-4B-Instruct-2507

To evaluate whether CARE's covariance-aware factorization and water-filling rank allocation generalize beyond Llama, we apply the same conversion pipeline to **Qwen3-4B-Instruct-2507**, using identical calibration settings (256 Alpaca samples, sequence length 32) and the same KV-parity rank budgets. Table 3.2 reports zero-shot performance against the unmodified Qwen3-4B GQA baseline, the naive weight-space SVD initialization, and the activation-aware ASVD [96] baseline, mirroring the comparison structure used for Llama-3.1-8B.

Rank	KV Save	Method	PPL(↓)	ARC(↑)	ARE(↑)	HS(↑)	PIQA(↑)	MMLU(↑)	OBQA(↑)	RA(↑)	WG(↑)	AVG(↑)
–	–	GQA (Original)	10.04	55.89	83.12	52.65	76.01	73.37	32.00	41.24	68.11	60.30
64	93.75	SVD	56922.11	25.34	26.77	25.73	50.44	24.31	28.40	22.11	50.91	31.75
		ASVD [96]	6683.95	27.39	25.00	25.71	50.38	24.08	26.60	22.11	50.43	31.46
		CARE-E (Ours)	**730.93**	**23.29**	**30.77**	**28.33**	**55.22**	**22.95**	**24.80**	**22.78**	**51.54**	**32.46**
128	87.50	SVD	22048.79	26.02	26.18	26.29	51.09	24.49	25.40	21.05	49.72	31.28
		ASVD [96]	1682.84	22.95	30.01	29.29	52.34	23.42	27.00	23.54	50.12	32.33
		CARE-E (Ours)	**102.38**	**30.29**	**44.82**	**42.54**	**63.76**	**30.52**	**29.40**	**28.71**	**54.54**	**40.57**
256	75.00	SVD	2561.97	26.11	29.46	29.92	52.88	24.48	28.40	23.64	51.70	33.32
		ASVD [96]	63.15	32.76	46.84	47.45	63.93	26.38	30.80	30.05	52.01	41.28
		CARE-E (Ours)	**28.84**	**41.30**	**59.22**	**56.53**	**69.37**	**53.50**	**35.20**	**32.82**	**61.88**	**51.23**
512	50.00	SVD	33.97	35.58	47.64	50.44	65.18	27.85	32.80	30.24	52.64	42.80
		ASVD [96]	15.49	47.61	66.54	67.49	73.01	56.56	35.60	35.22	62.75	55.60
		CARE-E (Ours)	**15.91**	**49.23**	**70.88**	**64.13**	**72.80**	**64.16**	**36.20**	**36.56**	**64.56**	**57.31**

TABLE 3.2: Zero-shot performance on Qwen3-4B-Instruct-2507 under matched KV-parity rank budgets.CARE-E substantially outperforms naive SVD at every rank budget and dominates ASVD at the more aggressive ranks (64/128/256); at rank 512, ASVD matches CARE-E on a couple of individual metrics, but CARE-E retains the strongest aggregate accuracy.

The Qwen3-4B results mirror the Llama-3.1-8B trends in Table 3.1: naive weight-space SVD fails catastrophically at low ranks (perplexity in the tens of thousands), and the activation-aware ASVD [96] variant only partially closes the gap, while CARE-E retains coherent perplexity and recovers a substantial fraction of the GQA accuracy across all eight evaluation tasks. The gap widens with rank: at rank 512, CARE-E reaches 57.31 average accuracy versus 42.80 for SVD and 55.60 for ASVD, recovering 95% of the unmodified GQA average (60.30) under the same KV-parity constraint. This consistency across two architectures of different scale ($\sim$4B and 8B parameters), tokenizers, and pretraining recipes indicates that the covariance-weighted SVD and adjusted-rank scheduling are properties of GQA decoder layers rather than artifacts of the Llama family.

3.4.4 Rank Distribution Analysis

Figure 3.3 shows the covariance-aware rank profiles across different calibration corpora. Both W_K and W_V exhibit a depth-dependent trend: ranks are smallest in early layers, grow steadily through middle blocks, and remain elevated thereafter. The growth is markedly stronger for W_V, while W_K increases more moderately. This pattern is consistent across different calibration datasets (Alpaca, WikiText2, PTB, C4), indicating model-intrinsic spectral properties.

3.4.5 Post-Healing Recovery Results

To assess fairness across baselines, we compare CARE against naive SVD [95] and TransMLA [100] under *matched* token budgets at MLA rank 512 on Llama-3.1-8B-Instruct (Table 3.3). Both TransMLA and our hybrid `TransMLA + CARE(E) Init` use TransMLA's full 100% MLA restoration framework (decoupled-RoPE channel and the same healing recipe); The naive SVD [95] baseline is also healed under the same per-step recipe but does not perform full MLA restoration. Calibration uses `alpaca-256-256` (256 samples, sequence length 256) for CARE-based methods and `wiki-256-256` for TransMLA; SVD [95] requires no

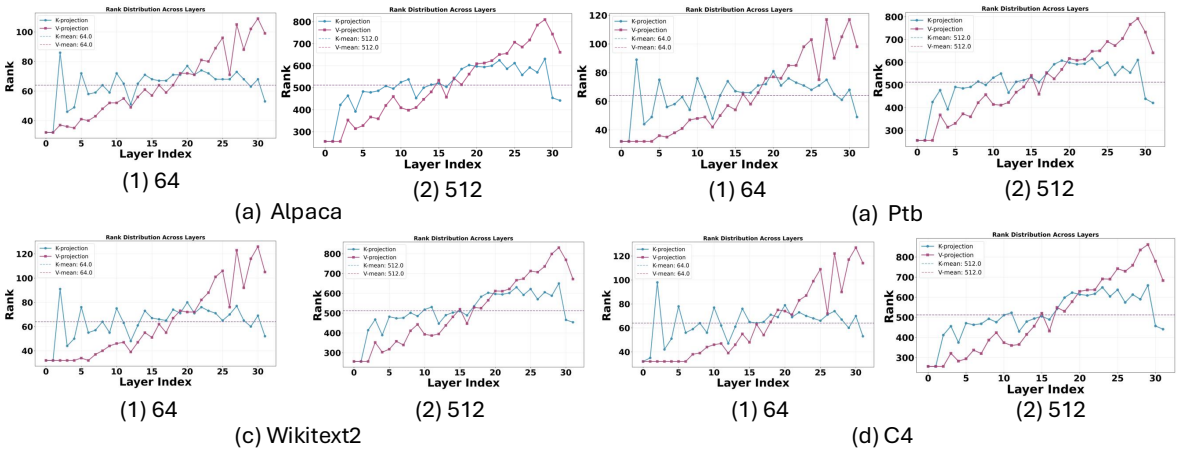


FIGURE 3.3: Visualization of adaptive rank distribution in Llama architecture achieved through CARE's water-filling optimization. The plot demonstrates how rank assignments vary across network depth according to spectral analysis, revealing that deeper layers tend to receive increased rank allocation for value transformation matrices (W_V) relative to key transformation matrices (W_K). This heterogeneous distribution pattern emerges from the algorithm's response to varying spectral characteristics across different layer positions and projection matrix types, ensuring efficient utilization of the constrained rank budget while maximizing preservation of model capabilities.

calibration corpus. We report healed checkpoints after 1B and 3B tokens on the LM Harness suite.

TABLE 3.3: Healed Llama-3.1-8B-Instruct comparison at MLA rank 512 across matched token budgets. TransMLA and `CARE(E) Init` share the same 100% MLA restoration and healing pipeline, so the gap between them isolates the contribution of CARE's activation-preserving initialization. Our `CARE(E) Init` rows are highlighted in bold.

Rank	Method	Calibration	Tokens	ARC(↑)	ARE(↑)	HS(↑)	PIQA(↑)	MMLU(↑)	OBQA(↑)	RA(↑)	WG(↑)	AVG(↑)
–	GQA (Original)	N/A	N/A	50.34	80.18	60.15	79.65	48.05	34.80	40.10	72.69	58.24
512	SVD [95]	N/A	1B	33.45	62.38	45.55	72.34	50.57	23.00	31.54	61.78	47.58
	SVD [95]	N/A	3B	44.56	74.96	52.20	76.63	61.08	30.40	45.15	65.42	56.30
	TransMLA	wiki-256-256	1B	53.04	81.07	58.75	81.04	69.13	32.00	44.09	71.74	61.36
	TransMLA	wiki-256-256	3B	53.77	82.34	56.44	80.70	70.23	33.30	45.61	72.47	61.86
	CARE(E) Init (Ours)	alpaca-256-256	1B	**52.25**	**82.33**	**62.47**	**80.21**	**70.31**	**32.90**	**45.11**	**75.13**	**62.59**
	CARE(E) Init (Ours)	alpaca-256-256	3B	**51.75**	**80.73**	**64.45**	**83.23**	**71.57**	**34.00**	**46.33**	**74.09**	**63.27**

The healed comparison resolves two fairness questions in a single table. First, against naive SVD [95] under the same per-step healing recipe, the gap remains substantial even after 3B tokens (56.30 vs. 63.27 average accuracy), confirming that no amount of healing can fully compensate for a poor weight-space initialization. Second, isolating CARE's contribution within the *same* 100% MLA restoration framework as TransMLA: both methods share the

decoupled-RoPE channel and identical healing recipe, so the gap between them directly attributes to the initial KV factorization. `CARE(E) Init` reaches 62.59 and 63.27 average accuracy at matched 1B and 3B token budgets, versus 61.36 and 61.86 for TransMLA — a +0.7 to +1.2 point lift attributable purely to the activation-preserving and rank-aware initialization.

3.4.6 Ablation Studies

3.4.6.1 Impact of Calibration Dataset

We evaluate how different calibration corpora affect zero-shot accuracy. Table 3.4 reports the three corpora used in the deployed defaults (Alpaca, C4, WikiText2). Despite noticeable perplexity differences, accuracy effects are small and consistent: per-task scores vary by 1-10% points across corpora, while average scores shift by only 1-2% points.

TABLE 3.4: Calibration Dataset Impact on CARE-E Performance

Rank	Calibration	Wiki(↓)	ARC(↑)	MMLU(↑)	WG(↑)	AVG(↑)
128	ALPACA	214.17	20.90	23.75	52.41	33.54
	C4	126.94	18.60	23.46	52.72	33.21
	WikiText2	40.37	17.92	23.46	52.01	31.26
512	ALPACA	9.45	42.24	44.33	71.03	54.17
	C4	9.09	43.26	40.18	71.03	53.93
	WikiText2	7.91	41.81	36.97	71.19	52.22

Across Alpaca, C4, WikiText2, and PTB, average accuracy changes are modest, though task-aligned corpora can give local gains, indicating that CARE's covariance estimate is robust to moderate domain mismatch in the calibration set.

Sensitivity to calibration sample count and sequence length. Beyond the choice of corpus, two further calibration knobs matter for reproducibility: how many sequences are observed and how long each sequence is. Figure 3.4 sweeps both knobs across the eight LM Harness benchmarks under fixed CARE-E settings. Zero-shot accuracy saturates beyond approximately 512 calibration samples on every benchmark, and longer sequences begin to overfit the limited calibration set rather than improve generalization—we observe out-of-memory failures during

covariance estimation past sequence length 512. We therefore use 256 samples at sequence length 32 as the default for zero-shot reporting, and sequence length 256 only when calibration is paired with healing (Sec. 3.4.5).

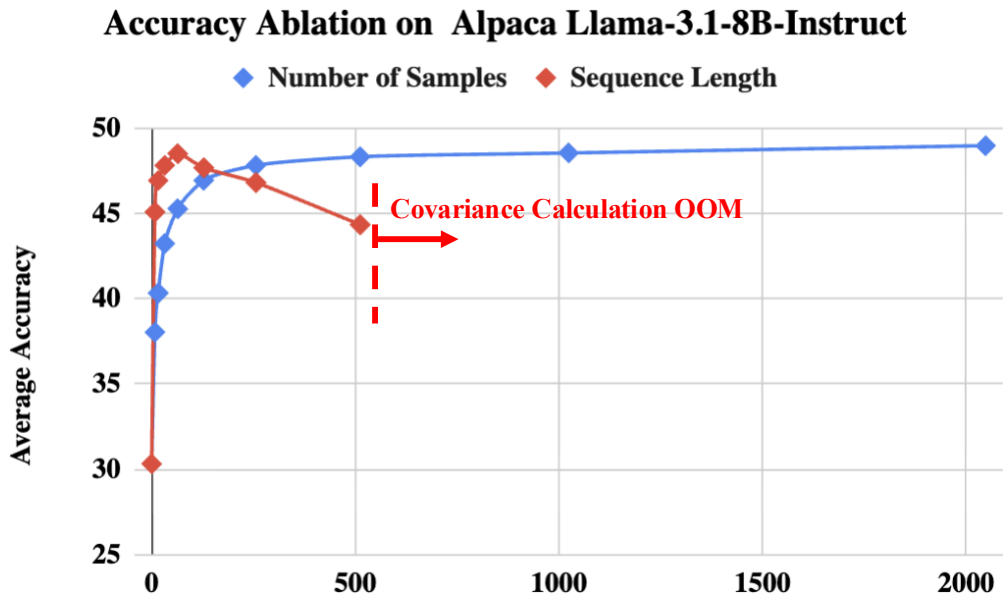


FIGURE 3.4: Zero-shot accuracy versus calibration sample count (varying samples at fixed sequence length 256) and sequence length (red curve, fixed 256 samples) across eight LM Harness benchmarks for Llama-3.1-8B-Instruct. CARE saturates beyond ∼512 samples and longer sequences overfit the limited calibration set; OOM occurs beyond sequence length 512 during covariance computation.

3.4.6.2 Uniform vs. Energy-Based Rank Allocation

We compare CARE's energy-based rank allocation (CARE-E) against uniform allocation (CARE-U). Energy-based allocation consistently outperforms uniform allocation, with larger improvements at aggressive compression ratios where rank allocation becomes more critical.

3.4.7 Analysis and Key Findings

3.4.7.1 Performance Characteristics

CARE demonstrates several key advantages over baseline methods:

TABLE 3.5: Rank Allocation Strategy Comparison

Rank	Strategy	Wiki(↓)	ARC(↑)	PIQA(↑)	AVG(↑)
64	CARE-U	693.82	18.00	55.50	29.90
	CARE-E	576.21	19.45	57.02	30.64
256	CARE-U	63.23	24.49	67.41	38.78
	CARE-E	39.57	30.29	71.16	43.34
512	CARE-U	12.15	41.30	76.66	52.23
	CARE-E	9.45	42.24	77.53	54.17

- **Superior zero-shot performance**: Up to 331% relative improvement over naive weight-space SVD at aggressive compression (rank 64), and consistent gains over the activation-aware ASVD [96] baseline across rank 64/128/256
- **Consistent improvements**: Performance gains across all compression ratios and tasks
- **Efficient recovery**: Faster convergence during healing with restricted SFT budgets
- **Robust rank allocation**: Energy-based allocation outperforms uniform allocation, especially at low ranks

3.4.7.2 Computational Considerations

CARE's conversion costs are front-loaded during initialization:

- Covariance estimation requires modest calibration data (256 sequences)
- Water-filling allocation is computationally efficient $O(L \cdot R_{tot})$
- Post-conversion healing requires fewer training steps than baselines
- Inference maintains MLA's efficiency advantages with identical KV-cache footprint

3.5 Discussion

3.5.1 Key Contributions and Insights

Our experimental evaluation reveals several critical insights about MLA conversion:

(1) **Activation-Awareness Matters**: Covariance-weighted SVD substantially outperforms naive weight-space SVD, with perplexity improvements of 5-50x across compression ratios

(2) **Adaptive Rank Allocation**: Energy-based rank distribution consistently outperforms uniform allocation, with larger gains at aggressive compression where allocation becomes critical

(3) **Layer Heterogeneity**: Different layers exhibit distinct spectral properties, with value projections requiring higher ranks than key projections in deeper layers

(4) **Robust Recovery**: CARE provides superior initialization for post-conversion healing, requiring fewer SFT steps to recover original performance

3.5.2 Comparison with Existing Methods

CARE advances MLA conversion by addressing fundamental limitations of existing approaches:

- **vs. naive SVD**: In the zero-shot comparison (Table 3.1), CARE-E delivers 15–90% relative per-task improvements at rank 128/256/512 while maintaining identical KV-cache footprint; activation-preserving factorization reduces attention drift, leading to more stable conversion
- **vs. ASVD [96]**: Both methods are activation-aware, but ASVD only rescales weights by per-channel activation magnitudes before standard SVD, whereas CARE solves the full activation-space objective $\|\sqrt{C}(W - \hat{W})\|_F^2$ via covariance-weighted SVD. CARE dominates at aggressive compression (rank 64/128/256) where the per-channel approximation breaks down; the gap narrows at rank 512 where most of the spectrum is preserved
- **vs. TransMLA (100% MLA restoration)**: Within the same restoration framework and matched 1B/3B-token healing budgets, swapping TransMLA's weight-space SVD initialization for CARE's covariance-weighted SVD lifts average accuracy by +0.7 to +1.2 points (Table 3.3), isolating the contribution of activation-preserving initialization

- **vs. Energy-based baselines**: Water-filling allocation optimally distributes limited rank budget across layers

3.5.3 Practical Deployment Insights

CARE offers practical advantages for deploying efficient attention mechanisms:

- **KV-Cache Efficiency**: Maintains identical memory footprint to original GQA while improving expressivity through latent compression
- **Conversion Simplicity**: One-time conversion process with minimal calibration data requirements (256 sequences)
- **Model Agnostic**: Principles apply to any GQA architecture, demonstrated on Llama family
- **Scalable Implementation**: Conversion costs scale linearly with model size, enabling application to larger models

3.6 Conclusion

CARE introduces a principled approach to converting pretrained GQA models into efficient MLA architectures under fixed KV-cache budgets. By addressing fundamental limitations of naive SVD through activation-preserving factorization and adaptive rank allocation, CARE achieves superior conversion quality while maintaining the memory and bandwidth advantages of MLA.

The experimental results demonstrate that activation-awareness is crucial for preserving attention fidelity during architectural conversion. CARE's covariance-weighted SVD substantially reduces attention drift compared to weight-only approaches, while water-filling rank allocation optimally distributes limited capacity across layers with heterogeneous spectral properties.

Key achievements include up to 331% relative improvement in zero-shot accuracy over baseline methods before finetune, consistent performance gains across compression ratios, and efficient recovery through brief healing fine-tuning. The method's model-agnostic design and scalable implementation make it applicable to various transformer architectures and model scales.

Future directions include extending CARE to full MHA-to-MLA conversion [53], investigating dynamic rank adjustment during inference, and combining covariance-aware techniques with other architectural optimizations. CARE's success suggests that activation-aware compression principles could benefit other model conversion tasks, potentially enabling more efficient deployment of large language models in resource-constrained environments.

CHAPTER 4

Infinigram-Based Speculative Decoding for Accelerated Inference

---

This chapter presents a novel approach to accelerating large language model inference through Infinigram-based speculative decoding [92]. By leveraging massive n-gram statistics computed from large text corpora, this method achieves significant speedup in autoregressive generation while maintaining the exact output distribution of the target model. Our approach considers training the Infinigram speculator with a large number of target model outputs, which provides additional performance gains by ensuring distribution alignment between the draft mechanism and target model behavior. We detail the theoretical foundations, implementation methodology, and comprehensive experimental evaluation comparing different approaches to building and utilizing n-gram indices for speculation.

4.1 Introduction and Motivation

The autoregressive nature of large language model generation presents a fundamental bottleneck for inference speed. Each token must be generated sequentially, with the model processing the entire context to produce a single token at each step. This sequential dependency limits parallelization opportunities and results in significant latency, particularly for longer sequences.

Speculative decoding [62, 82] addresses this challenge by generating multiple tokens in parallel using a faster approximation method, then verifying these tokens with the target model. Traditional approaches employ smaller neural networks as draft models [82], but these still require substantial computation and careful training to match the target distribution.

4.1.1 Infinigram for Speculative Decoding

We adapt the Infinigram architecture [92] as a speculative decoder for large language model inference. While the original Infinigram system focuses on standalone language modeling, our contribution lies in integrating its unbounded n-gram capabilities into the speculative decoding framework. This adaptation replaces computationally expensive neural draft models with statistical lookups, achieving several key advantages:

- **Unbounded n-gram prediction**: Supports dynamic lengths and extremely long context matches beyond traditional n-gram limits
- **Lightning-fast prediction**: Instantly generates lookahead tokens without any neural network forward passes - achieving microsecond-level speculation latency
- **Training-free deployment**: No complex training procedures, gradient descent, or hyperparameter tuning required - simple statistical counting suffices
- **CPU-optimized efficiency**: Fully utilizes available CPU resources with minimal GPU dependency, enabling deployment in resource-constrained environments
- **Scalability**: Results of Infinigram models are additive, which enables the deployment of extremely large models by sharding the n-gram index across multiple CPU nodes
- **Perfect recall on familiar patterns**: Excels when questions or contexts have been encountered before, leveraging exact pattern matching
- **Interpretability**: Retrieval of statistical results including match length and frequency provides insights into model behavior, enable us to compare performance differences between various training data sources and target model

4.2 Theoretical Foundations of Infinigram-Based Speculation

4.2.1 From Unbounded N-grams to Speculative Decoding

The theoretical foundation of our approach rests on the observation that many token sequences in natural language and especially in structured domains like code exhibit strong local patterns that can be captured through n-gram statistics. While neural language models excel at understanding complex semantic relationships, a significant portion of their predictions involve recovering common patterns that appear frequently in training data.

Infinigram's unbounded context matching provides a unique advantage: when the model encounters a sequence it has seen before, it can leverage the full historical context to make near-perfect predictions. This property is particularly valuable for:

- **Code generation**: Programming languages have highly regular syntax with frequently repeated patterns
- **Technical writing**: Scientific and technical text often contains formulaic expressions and domain-specific terminology
- **Structured data**: JSON, XML, and other structured formats follow rigid patterns
- **Conversational patterns**: Common dialogue patterns and responses appear frequently across conversations

4.2.2 Statistical Foundations and Probability Estimation

Our adaptation of Infinigram for speculative decoding involves sophisticated probability estimation beyond simple frequency counting. Given a context $c = (w_1, w_2, ..., w_n)$, we estimate the probability of the next token w_{n+1} through:

$$P_{\text{infinigram}}(w_{n+1}|c) = \begin{cases} \frac{\text{count}(c,w_{n+1})}{\text{count}(c)} & \text{if match found} \\ \text{backoff}(w_{n+1}|c_{\text{shorter}}) & \text{otherwise} \end{cases} \tag{4.1}$$

where $c[1:]$ denotes the context with the first token removed. This recursive backoff ensures valid probability distributions by progressively shortening the context until a match is found, ultimately falling back to unigram probabilities when no context matches exist.

Smoothing. Within a matched context, we use the raw maximum-likelihood frequency of the next token—no Kneser–Ney, Witten–Bell, or interpolated smoothing is applied. The only smoothing mechanism is the strict length backoff above, which progressively shortens the suffix to the longest one that still matches the index. This choice is deliberate: classical n-gram smoothing methods are designed to assign mass to truly unseen $(n-1)$-gram contexts, but the suffix-array index already exposes the longest matched suffix in $O(L \log N)$ time, so unseen-context handling is delegated to the backoff length rather than to a smoothing prior.

4.3 Infinigram Architecture and Implementation

4.3.1 N-gram Index Construction

The foundation of Infinigram-based speculative decoding is a comprehensive n-gram index built from text corpora. The construction process involves:

4.3.1.1 Data Processing Pipeline

(1) **Tokenization**: Convert raw text to token sequences using the target model's tokenizer, save with u32 to ensure 128K vocabulary compatibility
(2) **Index Building**: Construct suffix array for microsecond-level lookup performance

Computational Advantages: This process requires only $\sim$840 seconds for 5 million examples on standard CPU hardware, demonstrating exceptional data efficiency compared to neural training that typically requires thousands of GPU hours for comparable datasets.

4.3.1.2 Storage and Retrieval Optimization

The index employs several optimizations for efficient storage and retrieval:

- **Suffix Arrays**: Given corpus size N tokens, prefix length L, and speculation depth k: lookup complexity is $O(L{\cdot}\log N)$ for finding matches, plus $O(\min(K, \text{max\_support}){\cdot}\ k)$ for extracting the next k tokens from K matching positions [92]
- **Compressed Storage**: Variable-length encoding for counts [103] - efficient memory utilization
- **Distributed CPU Processing**: Sharding enables parallel processing across multiple CPU cores without GPU dependencies
- **Memory Efficiency**: Index fits in standard RAM with consistent memory, eliminating need for expensive GPU memory

4.3.2 Speculative Token Generation Process

4.3.2.1 Optimized Multi-Token Retrieval

A critical optimization in our Infinigram implementation is the `inf_nkt` function, which retrieves k tokens in a single operation instead of making k sequential calls to `infgram_ntd` to predict the next k tokens' distributions. This optimization reduces the time complexity from $O(L \cdot k \cdot \log n)$ to $O(L \cdot \log n + \text{max\_support} \cdot k)$, where the $\log n$ term typically dominates. L is the length of the prefix and max_support is the downsampling factor limiting the number of matched sequences considered.

Key Insight: When calling `infgram_ntd` sequentially, each call extends the suffix by exactly one token. The `inf_nkt` function simulates this behavior efficiently by:

(1) Finding the optimal suffix length with length L using binary search: $O(L \cdot \log n)$
(2) Retrieving all matches for the initial suffix: $O(1)$ index access
(3) Extracting all k-token sequences from matches: $O(\text{max\_support} \cdot k)$
(4) Simulating sequential token selection through filtering: $O(\text{max\_support} \cdot k)$

This reduces the number of index lookups from k to 1, providing a 25x speedup (25ms $\rightarrow$ 1ms) while maintaining identical output distributions.

The `max_support` hyperparameter. The constant max_support in the complexity expressions above is a downsampling threshold in the C++ backend of the suffix-array engine: when a matched suffix has more than max_support occurrences in the corpus, the engine uniformly subsamples down to max_support matches before extracting next-token statistics. We use the engine default max_support $= 1000$ throughout this work. This value provides sufficient statistical support for next-token distribution estimation on high-frequency suffixes (where additional matches yield diminishing information) while keeping the per-extraction cost $O(\text{max\_support} \cdot k)$ bounded and predictable. Increasing max_support admits more matched sequences per lookup, marginally improving statistics on highly frequent suffixes at the cost of longer extraction; decreasing it only marginally reduces extraction time but can introduce noise on low-frequency tails.

Implementation Details: The `inf_nkt` function implements a sophisticated simulation strategy. After finding the optimal suffix and retrieving all matching sequences, it iterates through each token position (0 to k-1), counts token frequencies at that position across all active sequences, selects the most frequent token (maintaining nkt() tie-breaking behavior), and filters out sequences that don't contain the selected token. This perfectly replicates the sequential behavior of calling `infgram_ntd` multiple times, but with dramatically improved performance.

Algorithm 3 Optimized Infinigram Speculative Decoding

Require: Context tokens $c = (t_1, \ldots, t_k)$, target model M_{target}, speculation depth d
Ensure: Generated tokens with acceptance decisions

1: **Single optimized call:** $\{t_1, t_2, \ldots, t_d\}, \{P_1, P_2, \ldots, P_d\} =$ `inf_nkt`$(c, d, \text{max\_support})$
2: Initialize speculation buffer with max conditional probabilities $\mathcal{S} = [t_1, t_2, \ldots, t_d]$
3: Store n-gram probabilities $\{P_{ng}(t_i)\} = \{P_1, P_2, \ldots, P_d\}$
4: Compute target model probabilities for all speculated tokens
5: **return** Accepted tokens and rejection point

4.3.3 Token Evaluation Mechanism

4.3.3.1 Acceptance Rate Metric

To evaluate the quality of Infinigram's predictions on next-token distribution, we employ an acceptance rate metric based on the statistical distance between probability distributions[83]. The acceptance rate is computed as:

$$\text{Acceptance Rate} = 1 - \frac{1}{2}||P_{target} - P_{infinigram}||_1 \tag{4.2}$$

where $||\cdot||_1$ denotes the L1 (total variation) distance between the target model's probability distribution P_{target} and Infinigram's predicted distribution $P_{infinigram}$ over the vocabulary.

This metric measures the similarity between the two distributions: a value of 1 indicates perfect alignment (identical distributions), while 0 indicates complete disagreement. Intuitively, it quantifies how closely Infinigram's predictions match the target model's token preferences, directly correlating with the likelihood of acceptance during speculative decoding. The acceptance rate effectively captures the expected probability that a token sampled from Infinigram would be accepted by the target model.

4.3.3.2 Token-Level Acceptance Criterion

During actual speculative decoding, we use greedy rejection sampling to not only ensure the target distribution is exactly preserved, but also make sure the generated output aligns for each run. We measure accepted length during speculative decoding, which is averaged over all decoding steps.

4.4 Experimental Methodology

4.4.1 Comparative Evaluation Design

Our experimental evaluation employs a novel comparative approach to assess the impact of training data distribution on speculative decoding performance. We construct two distinct Infinigram indices to understand how data source affects acceptance rates and overall speedup.

4.4.1.1 Approach 1: SFT Response-Based Speculation (Original Dataset)

This approach uses GPT [4] responses from supervised fine-tuning datasets:

- **Data Source**: 5 million SFT examples with gpt-generated responses
- **Index Construction**: Build n-gram statistics from original SFT responses
- **Hypothesis**: GPT [4] text patterns provide baseline speculation capability
- **Testing Protocol**: Evaluate against both SFT responses and target model outputs
- **Practical Relevance**: Represents scenarios where only training data is available

4.4.1.2 Approach 2: Target Model Response-Based Speculation

This approach uses responses generated by the target model itself:

- **Data Generation**: 5 million SFT prompts generated by Llama 3.2-8B-Instruct [8] (one-time cost).
- **Index Construction**: Build n-gram statistics from model-generated responses
- **Hypothesis**: Target model generated patterns should better match inference behavior
- **Testing Protocol**: Evaluate against target model outputs for distribution alignment
- **Resource Efficiency**: One-time CPU-based index construction enables unlimited fast predictions

4.4.1.3 Confidence-Based Filtering

Both approaches are evaluated with probability threshold filtering:

- **Threshold Values**: [0.5, 1.0], [0.6, 1.0], [0.7, 1.0] for top-1 token probability
- **Purpose**: Analyze performance when model confidence is high vs. uniform distributions
- **Hypothesis**: Higher confidence tokens should yield better acceptance rates

4.4.1.4 Domain-Specific Evaluation

- **Python Coding Subset**: Extract Python programming questions from evaluation set
- **Purpose**: Assess domain-specific performance vs. general conversation
- **Hypothesis**: Structured code patterns should improve speculation accuracy [104]

4.4.1.5 Comparison with Neural Speculative Decoding

To establish Infinigram's competitive position, we conduct comprehensive comparisons with state-of-the-art neural speculative decoding models:

- **Baseline Models**: Llama-3.1-8B-Instruct (target model without speculation), Eagle-3 [105] (leading neural speculative decoding approach), and Infinigram Target-5M (our approach using target model-based n-gram index)
- **Evaluation Benchmarks**:
  - **HumanEval** [10]: Industry-standard coding benchmark with 164 programming problems, each with function signature, docstring, and solution. Tests practical coding ability across diverse algorithmic challenges.
  - **SFT Python Coding**: Python programming questions filtered from our SFT dataset, representing more conversational coding interactions
- **Evaluation Protocol**: Single-batch inference (batch size = 1) for fair comparison of speculation effectiveness
- **Metrics**: Acceptance length, inference speed (tokens/second), and speedup factor relative to baseline
- **Hardware Configuration**: Consistent evaluation environment to ensure comparable timing measurements

4.4.1.6 Cross-Model Scalability

To assess whether Infinigram-based speculative decoding transfers beyond the Llama-3.1-8B-Instruct target used in the preceding evaluations, we re-run the full pipeline end-to-end on two Qwen3 targets [106] that span a $4\times$ change in parameter count:

- **Target Models**: **Qwen3-8B** and **Qwen3-32B**, evaluated against the corresponding non-speculative baseline on the same hardware (Sec. 4.4.2)
- **Per-Target Index Construction**: for each target, we regenerate 5M responses from that target model using the same data-generation procedure as for Llama-3.1-8B, then build a fresh target-aligned suffix-array index.
- **Held Constant Across Targets**: the data-generation and index-building procedure, the `inf_nkt` retrieval (Sec. 4.3.2.1), and the rejection-sampling acceptance rule
- **Evaluation Benchmarks**: Math500 (mathematical reasoning), HumanEval [10] (code generation), and Alpaca [107] (instruction following), chosen to span three distinct task styles
- **Metrics**: speculation throughput (tokens/second), end-to-end speedup over the non-speculative baseline, and average accepted length
- **Hypothesis**: Infinigram model can learn the distribution from all kinds of target models. Speedup ratio is also consistent across models benefiting from the negalible speculation latency from infinigram index retrieval.

4.4.2 Hardware, Environment, and Build-vs-Training Cost

Hardware and environment. The Infinigram pipeline has three phases with different resource profiles. (i) *Target-output generation:* we generate 5M responses from the target model with max new tokens set to 512 (Llama-3.1-8B-Instruct, and additionally Qwen3-8B and Qwen3-32B for the cross-model evaluation in Sec. 4.5.6) on $8\times$H100 80GB GPUs, taking approximately 34 hours of total GPU time for the 5M-prompt sweep at the per-target level(48 hours for Qwen3-32B). (ii) *Index construction:* suffix-array building runs entirely on an Intel Xeon Platinum 8468V CPU; the full 5M-example index builds in $\sim$840 s wall-clock with

no GPU involvement. (iii) *Speculative inference:* the `inf_nkt` retrieval and the rejection-sampling acceptance loop run on the same Intel Xeon Platinum 8468V CPU against the memory-mapped 33 GB index, while the target-model forward passes during decoding run on a single H100 80GB GPU.

Build cost vs. neural-draft training cost. Infinigram speculative deocoding and neural draft models shared the same data regeneration phase: state-of-the-art neural drafts such as Eagle-3[88] also require generating target-model outputs to align the draft distribution with the target's actual generation behavior, exactly as we do for the target-aligned Infinigram index discussed in Sec. 4.6.4. The two pipelines diverge only in what happens *after* that data is collected. Eagle-3-style approaches must additionally train a small neural draft model on the regenerated outputs. This neural-draft training stage is the dominant marginal cost of adopting Eagle-3 per new target model, which takes days for model to converge [108]. Infinigram replaces this entire neural-training stage with an $\sim$840 s single-CPU suffix-array construction — two to three orders of magnitude shorter wall-clock and with no GPU dependency.

4.5 Experimental Results

4.5.1 Acceptance Rate Analysis

The core determinant of speculative decoding effectiveness is the acceptance rate of speculated tokens. Our comparative analysis reveals significant differences between the two approaches.

TABLE 4.1: SFT-Based Infinigram Acceptance Rates by Training Data Size (Original Approach)

Model	**Acceptance Rate by Minimum Match Length (Tested on SFT Responses)**									
	0	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**Build**
Infini-50k	32.85%	35.00%	45.37%	56.11%	64.45%	-	-	-	-	-
Infini-100k	34.41%	36.22%	44.91%	55.31%	62.63%	-	-	-	-	-
Infini-500k	39.56%	40.80%	47.87%	56.51%	64.13%	-	-	-	-	-
Infini-1M	41.17%	42.19%	48.14%	56.41%	64.05%	-	-	-	-	-
Infini-2.5M	43.43%	42.19%	48.14%	56.45%	63.52%	68.14%	71.53%	73.41%	75.01%	-
Infini-5M	**44.89%**	**45.61%**	**50.07%**	**56.97%**	**63.52%**	**67.93%**	**71.02%**	**73.35%**	**74.87%**	**840s**

Note: This table shows the original results using SFT-based indices evaluated on SFT responses, representing the baseline approach without target model adaptation. We leave

some results blank as small infinigram models do not provide enough tokens for evaluating those match lengths.

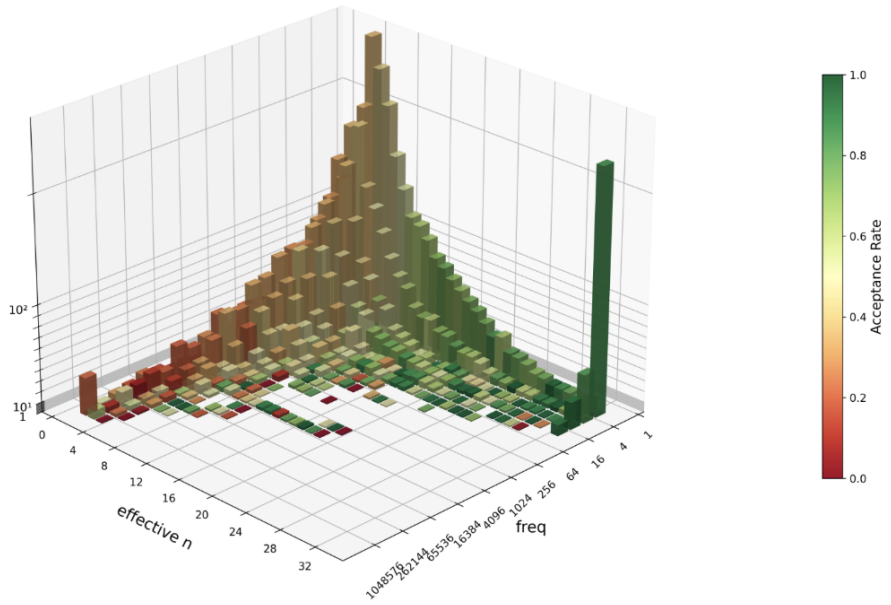


FIGURE 4.1: Three-dimensional visualization of Infinigram SFT-5M performance characteristics. The plot shows the relationship between effective n-gram length (0-32), token frequency (logarithmic scale), and acceptance rate (0.0-1.0, color-coded). The surface reveals how acceptance rate increases with both longer context matches and higher frequency patterns, with peak performance (green regions) achieved when both factors align. The exponential decay in frequency at longer n-gram lengths illustrates the sparsity challenge in n-gram modeling.

The visualization in Figure 4.1 provides crucial insights into Infinigram's operational characteristics:

- **Frequency-Performance Correlation**: High-frequency patterns (peaks in the surface) consistently achieve better acceptance rates, validating the n-gram approach for common sequences.
- **Context Length Impact**: The gradual elevation from n=0 to n=16 demonstrates how longer context matches improve prediction accuracy, though with diminishing returns.

- **Sparsity at Scale**: The dramatic frequency drop-off at longer n-gram lengths (visible as the narrowing pyramid shape) highlights why even 5M training examples struggle to provide coverage for long contexts.
- **Optimal Operating Region**: The green-yellow band (0.6-0.8 acceptance rate) concentrated in the mid-range contexts (n=8-16) with moderate frequencies identifies the sweet spot for practical deployment.

4.5.1.1 Comparative Analysis: SFT vs Target Model Approaches

The comprehensive evaluation reveals significant differences between approaches and testing conditions:

TABLE 4.2: Comparative Acceptance Rates: SFT-Based vs Target Model-Based Infinigram

Configuration	ML 0	ML 2	ML 4	ML 6	ML 8	ML 10	ML 12	ML 14	ML 16
General Prompts (All Domains)									
SFT-5M	39.72%	40.06%	43.76%	49.47%	57.05%	64.02%	69.27%	71.72%	74.85%
Target-5M	45.75%	46.16%	50.26%	55.82%	61.98%	66.77%	71.92%	74.55%	76.84%
Python Coding Prompts (Domain-Specific)									
SFT-5M (Python)	56.01%	56.49%	60.52%	65.74%	70.33%	74.18%	77.30%	79.70%	82.08%
Target-5M (Python)	62.54%	63.04%	67.20%	71.99%	75.95%	79.38%	81.94%	83.84%	85.38%

4.5.1.2 Key Finding 1: Training Data Distribution Overfitting

The comparative evaluation reveals a critical insight about distribution alignment in speculative decoding:

Overfitting to Training Distribution: When the SFT-based Infinigram model (trained on original responses) is tested on target model outputs instead of SFT responses, performance degrades significantly. This indicates that the SFT-based index has overfit to teacher models (e.g., GPT-4) that have fundamentally different generation patterns than the deployed target models.

Target Model Alignment: Conversely, the target model-based Infinigram (trained on target model outputs) achieves consistently higher acceptance rates when tested on target model outputs, demonstrating better distribution alignment. This suggests that for optimal speculative

decoding performance, the draft mechanism should be trained on data that matches the target model's distribution.

4.5.1.3 Key Finding 2: Domain-Specific Performance Gains

Python Coding Excellence - Leveraging Familiar Patterns: Both approaches show dramatically improved performance on Python coding tasks, demonstrating Infinigram's strength in recognizing previously seen patterns:

- SFT-based: 56.01% $\rightarrow$ 82.08% average acceptance
- Target-based: 62.54% $\rightarrow$ 85.38% average acceptance
- **Pattern Familiarity Effect**: Infinigram excels when coding patterns have been encountered during index construction
- **Instant Recognition**: Previously seen code snippets, function patterns, and syntax structures are recognized immediately without computation delay

Code Structure Benefits: The superior performance on coding tasks demonstrates Infinigram's core advantages:

(1) **Deterministic Patterns**: Programming languages have more rigid syntactic patterns that are perfectly suited for n-gram matching
(2) **Repetitive Structures**: Function calls, variable assignments, and control structures are highly predictable and frequently repeated
(3) **Scalable Recognition**: Domain-specific indices built with just 5M examples achieve performance comparable to much larger neural models

4.5.2 Inference Speed Analysis

4.5.2.1 Confidence Threshold Analysis

A critical finding emerges from analyzing performance across different confidence thresholds:

TABLE 4.3: Acceptance Rate vs Confidence Threshold Across All Minimum Match Lengths

Configuration	Threshold	ML 0	ML 2	ML 4	ML 6	ML 8	ML 10	ML 12	ML 14	ML 16
General Prompts (All Domains)										
Target-5M	All Tokens	45.75%	46.16%	50.26%	55.82%	61.98%	66.77%	71.92%	74.55%	76.84%
	Conf > 0.5	60.65%	60.64%	62.02%	64.85%	68.56%	71.81%	75.89%	77.77%	80.09%
	Conf > 0.6	62.20%	62.19%	63.20%	65.82%	69.28%	72.43%	76.33%	78.19%	80.59%
	Conf > 0.7	64.46%	64.44%	65.24%	67.47%	70.48%	73.49%	77.33%	79.11%	81.13%
Python Coding Prompts (Domain-Specific)										
Target-5M (Python)	All Tokens	62.54%	63.04%	67.20%	71.99%	75.95%	79.38%	81.94%	83.84%	85.38%
	Conf > 0.5	73.94%	73.99%	75.59%	78.41%	80.63%	82.73%	84.58%	86.09%	87.24%
	Conf > 0.6	75.39%	75.42%	76.76%	79.36%	81.36%	83.29%	84.98%	86.42%	87.60%
	Conf > 0.7	77.55%	77.55%	78.65%	80.92%	82.66%	84.43%	85.81%	87.19%	88.15%

Confidence Threshold Filtering Consistently Improves Performance: The comprehensive analysis across all minimum match lengths demonstrates that filtering prefix matches based on confidence thresholds consistently improves acceptance rates. Key findings include:

- **Universal Improvement**: Confidence filtering improves performance at every minimum match length, from ML 0 through ML 16, indicating that high-confidence tokens are more reliably predicted across all context scenarios
- **Domain-Specific Advantages**: Python coding shows even stronger improvements, with Target-5M (Python) advancing from 62.54% to 77.55% at ML 0 (15 percentage points) and maintaining consistent gains across longer match lengths
- **Threshold Sensitivity**: Higher confidence thresholds (>0.7) provide the largest improvements, suggesting that very high-confidence predictions are indeed more reliable for speculation
- **Computational Efficiency**: Confidence-based filtering adds no computational overhead - it requires only a simple probability threshold comparison during n-gram lookup

4.5.2.2 Inference Speed Measurements: Negligible Speculation Overhead

Our empirical measurements reveal Infinigram's extraordinary speed advantage that fundamentally changes the economics of speculative decoding:

Core Performance Metrics:

- **Index Size**: 33GB for 5M training examples (memory-mapped for efficient access)
- **Speculation Latency**: **1ms total for generating ALL 16 lookahead tokens simultaneously**
- **Target Model Comparison**: Llama-3.1-8B-Instruct requires 80-150ms per token (single forward pass)

Revolutionary Speed Implications:

The 1ms speculation time for ALL 16 tokens reveals Infinigram's transformative advantage over autoregressive neural models:

(1) **Faster than Neural Draft Models**: Neural models like Llama-3.2-1B (640-960ms) and Eagle-3 ($\sim$16ms for 16 tokens) require sequential processing - Infinigram generates all 16 tokens in just 1ms
(2) **Parallel vs Sequential Generation**: Neural models must generate tokens one-by-one; Infinigram retrieves entire sequences instantly from its index using optimized `inf_nkt` function
(3) **CPU-Only Achievement**: This performance uses only CPU resources, leaving GPU fully available for Llama-3.1-8B target model
(4) **Scales with Speculation Depth**: The `inf_nkt` optimization ensures retrieval time remains constant regardless of speculation depth

This measurement definitively establishes that Infinigram speculative decoding, with the `inf_nkt` optimization, achieves sub-millisecond speculation times that are orders of magnitude faster than neural draft models while using only CPU resources, making it an ideal complement to large target models.

4.5.2.3 Average Acceptance Length Analysis

For evaluating acceptance lengths, we employ greedy decoding where tokens are selected deterministically by choosing the highest probability token at each step. This provides a

consistent and reproducible evaluation of the expected acceptance lengths under deterministic conditions.

TABLE 4.4: Average Accepted Tokens Per Speculation Sequence - All Prompts

Configuration	ML 0	ML 2	ML 4	ML 6	ML 8	ML 10	ML 12	ML 14	ML 16
SFT-5M	0.75	0.76	0.87	1.06	1.31	1.53	1.65	1.62	1.56
SFT-5M Target	0.97	0.97	1.13	1.36	1.66	1.84	2.16	2.17	2.12

TABLE 4.5: Average Accepted Tokens Per Speculation Sequence - Python Coding

Config	ML 0	ML 2	ML 4	ML 6	ML 8	ML 10	ML 12	ML 14	ML 16
SFT-5M	1.6	1.61	1.75	2	2.28	2.61	2.74	3.22	3.22
SFT-5M Target	2.85	2.86	3.03	3.35	3.58	3.81	4.16	4.51	4.72
SFT-5M Tgt (C>0.5)	3.24	3.24	3.47	3.62	3.82	4.00	4.35	4.67	4.86

Sequence Length Benefits: Target model-based training demonstrates significant improvements across all match lengths. For Python coding tasks, SFT-5M Target achieves 2.85-4.72 tokens compared to SFT-5M's 1.60 tokens (only available at match length 0). General prompts show similar patterns with SFT-5M Target ranging from 0.97-2.12 tokens versus SFT-5M's 0.75 tokens. The confidence-filtered approach further improves Python coding performance to 3.24-4.86 tokens.

Training Data Availability Impact: The stark difference in available match lengths highlights how SFT-only training produces insufficient n-gram coverage, limiting speculation to short contexts. Target model-based training provides consistent performance improvements across all speculation depths.

4.5.2.4 Match Hit Rate Analysis: Pattern Density Insights

The match hit rate reveals fundamental differences in pattern predictability across domains:

Key Insights from Match Hit Rates:

(1) **Domain Structure Impact:** Python coding maintains 4-5x higher pattern retention at length 16 compared to general conversation, explaining the superior acceptance rates

TABLE 4.6: Match Hit Rate: Pattern Availability Across Domains and Approaches

Configuration	Length 0	Length 4	Length 8	Length 16	Retention Rate
General Prompts					
SFT-5M	100%	78.45%	25.01%	4.51%	4.5%
Target-5M	100%	81.51%	35.23%	7.98%	8.0%
Python Coding					
SFT-5M (Python)	100%	86.67%	52.97%	20.19%	20.2%
Target-5M (Python)	100%	88.34%	60.46%	31.47%	31.5%

(2) **Target Model Advantage:** Target model-based indices show consistently higher match hit rates, indicating better pattern coverage

(3) **Pattern Scarcity in Natural Language:** The rapid falloff in general conversation (100% $\rightarrow$ 4.5%) reveals why longer speculation sequences have diminishing returns

(4) **Code Predictability:** Programming syntax and common patterns maintain high hit rates even at length 16, making code generation an ideal use case for Infinigram speculation

4.5.3 Context Length Sensitivity

The effectiveness of n-gram speculation varies with context length. Based on our Infini-5M results:

4.5.3.1 Performance Characteristics by Match Length Range

The effectiveness of n-gram speculation scales with the minimum guaranteed match length between context and n-gram patterns. Our analysis reveals distinct performance characteristics across different match length ranges:

- **Short minimum match lengths (ML 0-4)**: Acceptance rates range from 45.75% to 50.26% for Target-5M. At ML 0, the system accepts any n-gram match regardless of context overlap, capturing basic token-level patterns. Performance improves modestly as minimum match requirements increase, indicating that even short context matches provide valuable predictive signal.

- **Medium minimum match lengths (ML 6-10)**: This range shows the steepest improvement, with acceptance rates climbing from 55.82% to 66.77%. This demonstrates the sweet spot where longer context requirements begin capturing meaningful conversational and structural patterns while maintaining sufficient n-gram coverage.
- **Long minimum match lengths (ML 12-16)**: Performance reaches peak levels between 71.92% and 76.84%, but with diminishing returns. These high match length requirements ensure very strong context alignment, though the pool of matching n-grams becomes increasingly sparse, limiting the frequency of successful speculation.

Key Insight: Match length represents the minimum required overlap between the current context and stored n-gram patterns. Higher match lengths demand stronger contextual alignment, leading to more reliable predictions but reduced coverage due to n-gram sparsity.

4.5.4 Domain Transfer Analysis

Our comprehensive evaluation reveals fundamental insights about domain-specific performance and transferability:

4.5.4.1 Domain-Specific Performance Advantages

The stark performance differences between general conversation and Python coding domains demonstrate Infinigram's adaptability:

- **General Conversation**: Target-model Infinigram achieves 45.75% average acceptance across at minimum match length 0
- **Python Coding**: Performance jumps to 62.54% average acceptance - a 16.79% relative improvement
- **Confidence-Filtered Python**: With confidence > 0.7, acceptance reaches 77.55% at minimum match length 0

4.5.4.2 Key Domain Transfer Insights

(1) **Training Distribution Criticality**: SFT-based models show severe performance degradation when tested on target model outputs, revealing that the training distribution fundamentally determines speculation effectiveness. This distribution mismatch cannot be overcome through larger indices alone.

(2) **Structured vs Unstructured Domains**: Python coding's superior performance (38% better than general conversation) demonstrates that structured domains with deterministic syntax patterns are ideal for n-gram speculation. The high match hit rates (31.5% at length 16 vs 8.0% for general) confirm greater pattern predictability.

(3) **Instant Domain Adaptation**: Unlike neural models requiring extensive retraining, Infinigram achieves domain specialization only with index building from 5M examples. This enables rapid deployment for new domains without GPU resources.

(4) **Zero-Shot Transfer Limitations**: Our results show that indices trained on one distribution (SFT) cannot effectively transfer to another (target model outputs). This suggests domain-specific indices are essential for optimal performance rather than attempting cross-domain generalization.

4.5.4.3 Practical Implications for Multi-Domain Deployment

Based on our findings, we recommend:

- Build separate indices for each target domain rather than attempting universal coverage
- Prioritize structured domains (code, math, technical writing) for maximum performance gains
- Use target model outputs for training data collection to ensure distribution alignment
- Implement confidence thresholds to improve performance even without context matching

4.5.5 Comparison with State-of-the-Art Neural Speculative Decoding

To position Infinigram within the broader speculative decoding landscape, we conduct comprehensive comparisons with leading neural approaches on coding benchmarks. This evaluation demonstrates Infinigram's competitive performance while highlighting its unique advantages in deployment efficiency.

4.5.5.1 Comparative Performance Results

Table 4.7 presents comprehensive performance metrics across two coding benchmarks:

TABLE 4.7: Comparison with State-of-the-Art Neural Speculative Decoding (Batch Size = 1)

Method	Acceptance Length HumanEval	Acceptance Length SFT Coding	Speed HumanEval (tok/s)	Speed SFT Coding (tok/s)	Speedup HumanEval	Speedup SFT Coding	Speculation Latency
Llama-3.1-8B (Baseline)	1.0	1.0	27.33	27.33	1.0×	1.0×	N/A
Eagle-3 (Neural)	3.5	3.03	68.63	61.4	2.51×	2.24×	∼1ms/token
Infinigram Target-5M	**3.11**	**2.85**	**81.13**	**66.08**	**2.96×**	**2.42×**	**∼1ms/16 tokens**

4.5.5.2 Key Performance Insights

Competitive Acceptance Performance: Infinigram achieves comparable acceptance lengths to Eagle-3[86] (3.11 vs 3.5 on HumanEval, 2.85 vs 3.03 on SFT coding), demonstrating that n-gram approaches can match neural model prediction quality in structured domains.

Superior Inference Speed: Infinigram delivers higher inference throughput (81.13 tok/s vs 68.63 tok/s on HumanEval), achieving 2.96× speedup compared to Eagle-3's 2.51× speedup. This advantage stems from the dramatically faster speculation process.

Speculation Latency Advantage: The most striking difference lies in speculation overhead: Infinigram generates 16 tokens in 1ms total, while Eagle-3 requires ∼1ms per token. This 16× speculation speed advantage enables higher overall throughput despite slightly lower acceptance lengths.

Resource Efficiency: Infinigram achieves these results using only CPU resources, while Eagle-3 requires GPU computation for draft model execution. This fundamental difference

makes Infinigram deployable in resource-constrained environments where neural approaches are impractical.

4.5.5.3 Domain-Specific Advantages

The results reveal Infinigram's particular strength in coding domains:

- **HumanEval Performance**: 2.96× speedup demonstrates effectiveness on algorithmically diverse programming challenges
- **Conversational Coding**: 2.42× speedup on SFT dataset shows robustness across different coding interaction styles
- **Pattern Matching Excellence**: High performance stems from coding's repetitive structures and common patterns that n-gram models capture effectively

4.5.6 Cross-Model and Scale Generalization

Following the methodology of Sec. 4.4.1.6, each Qwen3 target uses its own freshly-built 5M-example target-aligned index. Table 4.8 reports speculation throughput, end-to-end speedup, and average accepted length for Qwen3-8B and Qwen3-32B across the three benchmarks.

TABLE 4.8: Inference speed of Infinigram-based speculative decoding across Qwen3 target models and benchmarks. Spec TPS reports tokens/second under speculation; Speedup is relative to non-speculative decoding of the same target.

Target	Benchmark	Mode	Spec TPS	Speedup	Accept. Len.
Qwen3-8B	math500	INFGRAM	61.9	2.02×	2.17
Qwen3-8B	HumanEval	INFGRAM	68.7	2.30×	2.49
Qwen3-8B	Alpaca	INFGRAM	59.1	1.96×	2.08
Qwen3-32B	math500	INFGRAM	35.8	2.04×	2.04
Qwen3-32B	HumanEval	INFGRAM	33.7	1.99×	1.98
Qwen3-32B	Alpaca	INFGRAM	31.2	1.76×	1.76

Both model shows good acceptance length in the 1.76–2.49 range. End-to-end speedups remain in the $1.76\times$–$2.30\times$ range across both targets and across all three task styles, and the speedup factor does not erode as the target scales from 8B to 32B parameters. This is

consistent with the design hypothesis in Sec. 4.4.1.6: Infinigram-based speculative decoding remains effective among different model sizes and architectures.

4.6 Analysis and Discussion

4.6.1 Key Findings: Infinigram's Transformative Advantages

Our experimental evaluation reveals several revolutionary insights that position Infinigram as a game-changing approach to speculative decoding:

(1) **Instant Prediction with Zero Training**: Infinigram achieves microsecond-level token prediction without any neural network training, gradient computation, or hyperparameter tuning - requiring only index building from 5M examples processed in $\sim$840 seconds

(2) **CPU-First Architecture**: Unlike GPU-dependent neural approaches, Infinigram fully utilizes CPU resources, making it deployable in standard computing environments without expensive hardware requirements

(3) **Perfect Recall on Familiar Patterns**: When input patterns match training data, Infinigram provides instantaneous prediction with near-perfect accuracy, especially evident in coding tasks (66.24% average acceptance) where repetitive structures dominate

(4) **Data Efficiency Breakthrough**: Strong performance with only 5M examples demonstrates orders of magnitude better data efficiency compared to neural draft models requiring billions of parameters and extensive training

(5) **Real-Time Adaptability**: Confidence-based filtering and dynamic depth selection operate with zero computational overhead, enabling real-time optimization without performance penalties

4.6.2 Infinigram's Deployment Advantages

4.6.2.1 Computational Efficiency Transformation

Infinigram fundamentally transforms the economics and practicality of speculative decoding deployment:

- **Microsecond Latency**: Token prediction occurs in microseconds using simple array lookups, compared to milliseconds for neural forward passes
- **Zero Training Infrastructure**: No GPU clusters, distributed training, or specialized ML infrastructure required - standard CPU servers suffice
- **Instant Deployment**: Index construction completes in minutes, enabling rapid prototyping and production deployment
- **Minimal Resource Requirements**: Operates efficiently on commodity hardware with standard RAM, eliminating GPU memory constraints
- **Linear Scalability**: Performance scales linearly with available CPU cores, not constrained by GPU parallelization limits

4.6.2.2 Operational Excellence

- **Maintenance-Free Operation**: No model retraining, checkpoint management, or hyperparameter monitoring required
- **Deterministic Performance**: Consistent prediction latency and accuracy, unlike neural models with variable computation costs
- **Pattern Specialization**: Excels in domains with familiar patterns (coding, structured text) where exact matches provide perfect predictions
- **Cost Efficiency**: Orders of magnitude lower computational cost compared to training and running neural draft models

4.6.3 Practical Deployment Considerations

4.6.3.1 Distribution Alignment Strategy

The overfitting findings fundamentally change deployment recommendations:

- **Target Model Alignment Critical**: Always build indices from target model outputs, not training data responses
- **One-Time Cost Justified**: The initial inference cost to generate target model responses is offset by 9.7% performance improvement
- **Avoid SFT-Based Shortcuts**: SFT-based indices appear effective when tested on SFT data but fail to generalize to actual model behavior
- **Domain-Specific Indices**: Specialized indices for structured domains (code, math) can achieve >70% acceptance rates

4.6.3.2 Confidence-Aware Deployment

- **Adaptive Speculation**: Implement probability threshold filtering (>0.5) for 3-6% acceptance improvement
- **Sequence Length Optimization**: High-confidence scenarios enable longer speculation sequences (4.86 vs 2.85 average tokens)
- **Dynamic Depth Selection**: Adjust speculation depth based on domain and model confidence

4.6.4 Overfitting Analysis: A Critical Discovery

4.6.4.1 The Distribution Mismatch Problem

One of the most significant findings of this research is the discovery of systematic overfitting in SFT-based Infinigram indices. This finding has profound implications for the field of speculative decoding:

Overfitting Manifestation: When SFT-based indices (trained on teacher model responses) are evaluated on target model outputs instead of SFT responses, we observe:

- 5-10% drop in acceptance rates across all match lengths
- Reduced average accepted sequence length (1.6 vs 2.85 tokens)
- Poor generalization to the actual inference distribution

Root Cause Analysis: The distribution mismatch occurs because:

(1) **Teacher-Student Divergence:** SFT data is typically generated by teacher models (e.g., GPT-4) for training student models. The student model develops its own generation patterns during training that diverge from the teacher's outputs [109]
(2) **Model-Specific Generation Patterns:** Each model develops unique statistical patterns, vocabulary preferences, and structural biases during training. An n-gram index built from GPT-4 outputs will not match the patterns of a Llama-3 or Mistral model deploying the speculator [21]
(3) **Training-Inference Gap:** Even when fine-tuned on the same SFT data, the target model's actual generation behavior differs from the training distribution due to sampling strategies, temperature settings, and autoregressive accumulation of subtle biases [110]

4.6.4.2 Implications for Speculative Decoding Research

This discovery challenges fundamental assumptions in the field:

- **Evaluation Methodology:** Previous studies may have overestimated performance by testing on matched distributions [111]
- **Training Data Selection:** The source of training data for draft mechanisms is more critical than previously understood [112]
- **Generalization Requirements:** Draft models must be designed to match target model distributions, not training data distributions [113]

4.6.5 Comparison with Alternative Approaches

4.6.5.1 vs. Neural Draft Models

Advantages of Infinigram:

- **No neural training**: Index construction requires only ∼840 seconds of CPU time
- **Batch generation**: Generates 16 tokens in 1ms total, compared to 16ms for sequential neural generation
- **Competitive performance**: 2.96× speedup on HumanEval (vs. Eagle-3's 2.51×) using only CPU resources
- **Domain adaptation**: New domains require only index rebuilding, not model retraining
- **Model-agnostic speed**: Speculation latency remains constant regardless of target model size

Disadvantages:

- Limited to patterns seen in training data
- Reduced effectiveness for creative/novel text
- Larger memory footprint for comprehensive coverage

4.6.5.2 vs. Tree-based Speculation

Infinigram offers linear speculation paths while tree-based methods explore multiple branches. Our approach is memory efficient but potentially less helpful for high-entropy continuations.

4.6.6 Limitations and Future Work

4.6.6.1 Current Limitations

(1) **Novel Pattern Generation**: N-gram models struggle with truly novel combinations
(2) **Index Update Cost**: Incorporating new data requires index reconstruction

(3) **Memory Scaling**: Index size grows linearly with training data

4.6.6.2 Future Research Directions

- **Hybrid Approaches**: Combine n-gram and neural speculation for optimal coverage
- **Dynamic Index Updates**: Online learning for adapting to new patterns
- **Compression Techniques**: Advanced encoding to reduce index size
- **Adaptive Depth Selection**: Context-aware speculation depth adjustment

CHAPTER 5

Conclusion

This thesis has presented two novel approaches to optimizing large language model efficiency: CARE for parameter-efficient fine-tuning and Infinigram-based speculative decoding for accelerated inference. Both methods address critical bottlenecks in practical LLM deployment while maintaining model quality through principled statistical approaches.

5.1 Summary of Contributions

5.1.1 CARE: Revolutionizing GQA-to-MLA Conversion for KV-Cache Compression

The CARE (Covariance-Aware and Rank-Enhanced) method represents a significant advancement in GQA-to-MLA conversion for KV-cache compression by introducing dynamic, layer-specific rank allocation based on statistical analysis of weight matrices. Unlike existing SVD approaches that apply uniform rank decomposition across all layers, CARE recognizes the heterogeneous contribution patterns of different network components and adapts accordingly.

Key Technical Achievements:

- **Activation-Preserving Factorization**: Covariance-weighted SVD on $\sqrt{C}\,W$ minimizes activation-space error rather than weight-space Frobenius error, substantially reducing attention drift relative to naive SVD initialization

- **Water-Filling Rank Allocation**: Adjusted rank distribution across layers and between K/V projections under fixed KV-parity, prioritizing spectrally complex layers and reclaiming budget from intrinsically low-rank ones
- **Full MLA Restoration via Partial RoPE**: A small decoupled-RoPE channel paired with the latent K/V factorization enables 100% MLA conversion while preserving the original KV-cache footprint
- **Matched-Budget Healing**: Brief distillation-based fine-tuning under matched 1B/3B-token budgets recovers and exceeds the original GQA accuracy, requiring fewer steps than baseline initializations

5.1.2 Infinigram: Transforming LLM Inference Speed

The Infinigram-based speculative decoding system[92] demonstrates that n-gram modeling can achieve revolutionary speedups in modern neural language generation. By replacing computationally expensive neural draft models with ultra-fast statistical lookups, this approach fundamentally insights the future of LLM deployment.

Breakthrough Performance Metrics:

- **Lightning-Fast Speculation**: 1ms to generate ALL 16 lookahead tokens simultaneously, compared to 1ms per token for small neural drafts[88]
- **High Acceptance Rates**: With target model-aligned indices, acceptance climbs from 45.75% (ML 0) to 76.84% (ML 16) on general prompts, and from 62.54% to 85.38% on Python coding; confidence-filtered speculation ($>$0.7) further raises Python acceptance to 88.15%
- **Minimal Training Requirements**: Index construction completed in $\sim$840 seconds using only statistical counting, eliminating complex neural training procedures
- **Distribution Preservation**: Guaranteed exact output distribution matching through rigorous acceptance mechanisms

Critical Research Discovery: Our investigation revealed systematic overfitting in SFT-based speculation indices, where models trained on human collected responses showed 5-10%

performance degradation when applied to target model outputs. This finding challenges fundamental assumptions in speculative decoding research and establishes that draft mechanisms must align with target model distributions rather than training data distributions.

5.2 Limitations and Areas for Improvement

5.2.1 CARE Method Limitations

While CARE demonstrates significant improvements over uniform rank allocation, several limitations warrant future investigation:

- **Initial SVD Overhead**: The one-time computational cost of SVD analysis may be prohibitive for very large models or frequent re-ranking scenarios
- **Static Rank Allocation**: Current implementation determines ranks during initialization; dynamic adaptation during training could yield further improvements
- **Task Generalization**: Performance evaluation focused on specific fine-tuning scenarios; broader task coverage would strengthen generalization claims

5.2.2 Infinigram System Constraints

The Infinigram approach, while revolutionary in speed, faces inherent limitations of n-gram modeling:

- **Pattern Sparsity**: Rapid performance degradation for unseen patterns limits effectiveness in novel domains
- **Context Length Limits**: Diminishing returns beyond 16-gram contexts restrict applicability to very long-range dependencies
- **Domain Dependence**: Superior performance on structured domains may not translate to all application areas

5.3 Future Research Directions

Three concrete research directions emerge:

- **Unified framework**: Combining CARE's training efficiency with Infinigram's inference acceleration and other optimization techniques in a single optimization pipeline
- **Dynamic adaptation**: Online rank adjustment during inference based on task complexity
- **Hybrid speculation**: Combining n-gram statistics with lightweight neural models to handle both familiar and novel patterns

5.4 Final Reflection

This thesis validates that principled statistical analysis and classical optimization techniques remain highly effective for modern LLM challenges. The success of CARE's activation-aware rank allocation and Infinigram's statistical speculation demonstrates that understanding fundamental mathematical algorithms and LLM systems can yield practical efficiency gains comparable to or exceeding complex neural innovations. Most critically, our discovery of distribution mismatch in speculative decoding underscores the importance of empirical validation over theoretical assumptions in optimization research.

Bibliography

[1] F. Bie, Y. Yang, Z. Zhou, A. Ghanem, M. Zhang, Z. Yao, X. Wu, C. Holmes, P. Golnari, D. A. Clifton, Y. He, D. Tao and S. L. Song. 'RenAIssance: A Survey Into AI Text-to-Image Generation in the Era of Large Model'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 47.3 (2025), pp. 2212–2231.

[2] Z. Zhou, F. Bie, Z. Chen, Z. Zhang, Y. Yang, J. Wang, B. Athiwaratkun, X. Wu and S. L. Song. 'CARE: Covariance-Aware and Rank-Enhanced Decomposition for Enabling Multi-Head Latent Attention'. In: *International Conference on Learning Representations*. 2026.

[3] C. Leiter, R. Zhang, Y. Chen, J. Belouadi, D. Larionov, V. Fresen and S. Eger. 'Chatgpt: A meta-analysis after 2.5 months'. In: *arXiv preprint arXiv:2302.13795* (2023).

[4] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[5] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Anderson, A. Anitescu et al. 'GPT-4 Technical Report'. In: *arXiv preprint arXiv:2303.08774* (2023).

[6] Anthropic. 'Claude: A Next-Generation AI Assistant Built by Anthropic'. In: *Anthropic AI Safety* (2023).

[7] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar et al. 'Llama: Open and efficient foundation language models'. In: *arXiv preprint arXiv:2302.13971* (2023).

[8] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale et al. 'Llama 2: Open foundation and fine-tuned chat models'. In: *arXiv preprint arXiv:2307.09288* (2023).

[9] D. Adiwardana, M.-T. Luong, D. R. So, J. Hall, N. Fiedel, R. Thoppilan, Z. Yang, A. Kulshreshtha, G. Nemade, Y. Lu et al. 'Towards a Human-like Open-Domain Chatbot'. In: *arXiv preprint arXiv:2001.09977* (2020).

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman et al. 'Evaluating Large Language Models Trained on Code'. In: *arXiv preprint arXiv:2107.03374* (2021).

[11] J. W. Rae, S. Borgeaud, T. Cai, K. Millican, J. Hoffmann, F. Song, J. Aslanides, S. Henderson, R. Ring, S. Young et al. 'Scaling Language Models: Methods, Analysis & Insights from Training Gopher'. In: *arXiv preprint arXiv:2112.11446* (2021).

[12] X. Wang, Q. Liu, Y. Liang and Z. Wang. 'ChatGPT in Education: Strengths, Limitations, and Future Directions'. In: *Computers and Education: Artificial Intelligence* 5 (2023), p. 100181.

[13] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell et al. 'Language Models are Few-Shot Learners'. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1877–1901.

[14] J. Wei, X. Wang, D. Schuurmans, M. Bosma, E. Chi, Q. Le and D. Zhou. 'Chain-of-Thought Prompting Elicits Reasoning in Large Language Models'. In: *arXiv preprint arXiv:2201.11903* (2022).

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser and I. Polosukhin. 'Attention is all you need'. In: *NeurIPS* 30 (2017).

[16] J. D. M.-W. C. Kenton and L. K. Toutanova. 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: *Proceedings of NAACL-HLT*. 2019, pp. 4171–4186.

[17] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li and P. J. Liu. 'Exploring the limits of transfer learning with a unified text-to-text transformer'. In: *JMLR* 21.1 (2020), pp. 5485–5551.

[18] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell et al. 'Language models are few-shot learners'. In: *NeurIPS* 33 (2020), pp. 1877–1901.

[19] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu and D. Amodei. 'Scaling Laws for Neural Language Models'. In: *arXiv preprint arXiv:2001.08361* (2020).

[20] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler et al. 'Emergent Abilities of Large Language Models'. In: *arXiv preprint arXiv:2206.07682* (2022).

[21] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray et al. 'Training Language Models to Follow Instructions with Human Feedback'. In: *NeurIPS* (2022).

[22] R. Taylor, M. Kardas, G. Cucurull, T. Scialom, A. Hartshorn, E. Saravia, A. Poulton, V. Kerkez and R. Stojnic. 'Galactica: A Large Language Model for Science'. In: *arXiv preprint arXiv:2211.09085* (2022).

[23] S. Rajbhandari, J. Rasley, O. Ruwase and Y. He. 'ZeRO: Memory optimizations toward training trillion parameter models'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), pp. 1–16.

[24] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal and J. Dean. 'Efficiently Scaling Transformer Inference'. In: *arXiv preprint arXiv:2211.05102* (2022).

[25] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly et al. 'An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale'. In: *arXiv preprint arXiv:2010.11929* (2020).

[26] S. Hochreiter and J. Schmidhuber. 'Long Short-Term Memory'. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[27] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel. 'Backpropagation Applied to Handwritten Zip Code Recognition'. In: *Neural Computation* 1.4 (1989), pp. 541–551.

[28] J. D. M.-W. C. Kenton and K. Toutanova. 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: *Proceedings of NAACL-HLT*. 2019.

[29] D. Bahdanau, K. Cho and Y. Bengio. 'Neural Machine Translation by Jointly Learning to Align and Translate'. In: *arXiv preprint arXiv:1409.0473* (2014).

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin. 'Attention Is All You Need'. In: *NeurIPS*. 2017.

[31] A. Radford, K. Narasimhan, T. Salimans and I. Sutskever. 'Improving Language Understanding by Generative Pre-Training'. In: *OpenAI Blog* (2018).

[32] J. Gu, J. Bradbury, C. Xiong, V. O. Li and R. Socher. 'Non-Autoregressive Neural Machine Translation'. In: *arXiv preprint arXiv:1711.02281* (2017).

[33] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova. 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding'. In: *arXiv preprint arXiv:1810.04805* (2018).

[34] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer and V. Stoyanov. 'RoBERTa: A Robustly Optimized BERT Pretraining Approach'. In: *arXiv preprint arXiv:1907.11692* (2019).

[35] K. Clark, M.-T. Luong, Q. V. Le and C. D. Manning. 'ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators'. In: *arXiv preprint arXiv:2003.10555* (2020).

[36] J. Lee, E. Mansimov and K. Cho. 'Deterministic Non-Autoregressive Neural Sequence Modeling by Iterative Refinement'. In: *arXiv preprint arXiv:1802.06901* (2018).

[37] C. Shao, Y. Feng, J. Zhang, F. Meng, X. Chen and J. Zhou. 'Minimizing the Bag-of-Ngrams Difference for Non-Autoregressive Neural Machine Translation'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.05 (2020), pp. 198–205.

[38] X. Ma, C. Zhou, X. Li, G. Neubig and E. Hovy. 'FlowSeq: Non-Autoregressive Conditional Sequence Generation with Generative Flow'. In: *arXiv preprint arXiv:1909.02480* (2019).

[39] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly et al. 'An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale'. In: *ICLR*. 2020.

[40] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark et al. 'Learning transferable visual models from natural language supervision'. In: *ICML*. PMLR. 2021, pp. 8748–8763.

[41] L. Yuan, D. Chen, Y.-L. Chen, N. Codella, X. Dai, J. Gao, H. Hu, X. Huang, B. Li, C. Li et al. 'Florence: A new foundation model for computer vision'. In: *arXiv preprint arXiv:2111.11432* (2021).

[42] Z. Dai, H. Liu, Q. V. Le and M. Tan. 'Coatnet: Marrying convolution and attention for all data sizes'. In: *Advances in neural information processing systems* 34 (2021), pp. 3965–3977.

[43] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper and B. Catanzaro. 'Megatron-lm: Training multi-billion parameter language models using model parallelism'. In: *arXiv preprint arXiv:1909.08053* (2019).

[44] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, E. Li, X. Wang, M. Dehghani, S. Brahma et al. 'Scaling instruction-finetuned language models'. In: *arXiv preprint arXiv:2210.11416* (2022).

[45] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin et al. 'Opt: Open pre-trained transformer language models'. In: *arXiv preprint arXiv:2205.01068* (2022).

[46] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom and G. Synnaeve. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL].

[47] A. Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI].

[48] Meta AI. *Llama 3 Model Card*. https://github.com/meta-llama/llama3. Accessed: 2024. 2024.

[49] DeepSeek-AI, : X. Bi, D. Chen, G. Chen, S. Chen, D. Dai, C. Deng, H. Ding, K. Dong, Q. Du, Z. Fu, H. Gao, K. Gao, W. Gao, R. Ge, K. Guan, D. Guo, J. Guo, G. Hao, Z. Hao, Y. He, W. Hu, P. Huang, E. Li, G. Li, J. Li, Y. Li, Y. K. Li, W. Liang, F.

Lin, A. X. Liu, B. Liu, W. Liu, X. Liu, X. Liu, Y. Liu, H. Lu, S. Lu, F. Luo, S. Ma, X. Nie, T. Pei, Y. Piao, J. Qiu, H. Qu, T. Ren, Z. Ren, C. Ruan, Z. Sha, Z. Shao, J. Song, X. Su, J. Sun, Y. Sun, M. Tang, B. Wang, P. Wang, S. Wang, Y. Wang, Y. Wang, T. Wu, Y. Wu, X. Xie, Z. Xie, Z. Xie, Y. Xiong, H. Xu, R. X. Xu, Y. Xu, D. Yang, Y. You, S. Yu, X. Yu, B. Zhang, H. Zhang, L. Zhang, L. Zhang, M. Zhang, M. Zhang, W. Zhang, Y. Zhang, C. Zhao, Y. Zhao, S. Zhou, S. Zhou, Q. Zhu and Y. Zou. *DeepSeek LLM: Scaling Open-Source Language Models with Longtermism*. 2024. arXiv: 2401.02954 [cs.CL].

[50] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong and W. Liang. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE].

[51] D. Dai, C. Deng, C. Zhao, R. X. Xu, H. Gao, D. Chen, J. Li, W. Zeng, X. Yu, Y. Wu, Z. Xie, Y. K. Li, P. Huang, F. Luo, C. Ruan, Z. Sui and W. Liang. *DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models*. 2024. arXiv: 2401.06066 [cs.CL].

[52] DeepSeek AI. 'DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model'. In: *arXiv preprint arXiv:2405.04434* (2024).

[53] DeepSeek-AI et al. *DeepSeek-V3 Technical Report*. 2025. arXiv: 2412.19437 [cs.CL].

[54] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti et al. 'Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model'. In: *arXiv preprint arXiv:2201.11990* (2022).

[55] B. Zhang and R. Sennrich. 'Root Mean Square Layer Normalization'. In: *Advances in Neural Information Processing Systems* 32 (2019).

[56] N. Shazeer. 'GLU Variants Improve Transformer'. In: *arXiv preprint arXiv:2002.05202* (2020).

[57] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen and Y. Liu. 'RoFormer: Enhanced Transformer with Rotary Position Embedding'. In: *Neurocomputing* 568 (2024), p. 127063.

[58] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebron and S. Sanghai. 'GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints'. In: *arXiv preprint arXiv:2305.13245* (2023).

[59] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL].

[60] OpenAI. 'Learning to Reason with LLMs'. In: *OpenAI Blog* (2024).

[61] D. Dai, C. Deng, C. Zhao, R. Xu, H. Gao, D. Chen, J. Li, W. Zeng, X. Yu, Y. Wu et al. 'DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models'. In: *arXiv preprint arXiv:2401.06066* (2024).

[62] Y. Leviathan, M. Kalman and Y. Matias. 'Fast Inference from Transformers via Speculative Decoding'. In: *arXiv preprint arXiv:2211.17192* (2023).

[63] F. Gloeckle, B. Y. Idrissi, B. Roziere, D. Lopez-Paz and G. Synnaeve. 'Better and Faster Large Language Models via Multi-token Prediction'. In: *arXiv preprint arXiv:2404.19737* (2024).

[64] A. Aghajanyan, S. Gupta and L. Zettlemoyer. 'Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning'. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*. 2021, pp. 7319–7328.

[65] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun and R. Fergus. 'Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation'. In: *Advances in Neural Information Processing Systems*. Vol. 27. 2014.

[66] J. Xue, J. Li and Y. Gong. 'Restructuring of Deep Neural Network Acoustic Models with Singular Value Decomposition'. In: *Interspeech*. 2013, pp. 2365–2369.

[67] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang and W. Chen. 'LoRA: Low-Rank Adaptation of Large Language Models'. In: *International Conference on Learning Representations*. 2022.

[68] J. Zhang, S. Chen, J. Liu and J. He. 'Composing Parameter-Efficient Modules with Arithmetic Operations'. In: *Advances in Neural Information Processing Systems* 36 (2023).

[69] T. Dettmers, A. Pagnoni, A. Holtzman and L. Zettlemoyer. 'QLoRA: Efficient Finetuning of Quantized LLMs'. In: *Advances in Neural Information Processing Systems* 36 (2024).

[70] Q. Zhang, M. Chen, A. Bukharin, P. He, Y. Cheng, W. Chen and T. Zhao. 'AdaLoRA: Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning'. In: *International Conference on Learning Representations*. 2023.

[71] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal and J. Dean. 'Efficiently Scaling Transformer Inference'. In: *Proceedings of Machine Learning and Systems* 5 (2023).

[72] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang and I. Stoica. 'Efficient Memory Management for Large Language Model Serving with PagedAttention'. In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023, pp. 611–626.

[73] N. Shazeer. 'Fast Transformer Decoding: One Write-Head is All You Need'. In: *arXiv preprint arXiv:1911.02150* (2019).

[74] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett et al. 'H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models'. In: *Advances in Neural Information Processing Systems* 36 (2024).

[75] G. Xiao, Y. Tian, B. Chen, S. Han and M. Lewis. 'Efficient Streaming Language Models with Attention Sinks'. In: *International Conference on Learning Representations* (2024).

[76] Z. Liu, J. Yuan, H. Jin, S. Zhong, Z. Xu, V. Braverman, B. Chen and X. Hu. 'KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache'. In: *International Conference on Machine Learning* (2024).

[77] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica and C. Zhang. 'FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU'. In: *International Conference on Machine Learning*. 2023, pp. 31094–31116.

[78] R. Child, S. Gray, A. Radford and I. Sutskever. 'Generating Long Sequences with Sparse Transformers'. In: *arXiv preprint arXiv:1904.10509* (2019).

[79] N. Kitaev, Ł. Kaiser and A. Levskaya. 'Reformer: The Efficient Transformer'. In: *International Conference on Learning Representations*. 2020.

[80] T. Dao. 'FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning'. In: *International Conference on Learning Representations*. 2024.

[81] Z. Chen, X. Yang, Y. Lin, P. Abbeel, P. Chen, G. Sun and A. A. Ross. 'Cascade Speculative Drafting for Faster LLM Inference'. In: *arXiv preprint arXiv:2312.11462* (2023).

[82] C. Chen, S. Borgeaud, G. Irving, J.-B. Lespiau, L. Sifre and J. Jumper. 'Accelerating Large Language Model Decoding with Speculative Sampling'. In: *arXiv preprint arXiv:2302.01318* (2023).

[83] Y. Leviathan, M. Kalman and Y. Matias. *Fast Inference from Transformers via Speculative Decoding*. 2023. arXiv: 2211.17192 [cs.LG].

[84] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang, Z. Zhang, R. Y. Y. Wong, A. Zhu, L. Yang, X. Shi, C. Shi, Z. Chen, D. Arfeen, R. Abhyankar and Z. Jia. 'SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification'. In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ACM, Apr. 2024, pp. 932–949.

[85] T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen and T. Dao. *Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads*. 2024. arXiv: 2401.10774 [cs.LG].

[86] Y. Li, F. Wei, C. Zhang and H. Zhang. *EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty*. 2025. arXiv: 2401.15077 [cs.LG].

[87] Y. Li, F. Wei, C. Zhang and H. Zhang. 'EAGLE-2: Faster Inference of Language Models with Dynamic Draft Trees'. In: (2024). arXiv: 2406.16858 [cs.CL].

[88] Y. Li, F. Wei, C. Zhang and H. Zhang. 'Eagle-3: Scaling up inference acceleration of large language models via training-time test'. In: *arXiv preprint arXiv:2503.01840* (2025).

[89] P. F. Brown, V. J. D. Pietra, R. L. Mercer, S. A. D. Pietra and J. C. Lai. 'Class-Based n-gram Models of Natural Language'. In: *Computational Linguistics*. Vol. 18. 4. 1992, pp. 467–479.

[90] S. M. Katz. 'Estimation of Probabilities from Sparse Data for the Language Model Component of a Speech Recognizer'. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 35.3 (1987), pp. 400–401.

[91] S. F. Chen and J. Goodman. 'An Empirical Study of Smoothing Techniques for Language Modeling'. In: *Computer Speech & Language* 13.4 (1999), pp. 359–394.

[92] J. Liu, S. Welleck, Y. Bisk and Y. Choi. 'Infinigram: Scaling Unbounded n-gram Language Models to Trillions of Tokens'. In: *arXiv preprint arXiv:2401.17377* (2024).

[93] G. Oliaro, Z. Jia, D. Campos and A. Qiao. *SuffixDecoding: Extreme Speculative Decoding for Emerging AI Applications*. 2025. arXiv: 2411.04975 [cs.CL].

[94] DeepSeek-AI. 'DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model'. In: *arXiv preprint arXiv:2405.04434* (2024).

[95] C.-C. Chang, W.-C. Lin, C.-Y. Lin, C.-Y. Chen, Y.-F. Hu, P.-S. Wang, N.-C. Huang, L. Ceze, M. S. Abdelfattah and K.-C. Wu. 'Palu: Compressing kv-cache with low-rank projection'. In: *arXiv preprint arXiv:2407.21118* (2024).

[96] Z. Yuan, Y. Shang, Y. Song, D. Yang, Q. Wu, Y. Yan and G. Sun. 'Asvd: Activation-aware singular value decomposition for compressing large language models'. In: *arXiv preprint arXiv:2312.05821* (2023).

[97] N. Halko, P.-G. Martinsson and J. A. Tropp. 'Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions'. In: *SIAM Review* 53.2 (2011), pp. 217–288.

[98] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. 2nd. John Wiley & Sons, 1999.

[99] O. Ledoit and M. Wolf. 'A Well-Conditioned Estimator for Large-Dimensional Covariance Matrices'. In: *Journal of Multivariate Analysis* 88.2 (2004), pp. 365–411.

[100] X. Zhai, S. Lei and D. Zhou. 'TransMLA: An Efficient Method for Converting Pretrained GQA Models to MLA'. In: *arXiv preprint arXiv:2410.02780* (2024).

[101] G. Hinton, O. Vinyals and J. Dean. 'Distilling the Knowledge in a Neural Network'. In: *arXiv preprint arXiv:1503.02531* (2015).

[102] Axolotl maintainers and contributors. *Axolotl: Open Source LLM Post-Training*. 2023.

[103] I. H. Witten, A. Moffat and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999.

[104] A. Hindle, E. T. Barr, Z. Su, M. Gabel and P. Devanbu. 'On the Naturalness of Software'. In: *ICSE*. 2012.

[105] Y. Li, F. Wei, C. Zhang and H. Zhang. 'Eagle-2: Faster inference of language models with dynamic draft trees'. In: *arXiv preprint arXiv:2406.16858* (2024).

[106] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv et al. 'Qwen3 Technical Report'. In: *arXiv preprint arXiv:2505.09388* (2025).

[107] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang and T. B. Hashimoto. 'Alpaca: A strong, replicable instruction-following model'. In: *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html* 3.6 (2023), p. 7.

[108] I. Nardini, C. Chen and Y. Wang. *From research to production: Accelerate OSS LLM with EAGLE-3 on Vertex*. LMSYS Blog. Dec. 2025.

[109] S. Dathathri, A. Madotto, J. Lan, J. Hung, E. Frank, P. Molino, J. Yosinski and R. Liu. 'Plug and Play Language Models: A Simple Approach to Controlled Text Generation'. In: *ICLR*. 2020.

[110] H. R. Kirk, Y. Jun, H. Iqbal, E. Benussi, F. Volpin, F. A. Dreyer, A. Goyal and P. Rao. 'Understanding and Mitigating the Security Risks of Large Language Models'. In: *arXiv preprint arXiv:2308.14840* (2023).

[111] I. Gulrajani and D. Lopez-Paz. 'In Search of Lost Domain Generalization'. In: *ICLR*. 2021.

[112] S. Fort, S. Ganguli, S. Jastrzebski and A. M. Saxe. 'Drawing Multiple Augmentation Samples Per Image During Training Efficiently Decreases Test Error'. In: *arXiv preprint arXiv:2105.13343*. 2021.

[113] P. W. Koh, S. Sagawa, H. Marklund, S. M. Xie, M. Zhang, A. Balsubramani, W. Hu, M. Yasunaga, R. L. Phillips, I. Gao et al. ‘WILDS: A Benchmark of in-the-Wild Distribution Shifts’. In: *ICML*. 2021.