# Frankestein Transformer: Unified Encoder-Decoder Library, CLI, and Research-Grounded Design Notes

Erick F. Merino M.
This work is not affiliated
erickfmm@gmail.com

February 2026

**Abstract**

Frankestein Transformer presents a unified configuration-driven toolkit for systematic experimentation with modern transformer architectures, spanning seventeen sequence mixer variants and twenty-two optimizer families. The system supports both encoder-style masked language modeling (MLM) and decoder-style autoregressive (AR) next-token prediction through flexible model class and mode configuration, with specialized fine-tuning workflows for both architectures. The research contributions are threefold: (i) a strict schema-based configuration contract that enables reproducible experimentation across diverse attention mechanisms, including standard softmax attention, sigmoid attention, retentive networks, selective state-space models, continuous-depth transformers, adaptive depth routing (Mixture-of-Depths) [57], conditional memory augmentation (Engram) [8], sparse attention patterns, and gated mechanisms; (ii) a comprehensive optimizer routing framework supporting variance-reduction methods (MARS, Adan, AdEMAMix), memory-efficient variants (Adafactor, GaLore, Lion), schedule-free approaches, second-order preconditioners (Shampoo, SOAP, Sophia), and low-rank APOLLO-family optimizers (Apollo, Apollo-Mini, Q-Apollo) [55]; and (iii) end-to-end workflows spanning quantized deployment via ternary weight packing and sentence-embedding training inspired by SBERT, backed by an expanded quality-assurance stack with broad unit test coverage, YAML example validation, and continuous integration execution. The toolkit implements a web-based configuration interface that provides schema-driven form rendering with inline documentation and real-time validation. This technical reference document includes architectural diagrams, execution-flow visualizations, decision tables, and comprehensive appendices synthesizing literature on transformer architectures, sparse attention mechanisms, gated attention variants, and optimization algorithms. The system enables rapid iteration while maintaining reproducible experimental conditions through its schema-first design philosophy.

## Contents

**1 Introduction** **6**
1.1 Motivation and Problem Statement . . . 6
1.2 Contributions . . . 7
1.3 Reading Guide . . . 8
1.4 Web-Based Configuration Builder . . . 8

**2 Related Work** **9**
2.1 Sequence Mixer Architectures . . . 9
2.1.1 Dense Attention Baselines . . . 9
2.1.2 Recurrent and Retentive Architectures . . . 9

2.1.3 Sparse Attention Mechanisms . . . 9
2.1.4 Gated Attention Mechanisms . . . 10
2.2 Optimization Algorithms . . . 10

**3 System Design and Architecture** **10**
3.1 Configuration-Centric Architecture . . . 10
3.2 Schema Scope and Validation Rules . . . 13
3.3 Complete Model Feature Inventory . . . 13
3.4 Training Task Types . . . 14
3.5 Complete Training Feature Inventory . . . 14
3.6 Optimizer Prefix Contract (Full) . . . 15
3.7 Training Safety and Runtime Semantics . . . 16
3.8 Normalization Variants: RMSNorm, Dynamic Tanh, and Dynamic Erf . . . 16
3.9 RMSNorm . . . 16
3.10 Dynamic Tanh (DyT) . . . 17
3.11 Dynamic Erf (Derf) . . . 17
3.12 Schema Implications . . . 17

**4 Architecture Taxonomy and Implementation** **18**
4.1 Attention and Sequence-Mixer Families . . . 18
4.2 Standard Attention . . . 18
4.3 Sigmoid Attention . . . 18
4.4 Retentive Formulation . . . 19
4.5 Selective SSM (Mamba) . . . 19
4.6 ODE-style Continuous Updates . . . 19
4.7 Test-time Memory (Titans) . . . 19
4.8 Sparse and Gated Extensions in the Current Codebase . . . 20
4.9 Implemented Sparse Attention Blocks (Detailed) . . . 20
4.10 Implemented Gated Attention Blocks (Detailed) . . . 21

**5 Optimizer Families and Training Dynamics** **23**

**6 Quantization and Deployment** **23**
6.1 Ternary Quantization . . . 24
6.2 Activation Quantization . . . 24
6.3 Size Estimates . . . 24

**7 SBERT Downstream Tasks** **24**

**8 Summary Tables** **25**
8.1 Attention and Sequence-Mixer Summary . . . 25
8.2 Optimizer Summary . . . 26

**9 Discussion** **27**
9.1 Schema-Driven Design Trade-offs . . . 27
9.2 Architectural Coverage and Gaps . . . 27
9.3 Optimizer Landscape Fragmentation . . . 27
9.4 Deployment and Production Considerations . . . 28
9.5 Integration and Extensibility Challenges . . . 28

**10 Conclusion** **28**
10.1 Limitations and Future Directions . . . 29

**Bibliography** **30**

**A Annex A: Optimizer Families** **33**
A.1 Memory and Complexity Comparison . . . 33
A.2 The Evolution of Optimization in Neural Networks . . . 34
A.3 Standard Baseline and Adaptive Optimizers . . . 34
A.4 Advanced Momentum and Variance Reduction (2024–2025) . . . 36
A.5 Large-Batch, Memory-Efficient, and Parameter-Free Optimizers . . . 40
A.6 Second-Order, Geometric, and Orthogonality Optimizers . . . 45

**B Annex B: Dense, Recurrent, and Memory-Augmented Transformers** **49**
B.1 Dense Attention Baselines: Standard and Sigmoid . . . 49
B.1.1 Standard Softmax Attention . . . 49
B.1.2 Sigmoid Attention . . . 50
B.2 Recurrent and Retentive Architectures . . . 50
B.2.1 Retentive Networks (RetNet) . . . 50
B.2.2 Mamba: Selective State-Space Models . . . 51
B.3 Continuous-Depth Transformers: ODE Integration . . . 51
B.3.1 ODE Transformer . . . 51
B.4 Test-Time Memory: Titans . . . 52
B.5 Architectural Comparison and Synthesis . . . 52

**C Annex C: Comprehensive Sparse Attention Mechanisms** **53**
C.1 Executive Summary . . . 53
C.2 Sparse Transformer: Factorized Strided and Fixed Patterns . . . 53
C.2.1 Mathematical Formulation . . . 53
C.2.2 Algorithmic Pseudocode . . . 54
C.2.3 Key Characteristics . . . 54
C.3 Longformer: Sliding Window, Dilation, and Global Tokens . . . 55
C.3.1 Mathematical Formulation . . . 55
C.3.2 Algorithmic Pseudocode . . . 55
C.3.3 Key Characteristics . . . 55
C.4 BigBird: Random, Local, and Global Sparse Graph . . . 56
C.4.1 Mathematical Formulation . . . 56
C.4.2 Theoretical Guarantee . . . 56
C.4.3 Algorithmic Pseudocode . . . 56
C.4.4 Key Characteristics . . . 57
C.5 SparseK Attention: Differentiable Top-$k$ Selection . . . 57
C.5.1 Mathematical Formulation . . . 57
C.5.2 Algorithmic Pseudocode . . . 58
C.5.3 Key Characteristics . . . 58
C.6 NSA (Native Sparse Attention): Hardware-Aligned Hierarchical Branches . . . 58
C.6.1 Mathematical Formulation . . . 58
C.6.2 Algorithmic Pseudocode . . . 59
C.6.3 Key Characteristics . . . 59
C.7 FASA: Frequency-Aware Sparse Attention . . . 60
C.7.1 Mathematical Formulation . . . 60
C.7.2 Algorithmic Pseudocode . . . 61
C.7.3 Key Characteristics . . . 61
C.8 SpargeAttn: Two-Stage Block-Level Filtering . . . 61
C.8.1 Mathematical Formulation . . . 61
C.8.2 Algorithmic Pseudocode . . . 62

C.8.3 Key Characteristics . . . . . 62
C.9 Comparative Summary . . . . . 63
C.10 Design Space and Selection Criteria . . . . . 63

**D Annex D: Gated Attention Families—Complete Literature Analysis** **63**
D.1 Executive Summary: Gating for Memory Control . . . . . 63
D.2 1. Gated Linear Attention (GLA) . . . . . 64
D.2.1 Mathematical Foundation . . . . . 64
D.2.2 Hardware-Efficient Training . . . . . 65
D.2.3 Strengths and Limitations . . . . . 65
D.3 2. DeltaNet: Error-Correcting Linear Attention . . . . . 65
D.3.1 Mathematical Foundation . . . . . 65
D.3.2 Efficient Parallel Training . . . . . 66
D.3.3 Strengths and Limitations . . . . . 67
D.4 3. Gated DeltaNet: Synthesis of Gating and Error Correction . . . . . 67
D.4.1 Mathematical Foundation . . . . . 67
D.4.2 Empirical Performance and Trade-offs . . . . . 68
D.5 4. HGRN2: Hierarchical Gating with Outer-Product Expansion . . . . . 68
D.5.1 Mathematical Foundation . . . . . 68
D.5.2 Scaling and Empirical Results . . . . . 69
D.6 5. Forgetting Transformer (FoX): Gating in Softmax Logit Space . . . . . 69
D.6.1 Mathematical Foundation . . . . . 69
D.6.2 Integration with FlashAttention . . . . . 70
D.6.3 Strengths and Limitations . . . . . 70
D.7 6. Gated Attention (Post-SDPA Sigmoid Gating) . . . . . 70
D.7.1 Mathematical Foundation . . . . . 70
D.7.2 Key Findings . . . . . 71
D.8 Comparative Analysis: Taxonomy of Gated Mechanisms . . . . . 71
D.9 Unified Gating Principle . . . . . 73
D.10 Implementation and Practical Guidance . . . . . 73
D.10.1 When to Use Each Architecture . . . . . 73
D.10.2 Hardware Considerations . . . . . 74

**E Annex E: Conceptual Introduction—Transformers and Attention for Beginners** **74**
E.1 What is a Transformer? . . . . . 74
E.2 The Attention Mechanism: An Analogy . . . . . 74
E.3 Practical Example Step-by-Step . . . . . 75
E.3.1 Context 1: “The cat eats fish” . . . . . 75
E.3.2 Context 2: “The restaurant eats into profit margins” . . . . . 75
E.4 How Attention Works Mathematically (Simple Version) . . . . . 75
E.4.1 Step 1: Queries, Keys, and Values . . . . . 75
E.4.2 Step 2: Compatibility . . . . . 75
E.4.3 Step 3: Focus . . . . . 75
E.4.4 Step 4: Combine . . . . . 76
E.5 Multiple Attention Heads: Multiple Perspectives . . . . . 76
E.6 Stacking Layers . . . . . 76
E.7 Why Does It Matter? . . . . . 76
E.8 Practical Challenges . . . . . 76
E.8.1 Computational Complexity . . . . . 77
E.8.2 Memory Requirements . . . . . 77
E.8.3 Device Efficiency . . . . . 77

E.9 Modern Solutions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
E.10 Key Takeaways . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

# 1 Introduction

## 1.1 Motivation and Problem Statement

Transformer architectures have fundamentally transformed the landscape of sequence modeling across natural language processing [38], computer vision, and computational biology. The success of models such as BERT [11], GPT, and their variants has established the transformer as the de facto standard for representation learning in deep learning. However, the rapid proliferation of architectural innovations presents significant practical challenges for researchers and practitioners.

Recent years have witnessed an explosion of alternative attention and sequence-mixing mechanisms, each addressing specific limitations of standard softmax attention: quadratic computational complexity [9], memory-efficient inference requirements [36, 12], content-based selective state management [2], and hardware-aware optimizations [30]. Simultaneously, the optimization literature has diversified beyond classical AdamW, introducing variance-reduction techniques [42, 26, 49], memory-efficient variants [33, 54], schedule-free approaches [10], and second-order preconditioning methods [14, 39].

The practical consequence is a fragmented research ecosystem where experimental comparison across architectures and optimizers requires significant engineering effort. Researchers must implement and debug multiple variants from scratch, ensure consistent training pipelines, and manage complex hyperparameter spaces. This fragmentation hampers reproducibility, slows scientific progress, and increases the barrier to entry for new researchers.

This work addresses these challenges through a unified, configuration-driven experimentation toolkit available at `https://github.com/erickfmm/frankestein-transformer` that provides:

1. **Schema-First Design**: A strict, validated configuration contract that enforces reproducibility while supporting seventeen distinct sequence mixer architectures and twenty-two optimizer families.

2. **Architecture Agnostic Training**: Common training infrastructure supporting dense attention baselines (standard, sigmoid), recurrent alternatives (RetNet, Mamba, ODE-style blocks), adaptive depth routing (Mixture-of-Depths) [57], conditional memory blocks (Engram) [8], sparse attention patterns (Sparse Transformer, Longformer, BigBird, SparseK, NSA, SpargeAttn, FASA), and gated mechanisms (GLA, DeltaNet, Gated DeltaNet, HGRN2, FoX, Gated Softmax).

3. **Optimizer Routing Framework**: Prefixed hyperparameter groups enabling fine-grained control over embeddings, normalization layers, recurrent blocks, attention blocks, and other parameter subsets across diverse optimizers, including the APOLLO family (Apollo, Apollo-Mini, Q-Apollo) [55].

4. **End-to-End Workflows**: Integrated deployment via quantization (ternary weight packing, INT8 activations) and sentence-embedding capabilities inspired by SBERT [31].

5. **Interactive Configuration**: Web-based interface providing schema-driven form rendering, real-time validation, and CLI command generation.

6. **Reliability Tooling**: Comprehensive automated testing with unit-test suites, YAML preset validation, and CI automation across supported Python versions.

The project command surface is:

```
frankestein-transformer
```

with subcommands for encoder and decoder workflows: `train`, `finetune`, `deploy`, `quantize`, `infer`, `sbert-train`, `sbert-infer`, `web-server`.

The `web-server` command launches a Streamlit-based configuration builder that provides:

- Schema-driven form fields with parameter titles and detailed descriptions
- Real-time tooltips and help text for each configuration option
- Live YAML preview and download functionality
- Generated CLI commands for training, deployment, inference, and SBERT workflows

This interactive interface serves as an alternative to manual YAML editing, improving usability for users exploring available configuration options and understanding their impact on model behavior and training dynamics.

## 1.2 Contributions

This work makes the following primary contributions:

1. **Unified Configuration Schema**: A YAML-based schema contract with strict validation that supports seventeen distinct sequence mixer architectures across four categories (dense baselines, recurrent/retentive blocks, sparse attention patterns, and gated mechanisms), alongside adaptive depth and conditional memory controls, while enforcing reproducibility through `additionalProperties: false` constraints.

2. **Comprehensive Architecture Support**: Implementation of modern transformer variants including standard softmax attention [38], sigmoid attention [30], RetNet [36], Mamba [12], ODE-style continuous transformers [52], Titans memory-augmented attention [2], Engram conditional memory layers [8], Mixture-of-Depths token routing [57], sparse attention mechanisms [9, 3, 50, 23, 48, 53, 41], and gated attention architectures [44, 45, 46, 28, 19, 29]. System supports both encoder-mode training with bidirectional attention for masked language modeling (MLM) and decoder-mode training with causal masking for autoregressive next-token prediction.

3. **Optimizer Routing Framework**: Prefixed hyperparameter system enabling per-parameter-group control across twenty-two optimizers, including variance-reduction methods (MARS, Adan, AdEMAMix, Cautious AdamW), memory-efficient variants (Adafactor, GaLore, Lion), schedule-free approaches (Schedule-Free AdamW), curvature-aware methods (Sophia), second-order preconditioners (Shampoo, SOAP), orthogonality-oriented optimizers (Muon, Turbo-Muon), large-batch scaling (LAMB), and the APOLLO family (Apollo, Apollo-Mini, Q-Apollo) [55].

4. **Quantization and Deployment**: Integrated deployment pipeline supporting ternary weight packing and INT8 activation quantization with size estimates following 1.58-bit storage approximation.

5. **Sentence-Embedding Workflows**: SBERT-inspired training and inference pipelines supporting similarity scoring, retrieval, clustering, and persistent embedding export.

6. **Interactive Configuration Interface**: Streamlit-based web server providing schema-driven form generation, real-time validation, inline documentation, and CLI command generation.

7. **Automated Validation Pipeline**: Continuous integration and expanded unit testing that verify model training paths, optimizer/schema integration, and YAML example compatibility.

### 1.3 Reading Guide

This document is organized as a technical reference addressing four operational concerns:

1. **Prerequisites**: Appendix E provides an accessible introduction to transformers and attention mechanisms for readers new to the field.
2. **Configuration Contract**: Section 3.1 describes the YAML schema that enforces valid experiments and Section 3.2 explains validation rules.
3. **Architecture Selection**: Section 3.8 covers normalization options; Section 4.1 provides comprehensive comparison of sequence mixer families; Appendices B, C, and D synthesize supporting literature.
4. **Optimization Dynamics**: Section 5 details optimizer routing and training dynamics; Appendix A provides comprehensive optimizer family analysis.
5. **Deployment and Inference**: Sections 6 and 7 describe quantized deployment and sentence-embedding workflows.

### 1.4 Web-Based Configuration Builder

In addition to direct YAML editing, this project provides a Streamlit-based web interface (accessed via `web-server` command) that improves configuration accessibility and discoverability. The interface presents schema fields with:

- **Schema-driven form rendering** — All fields are dynamically generated from the authoritative schema, ensuring consistency and validation.
- **Inline parameter documentation** — Each form field displays a *title* from the schema as its label, with the *description* shown as a help tooltip on hover.
- **Real-time configuration preview** — Users see live YAML output as they modify form fields, enabling immediate validation feedback.
- **CLI command generation** — The interface generates complete CLI commands for training, deployment, inference, and SBERT workflows based on current configuration.
- **Accessibility improvements** — Tooltips and structured forms make configuration options easier to understand, especially for users new to the project or exploring novel architectures.

This web-based approach addresses common usability barriers in configuration-driven experimentation:

- Reduces need to memorize YAML structure and field names
- Prevents typos through schema validation
- Provides educational context through inline documentation
- Enables rapid experimentation with guided parameter tuning
- Serves as both a configuration tool and a learning resource

The web interface implementation uses Streamlit's form widgets with:

- `st.checkbox()` for binary toggles with help text
- `st.number_input()` for numeric fields with step size and format

- `st.selectbox()` for enum choices with options display
- `st.multiselect()` for array selections from defined options
- `st.text_input()` for string fields
- `st.info()`, `st.caption()` for supplementary information

Schema metadata (title and description fields) are extracted and rendered systematically across all form sections, including:

- Model architecture parameters (hidden size, layers, attention heads, etc.)
- Training runtime settings (batch size, accumulation, scheduler)
- Optimizer configuration with per-parameter-group hyperparameters
- Deployment and quantization options
- SBERT-specific training and inference parameters

## 2 Related Work

This work sits at the intersection of three active research areas: alternative attention architectures, sparse attention mechanisms, and advanced optimization algorithms. This section provides a concise survey; detailed mathematical formulations and algorithmic descriptions are deferred to the appendices.

### 2.1 Sequence Mixer Architectures

#### 2.1.1 Dense Attention Baselines

Standard softmax attention [38] achieves full global context through $n^2$ pairwise interactions but incurs prohibitive memory and compute costs for long sequences. Sigmoid attention [30] replaces row-wise softmax with element-wise sigmoid, yielding faster convergence and up to 17% kernel speedup, though requiring hybrid-norm stabilization at scale. Both serve as dense baselines in this toolkit.

#### 2.1.2 Recurrent and Retentive Architectures

RetNet [36] resolves the "impossible triangle" of parallel training, $\mathcal{O}(1)$ inference, and strong performance via a dual attention–retention formulation. Mamba [12] introduces input-dependent selectivity into state-space models, achieving linear complexity with hardware-aware scan algorithms. ODE-style transformers [52] treat depth as numerical integration over a continuous dynamical system. Titans [2] augments inference with test-time neural memorization for extremely long contexts. Detailed formulations are provided in Appendix B.

#### 2.1.3 Sparse Attention Mechanisms

Sparse attention methods reduce the quadratic bottleneck by restricting token interactions: Sparse Transformer [9] uses factorized strided and fixed patterns ($\mathcal{O}(n\sqrt{n})$); Longformer [3] employs sliding windows with global tokens ($\mathcal{O}(n \cdot w)$); BigBird [50] combines local, random, and global paths; SparseK [23] applies differentiable top-$k$ selection; NSA [48] uses a three-branch hierarchical design; SpargeAttn [53] performs two-stage block-level pruning; and FASA [41] leverages RoPE frequency features for token selection. Complete descriptions are in Appendix C.

#### 2.1.4 Gated Attention Mechanisms

Gated architectures control information flow through learnable gates: GLA [44] adds data-dependent diagonal gating to linear attention; DeltaNet [45] applies the delta learning rule to recurrent state updates; Gated DeltaNet [46] synthesizes both mechanisms; HGRN2 [28] introduces hierarchical forget gates with outer-product state expansion; FoX [19] embeds forget gates directly into softmax attention; and Gated Softmax [29] applies post-SDPA channel gating. Full mathematical derivations appear in Appendix D.

### 2.2 Optimization Algorithms

Optimization of highly parameterized transformer architectures requires navigating non-convex loss landscapes with heterogeneous Hessian spectra. The literature has diversified into several families beyond the classical AdamW baseline [22]:

- **Variance reduction and momentum**: RAdam [21], Adan [42], ADOPT [37], AdEMAMix [26], MARS [49], and Cautious optimizers [18] address early-step instability, convergence guarantees, and dual-EMA history mixing.
- **Memory-efficient**: Adafactor [33], GaLore [54], Lion [7], and APOLLO [55] reduce optimizer-state memory through factorization, low-rank projection, or sign-based updates.
- **Schedule-free and parameter-free**: Schedule-Free AdamW [10] and Prodigy [24] absorb scheduler or learning-rate tuning into the optimizer dynamics.
- **Second-order and curvature-aware**: Shampoo [14], SOAP [39], and Sophia [20] incorporate approximate second-order information.
- **Geometry-oriented**: Muon [34] and Turbo-Muon [4] reshape update geometry through orthogonalization.

Detailed algorithmic descriptions, pseudocode, and a comprehensive complexity comparison are provided in Appendix A.

## 3 System Design and Architecture

### 3.1 Configuration-Centric Architecture

The core design choice in this repository is that experimentation is *schema first*. Instead of exposing a large number of loosely checked flags, the project forces model topology, optimizer family, training limits, and telemetry options through a single validated configuration document. This reduces ambiguity when reproducing results and makes it possible to compare many architectures under a consistent operational interface.

The authoritative contract is `configs/schema.yaml`. It enforces three top-level objects:

- `model_class`
- `model`
- `training`

**Model Class Selection.** The `model_class` field determines the architectural variant instantiated by the training pipeline. Three options are supported:

- **frankenstein**: Mixed-architecture encoder models supporting diverse attention mechanisms (standard, sigmoid, retentive, state-space, sparse, and gated mixers) with MoE (Mixture of Experts) and advanced features. Optimized for bidirectional encoder-style training with masked language modeling (MLM) objectives.
- **mini**: Simplified encoder variant designed for smaller-scale training scenarios. Provides reduced parameter overhead and faster iteration for experimentation and prototyping.
- **frankesteindecoder**: Autoregressive causal decoder for LLM-style next-token generation. Enables causal attention masking for sequential text generation tasks. When this class is selected, runtime enforces `mode='decoder'`.

**Training Mode Selection.** The `model.mode` field controls attention masking behavior across the model:

- **encoder**: Uses bidirectional attention where all tokens attend to all other tokens in the sequence. Suitable for masked language modeling (MLM) pre-training tasks where the model learns to predict randomly masked tokens based on full context.
- **decoder**: Uses causal masking where each token can only attend to previous tokens in the sequence. Required for autoregressive (AR) next-token prediction tasks such as language modeling and text generation. When `model_class='frankesteindecoder'`, the system automatically forces `mode='decoder'` at runtime.

This dual architecture support enables the system to handle both encoder-style pre-training (MLM on bidirectional contexts) and decoder-style generation (causal autoregressive prediction) through a unified configuration interface.

The `model.layer_pattern` supports legacy, sparse, and gated blocks:

- **Retentive Network (RetNet)** — internal reference: `sun_retentive_2023` — code name: `retnet`, `retnet_attn`
- **Mamba (Selective State Space Model)** — internal reference: `gu_mamba_2023` — code name: `mamba`
- **ODE-style Continuous Depth Block** — internal reference: `zhang_continuous_2021` — code name: `ode`
- **Titans Memory-Augmented Attention** — internal reference: `behrouz_titans_2025` — code name: `titan_attn`
- **Standard Softmax Attention** — internal reference: `vaswani_attention_2017` — code name: `standard_attn`
- **Sigmoid Self-Attention** — internal reference: `ramapuram_theory_2024` — code name: `sigmoid_attn`
- **Sparse Transformer** — internal reference: `child_sparse_transformer_2019` — code name: `sparse_transformer_attn`
- **Longformer** — internal reference: `beltagy_longformer_2020` — code name: `longformer_attn`
- **BigBird** — internal reference: `zaheer_bigbird_2020` — code name: `bigbird_attn`

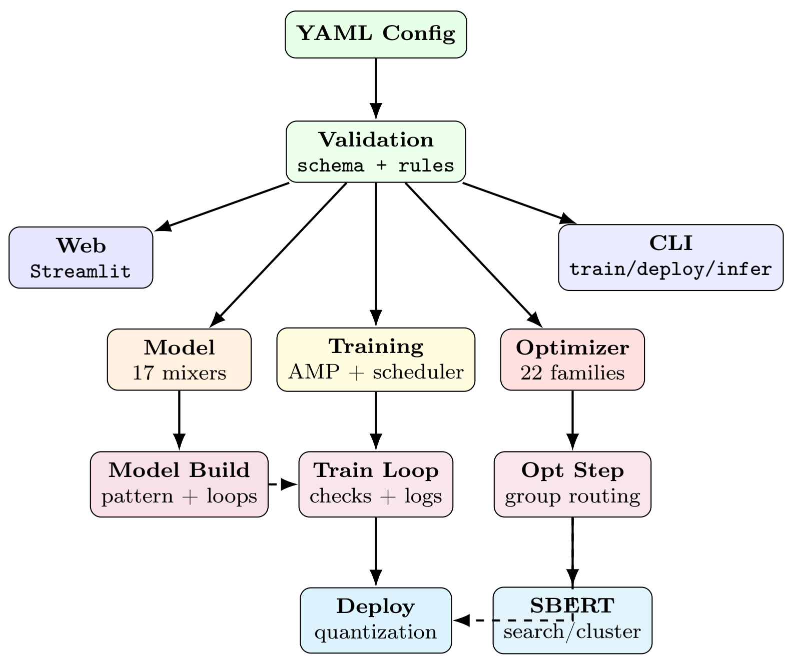

Figure 1: System architecture: configuration flows through validation, partitions into model/training/optimizer, executes runtime, produces deployment/SBERT artifacts.

- **SparseK Attention** — internal reference: `lou_sparsek_2024` — code name: `sparsek_attn`
- **Native Sparse Attention (NSA)** — internal reference: `yuan_nsa_2025` — code name: `nsa_attn`
- **SpargeAttn** — internal reference: `zhang_spargeattn_2025` — code name: `sparge_attn`
- **FASA (Frequency-aware Sparse Attention)** — internal reference: `wang_fasa_2026` — code name: `fasa_attn`
- **Gated Linear Attention (GLA)** — internal reference: `yang_gla_2023` — code name: `gla_attn`
- **DeltaNet** — internal reference: `yang_deltanet_2024` — code name: `deltanet_attn`
- **Gated DeltaNet** — internal reference: `yang_gated_deltanet_2024` — code name: `gated_deltanet_attn`
- **HGRN2** — internal reference: `qin_hgrn2_2024` — code name: `hgrn2_attn`
- **Forgetting Transformer (FoX)** — internal reference: `lin_forgetting_transformer_2025` — code name: `fox_attn`
- **Gated Softmax Attention** — internal reference: `qiu_gated_attention_2025` — code name: `gated_softmax_attn`

which corresponds to current attention and sequence-mixer literature [36, 12, 52, 2, 30, 38, 9, 3, 50, 23, 48, 53, 41, 44, 45, 46, 28, 19, 29]

The `training.optimizer.optimizer_class` supports a broad optimizer family: `sgd_momentum`, `adamw`, `adafactor`, `galore_adamw`, `prodigy`, `lion`, `sophia`, `muon`, `turbo_muon`, `radam`, `adan`, `adopt`, `ademamix`, `mars_adamw`, `cautious_adamw`, `lamb`, `schedulefree_adamw`, `shampoo`, `soap`, `apollo`, `apollo_mini`, and `q_apollo`.

## 3.2 Schema Scope and Validation Rules

The schema is strict: top-level and nested objects set `additionalProperties: false`. This guarantees that unknown keys fail fast instead of being silently ignored. The `training.optimizer.parameters` object is additionally constrained by optimizer-specific prefix rules through `allOf`+`if/then` pattern checks.

Normalization values currently accepted by schema are:

$$\texttt{norm\_type} \in \{\texttt{layer\_norm}, \texttt{dynamic\_tanh}, \texttt{derf}\}$$

Thus, `rms_norm` is **not** a valid schema value in the current contract.

## 3.3 Complete Model Feature Inventory

| **Field** | **Type/Range** | **Meaning** |
|---|---|---|
| `model_class` | enum | Architecture variant: `frankenstein` (mixed-architecture encoder), `mini` (simplified encoder), `frankesteindecoder` (autoregressive decoder). |
| `model.mode` | enum | Attention mode: `encoder` (bidirectional, for MLM) or `decoder` (causal, for AR). When `model_class='frankesteindecoder'`, forces `mode='decoder'`. |
| `vocab_size` | int $\geq 1$ | Vocabulary size. |
| `hidden_size` | int $\geq 1$ | Hidden dimension. |
| `num_layers` | int $\geq 1$ | Physical layer count. |
| `num_loops` | int $\geq 1$ | Logical loop count (looped blocks). |
| `num_heads` | int $\geq 1$ | Attention heads. |
| `retention_heads` | int $\geq 1$ | Retention heads for RetNet-style mixers. |
| `num_experts` | int $\geq 1$ | MoE expert count. |
| `top_k_experts` | int $\geq 1$ | Top-$k$ expert routing in MoE. |
| `dropout` | float $[0, 1]$ | Global dropout. |
| `layer_pattern` | array enum | Ordered block list: legacy (`retnet`, `retnet_attn`, `mamba`, `ode`, `titan_attn`, `standard_attn`, `sigmoid_attn`), sparse (`sparse_transformer_attn`, `longformer_attn`, `bigbird_attn`, `sparsek_attn`, `nsa_attn`, `sparge_attn`, `fasa_attn`), and gated (`gla_attn`, `deltanet_attn`, `gated_deltanet_attn`, `hgrn2_attn`, `fox_attn`, `gated_softmax_attn`). |
| `ode_solver` | enum | `rk4` or `euler`. |
| `ode_steps` | int $\geq 1$ | ODE integration steps. |
| `use_bitnet` | bool | Enable low-bit BitLinear path. |
| `norm_type` | enum | `layer_norm`, `dynamic_tanh`, `derf`. |
| `use_factorized_embedding` | bool | Enable factorized embeddings. |
| `factorized_embedding_dim` | int $\geq 1$ | Reduced embedding dimension for factorization. |
| `use_embedding_conv` | bool | Enable Conv1d over embedding stream. |
| `embedding_conv_kernel` | int $\geq 1$ | Conv1d kernel size. |
| `hope_base` | float $\geq 0$ | HoPE base value (optional in schema). |
| `hope_damping` | float $\geq 0$ | HoPE damping (optional in schema). |
| `use_hope` | bool | Apply HoPE in `titan_attn`. |
| `use_moe` | bool | Enable MoE FFN routing path. |
| `ffn_hidden_size` | int $\geq 1$ | FFN intermediate width. |

| Field | Type/Range | Meaning |
|---|---|---|
| ffn_activation | enum | silu or gelu. |

Looped depth induced by schema is:

$$L_{\text{logical}} = \texttt{num\_layers} \times \texttt{num\_loops}$$

which is the configuration-level definition of looped blocks.

## 3.4 Training Task Types

The `training.task` field determines the training objective, working in conjunction with `model.mode` to define how the model learns:

**Masked Language Modeling (MLM).**

- **Mode**: Requires `mode='encoder'` for bidirectional attention.
- **Objective**: Randomly mask tokens in input sequence (typically 15%) and train model to predict masked tokens based on full bidirectional context.
- **Use Case**: Pre-training encoders for representation learning, following BERT-style methodology [11]. Model learns bidirectional representations capturing context from both left and right.
- **Configuration**: Uses `mlm_probability` parameter to control masking fraction.

**Autoregressive (AR) Next-Token Prediction.**

- **Mode**: Requires `mode='decoder'` for causal masking.
- **Objective**: Train model to predict next token in sequence given all previous tokens only.
- **Use Case**: Language generation and LLM-style tasks following GPT methodology. Model learns sequential dependencies with causal attention where each token can only attend to preceding tokens.
- **Model Class**: Typically uses `model_class='frankesteindecoder'` for autoregressive decoder architectures.

This dual task support enables unified experimentation across both encoder-style pre-training (MLM for bidirectional understanding) and decoder-style generation (AR for sequential text production) within the same codebase.

## 3.5 Complete Training Feature Inventory

| Field | Type/Range | Meaning |
|---|---|---|
| batch_size | int $\geq 1$ | Loader batch size. |
| dataloader_workers | int $\geq 0$ | PyTorch dataloader workers. |
| max_length | int $\geq 1$ | Sequence length cap. |
| task | enum | Training objective: `mlm` (masked language modeling) or `sbert` (sentence embedding). |
| mlm_probability | float $[0, 1]$ | MLM masking probability (applies only when `task='mlm'`). |

| **Field** | **Type/Range** | **Meaning** |
|---|---|---|
| `max_samples` | int $\geq 1$ | Maximum streamed samples. |
| `dataset_batch_size` | int $\geq 1$ | Internal streaming dataset chunk size. |
| `num_workers` | int $\geq 0$ | Streaming dataset workers. |
| `cache_dir` | string | Dataset cache directory. |
| `local_parquet_dir` | string | Optional local parquet path. |
| `prefer_local_cache` | bool | Prefer local cache when available. |
| `stream_local_parquet` | bool | Stream from local parquet mode. |
| `use_amp` | bool | Mixed precision toggle. |
| `gradient_accumulation_steps` | int $\geq 1$ | Effective batch through accumulation. |
| `optimizer` | object | Contains `optimizer_class` and prefixed `parameters`. |
| `scheduler_total_steps` | int $\geq 1$ | Scheduler horizon. |
| `scheduler_warmup_ratio` | float $[0, 1]$ | Warmup ratio. |
| `scheduler_type` | enum | `cosine`, `constant`, `linear_warmup_then_constant`. |
| `grad_clip_max_norm` | float $\geq 0$ | Global norm clipping threshold. |
| `inf_post_clip_threshold` | float $\geq 0$ | Exploding-gradient guard threshold after clipping. |
| `max_nan_retries` | int $\geq 0$ | Retry budget for NaN/Inf instability. |
| `checkpoint_every_n_steps` | int $\geq 1$ | Rolling checkpoint frequency. |
| `max_rolling_checkpoints` | int $\geq 1$ | Number of rolling checkpoints to keep. |
| `num_best_checkpoints` | int $\geq 1$ | Number of best checkpoints tracked. |
| `nan_check_interval` | int $\geq 1$ | NaN/Inf check cadence. |
| `log_gradient_stats` | bool | Enable gradient statistics logging. |
| `gradient_log_interval` | int $\geq 1$ | Gradient logging cadence. |
| `csv_log_path` | string | Step-level CSV output path. |
| `csv_rotate_on_schema_change` | bool | Rotate CSV if logging schema changes. |
| `gpu_metrics_backend` | enum | `nvml` or `none`. |
| `nvml_device_index` | int $\geq 0$ | Device index for NVML telemetry. |
| `enable_block_grad_norms` | bool | Include per-block gradient norm telemetry. |
| `telemetry_log_interval` | int $\geq 1$ | Heavy telemetry interval (optimizer steps). |
| `use_galore` | bool | Enable GaLore strategy. |
| `galore_rank` | int $\geq 1$ | GaLore low-rank projection dimension. |
| `galore_update_interval` | int $\geq 1$ | Projection refresh interval. |
| `galore_scale` | float $\geq 0$ | Gradient scaling in projected space. |
| `galore_max_dim` | int $\geq 1$ | Maximum tensor dimension for GaLore projection. |

## 3.6 Optimizer Prefix Contract (Full)

Supported `optimizer_class` values are: `sgd_momentum`, `adamw`, `adafactor`, `galore_adamw`, `prodigy`, `lion`, `sophia`, `muon`, `turbo_muon`, `radam`, `adan`, `adopt`, `ademamix`, `mars_adamw`, `cautious_adamw`, `lamb`, `schedulefree_adamw`, `shampoo`, `soap`, `apollo`, `apollo_mini`, `q_apollo`.

Shared per-group suffix families (all prefixed by optimizer name) are:

- LR groups: `lr_embeddings`, `lr_norms`, `lr_ode`, `lr_retnet`, `lr_mamba`, `lr_attention`, `lr_other`
- Weight decay groups: `wd_embeddings`, `wd_norms`, `wd_ode`, `wd_retnet`, `wd_mamba`, `wd_attention`, `wd_other`
- Beta groups: `betas_embeddings`, `betas_norms`, `betas_ode`, `betas_retnet`, `betas_mamba`, `betas_attention`, `betas_other`
- Epsilon groups: `eps_embeddings`, `eps_norms`, `eps_ode`, `eps_retnet`, `eps_mamba`, `eps_attention`, `eps_other`

Optimizer-specific global suffixes:

- `sgd_momentum`: `momentum`, `nesterov`
- `adafactor`: `beta2_decay`, `clip_threshold`, `eps1`, `eps2`
- `galore_adamw`: `rank`, `update_proj_gap`
- `prodigy`: `d_coef`
- `sophia`: `rho`, `update_k`
- `muon` / `turbo_muon`: `momentum`, `nesterov`, `ns_steps`, `ns_eps`
- `cautious_adamw`: `cautious_clip`
- `apollo`: `rank`, `update_proj_gap`, `scale`, `scale_type`, `proj_type`, `scale_front`, `disable_nl`
- `apollo_mini`: `update_proj_gap`, `scale`, `proj_type`, `scale_front`, `disable_nl`
- `q_apollo`: `rank`, `update_proj_gap`, `scale`, `scale_type`, `proj_type`, `scale_front`, `disable_nl`, `quant_bits`

All other classes in the list above accept only prefixed shared groups.

## 3.7 Training Safety and Runtime Semantics

Schema-level safety features include accumulation, clipping, post-clip explosion checks, and NaN retries:

$$g_{\text{acc}} = \frac{1}{K}\sum_{i=1}^{K} g_i, \quad K = \texttt{gradient\_accumulation\_steps}$$

$$g_{\text{clip}} = g_{\text{acc}} \cdot \min\left(1, \frac{\tau}{\|g_{\text{acc}}\|_2 + \epsilon}\right), \quad \tau = \texttt{grad\_clip\_max\_norm}$$

then overflow guards use `inf_post_clip_threshold` and retry logic bounded by `max_nan_retries`.

## 3.8 Normalization Variants: RMSNorm, Dynamic Tanh, and Dynamic Erf

Normalization determines how activation scale is controlled across depth. In this repository, normalization is not only a modeling choice but also a schema compatibility question, because only certain values are currently accepted by `norm_type`. The three formulations most relevant to this codebase are:

## 3.9 RMSNorm

RMSNorm removes mean-centering and only rescales by root mean square magnitude [51]:

$$\text{RMS}(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}, \qquad y_i = \gamma_i \frac{x_i}{\text{RMS}(x)}$$

Compared with LayerNorm, RMSNorm is computationally simpler (no subtraction of feature mean) and is often used when reducing normalization overhead is important.

**Algorithm 1** Schema-Driven Training Step with Stability Controls

```
Require: Batch stream, config $C$
 1: Initialize retry counter $r \leftarrow 0$
 2: for each optimizer step do
 3:     Accumulate gradients for $K = C$.gradient_accumulation_steps micro-batches
 4:     Apply global norm clipping with $\tau = C$.grad_clip_max_norm
 5:     if post-clip gradient exceeds $C$.inf_post_clip_threshold or NaN/Inf detected then
 6:         if $r < C$.max_nan_retries then
 7:             restore safe state / skip step; $r \leftarrow r + 1$
 8:             continue
 9:         else
10:             stop training with failure state
11:         end if
12:     end if
13:     run optimizer step selected by optimizer_class
14:     update scheduler (cosine, constant, or linear_warmup_then_constant)
15:     if step mod checkpoint_every_n_steps$= 0$ then
16:         save rolling checkpoint and prune to max_rolling_checkpoints
17:     end if
18:     update best checkpoints up to num_best_checkpoints
19:     emit CSV + telemetry following gradient_log_interval and telemetry_log_interval
20: end for
```

## 3.10 Dynamic Tanh (DyT)

Dynamic Tanh proposes replacing explicit normalization with a bounded elementwise map [56]:

$$\mathrm{DyT}(x) = \tanh(\alpha x)$$

where $\alpha$ is learned. The core idea is that bounded nonlinear contraction can provide stable signal scaling without explicitly computing per-token normalization statistics.

## 3.11 Dynamic Erf (Derf)

Derf extends the same normalization-free direction by using an error-function based map [6]:

$$\mathrm{Derf}(x) = \mathrm{erf}(\alpha x + s)$$

with learnable scale/shift. Reported results in the cited work indicate stronger performance than DyT and common normalization baselines across multiple domains.

## 3.12 Schema Implications

Current configuration contract in this repository allows:

$$\texttt{norm\_type} \in \{\texttt{layer\_norm}, \texttt{dynamic\_tanh}, \texttt{derf}\}$$

so DyT and Derf are directly available in schema-driven runs, while RMSNorm is not currently an accepted enum value and would require code/schema extension.

| **Method** | **Formula** | **Stats Needed** | **Notes** |
|---|---|---|---|
| RMSNorm | $\gamma_i x_i / \sqrt{\frac{1}{d}\sum_j x_j^2 + \epsilon}$ | RMS only | Lower overhead; widely used baseline [51]. |

| Method | Formula | Stats Needed | Notes |
|---|---|---|---|
| Dynamic Tanh | $\tanh(\alpha x)$ | none | Normalization-free bounded transform; drop-in replacement [56]. |
| Dynamic Erf | $\mathrm{erf}(\alpha x + s)$ | none | Normalization-free alternative; improves over DyT [6]. |

## 4 Architecture Taxonomy and Implementation

### 4.1 Attention and Sequence-Mixer Families

This system implements seventeen distinct sequence mixer architectures organized into five functional categories reflecting research trends in sequence modeling design. The taxonomical organization reflects evolving understanding of how to balance expressivity, computational efficiency, and memory constraints.

1. **Dense Attention Baselines**: Standard softmax attention and sigmoid attention provide full global contextualization at quadratic computational cost, serving as reference baselines for comparison with more efficient alternatives.

2. **Recurrent and Retentive Architectures**: RetNet, Mamba, ODE-style blocks, and Titans maintain state representations enabling $\mathcal{O}(1)$ inference cost while preserving expressivity through recurrent dynamics, selective parameters, or test-time memory adaptation.

3. **Sparse Attention Patterns**: Seven sparse variants (Sparse Transformer, Longformer, BigBird, SparseK, NSA, SpargeAttn, FASA) reduce quadratic complexity through structured sparsity, token selection, or training-free pruning strategies.

4. **Gated Memory Mechanisms**: Six gated architectures (GLA, DeltaNet, Gated DeltaNet, HGRN2, FoX, Gated Softmax) introduce data-dependent control over memory retention, forgetting, and update strength.

### 4.2 Standard Attention

Given projected matrices $(Q, K, V)$:

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

This is the baseline mechanism for content routing [38].

### 4.3 Sigmoid Attention

Sigmoid attention removes row-wise probability normalization:

$$\mathrm{SigmoidAttn}(Q, K, V) = \sigma\left(\frac{QK^\top}{\sqrt{d_k}} + b\right) V$$

and has different training stability requirements, often with additional normalization [30].

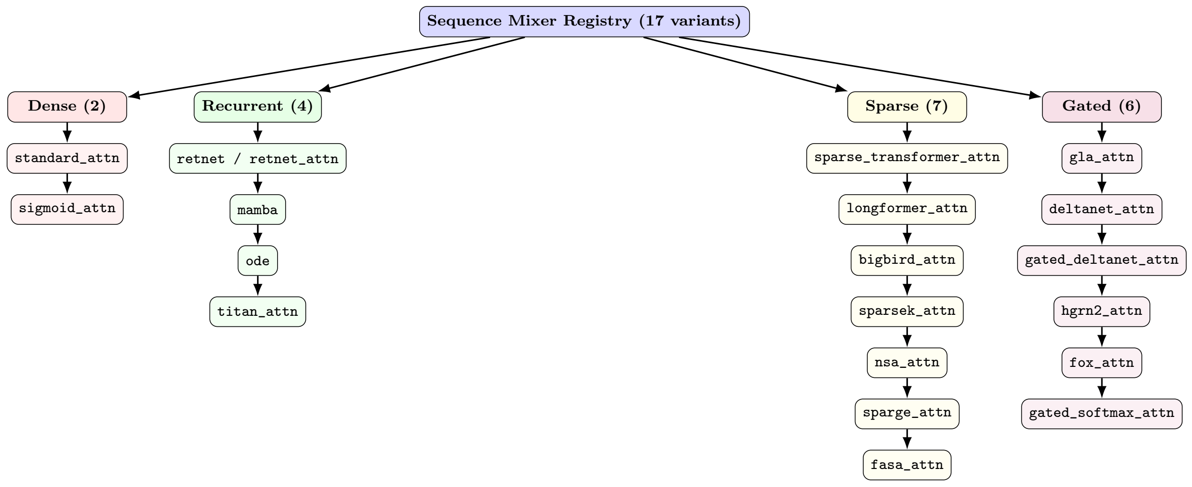

Figure 2: Comprehensive taxonomy of seventeen supported sequence mixer architectures. Dense baselines provide full global routing at quadratic cost. Recurrent architectures enable constant-time inference through state compression. Sparse variants reduce complexity through structured patterns or token selection. Gated mechanisms introduce data-dependent control over memory retention and forgetting.

### 4.4 Retentive Formulation

RetNet uses retention with decay matrix $D$:

$$\text{Retention}(Q, K, V) = \left(QK^\top \odot D\right) V$$

with recurrent form:

$$S_n = \gamma S_{n-1} + k_n^\top v_n, \qquad o_n = q_n S_n$$

enabling low-cost recurrent inference [36, 43].

### 4.5 Selective SSM (Mamba)

Discrete selective state-space recurrence is:

$$h_t = \bar{A}_t h_{t-1} + \bar{B}_t x_t, \qquad y_t = C_t h_t$$

where $(\bar{A}_t, \bar{B}_t, C_t)$ depend on input, preserving linear-time scaling with hardware-aware scan [12, 15].

### 4.6 ODE-style Continuous Updates

Continuous-depth framing:

$$\frac{dh(t)}{dt} = f_\theta(h(t), t)$$

with practical RK integrators for discrete execution [52].

### 4.7 Test-time Memory (Titans)

A memory-augmented update can be written:

$$M_t = (1 - \alpha_t) M_{t-1} + S_t, \qquad S_t = \eta_t S_{t-1} - \theta_t \nabla \ell(M_{t-1}; x_t)$$

to adapt memory at inference time [2, 1].

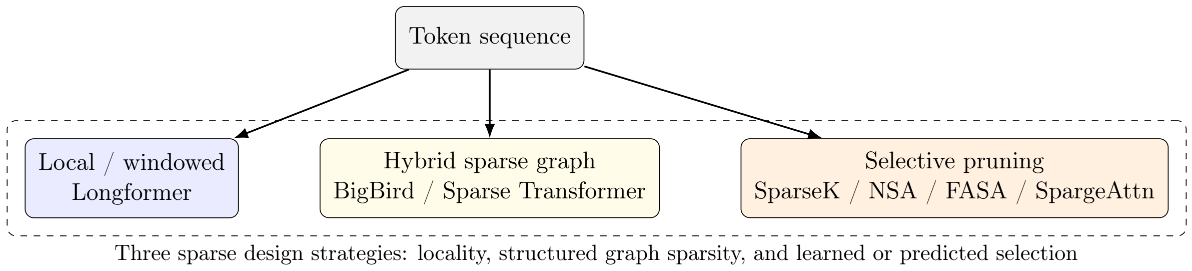
Token sequence
Local / windowed
Longformer
Hybrid sparse graph
BigBird / Sparse Transformer
Selective pruning
SparseK / NSA / FASA / SpargeAttn
Three sparse design strategies: locality, structured graph sparsity, and learned or predicted selection

Figure 3: Conceptual map of sparse attention design choices used in the codebase. Different methods reduce cost by restricting neighborhoods, constructing sparse graphs, or selecting only high-value tokens/blocks.

### 4.8 Sparse and Gated Extensions in the Current Codebase

The mixer registry now includes sparse blocks: `sparse_transformer_attn`, `longformer_attn`, `bigbird_attn`, `sparsek_attn`, `nsa_attn`, `sparge_attn`, and `fasa_attn`; and gated blocks: `gla_attn`, `deltanet_attn`, `gated_deltanet_attn`, `hgrn2_attn`, `fox_attn`, and `gated_softmax_attn`.

The implementation enforces an explicit execution policy for training-free sparse methods: `fasa_attn` and `sparge_attn` are eval/inference-only and raise runtime errors if used while the model is in training mode.

### 4.9 Implemented Sparse Attention Blocks (Detailed)

This codebase includes seven sparse attention families [9, 3, 50, 23, 48, 53, 41].

**Sparse Transformer (`sparse_transformer_attn`).** Uses factorized sparse masks (strided + fixed) to approximate dense connectivity at lower cost than full $\mathcal{O}(n^2)$ attention:

$$\mathrm{Attn}_i = \mathrm{softmax}\left(\frac{q_i K_{A_i}^\top}{\sqrt{d_k}}\right) V_{A_i}$$

where $A_i$ is the sparse neighborhood induced by stride/fixed rules. [9]

**Longformer (`longformer_attn`).** Uses sliding-window locality with optional global tokens:

$$A_i = \{j : |i - j| \le w/2\} \cup \mathcal{G}$$

yielding linear scaling in sequence length for fixed window $w$. [3]

**BigBird (`bigbird_attn`).** Combines local windows, random links, and global tokens:

$$A_i = A_i^{\mathrm{window}} \cup A_i^{\mathrm{random}} \cup A_i^{\mathrm{global}}$$

to preserve strong long-context connectivity with sparse computation. [50]

**SparseK (`sparsek_attn`).** Uses a differentiable top-$k$ style projection over importance scores before attention, so only selected KV pairs participate in the expensive dot-product path. [23]

**NSA (`nsa_attn`).** Implements a three-branch sparse design: compressed branch, selected branch, and local window branch, then combines them with learned gates:

$$o_t = \sum_{c \in \{\mathrm{cmp},\mathrm{sel},\mathrm{win}\}} g_t^c \,\mathrm{Attn}(q_t, \tilde{K}_t^c, \tilde{V}_t^c)$$

[48]

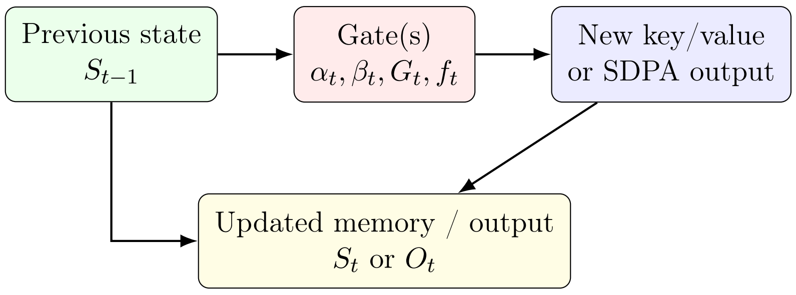

Figure 4: Generic gating template. A gate can decay existing memory, regulate write strength, or modulate dense attention outputs, depending on the block family.

**SpargeAttn (`sparge_attn`).** Two-stage training-free block filtering: first predicts negligible block interactions, then applies softmax-aware pruning to remove low-contribution blocks. [53]

**FASA (`fasa_attn`).** Frequency-aware training-free attention: uses dominant RoPE frequency chunks for token importance prediction, then applies full attention only on selected tokens. [41]

| **Block** | **Trainable** | **Asymptotic Trend** | **Primary Sparsity Unit** | **Current Integration Notes** | **Ref** |
|---|---|---|---|---|---|
| Sparse Transformer | Yes | sub-quadratic | mask pattern (token-level) | Factorized strided/fixed masks inside SDPA pipeline. | [9] |
| Longformer | Yes | linear in $n$ (fixed $w$) | sliding window + global tokens | Window mask with optional global indices. | [3] |
| BigBird | Yes | near-linear | window + random + global edges | Randomized sparse mask plus local/global paths. | [50] |
| SparseK | Yes | linear-like (selected KV) | differentiable top-$k$ KV selection | Learned score net + SparseK projection + gathered KV attention. | [23] |
| NSA | Yes | reduced-token multi-branch | compressed blocks + selected blocks + local window | Three sparse branches gated into one output tensor. | [48] |
| SpargeAttn | No (training-free) | sparse block dependent | block-level predicted sparsity | Eval-only in this repo; raises in training mode. | [53] |
| FASA | No (training-free) | selected-token dependent | dominant frequency chunks + selected tokens | Eval-only in this repo; raises in training mode. | [41] |

## 4.10 Implemented Gated Attention Blocks (Detailed)

This codebase includes seven gated blocks [44, 45, 46, 36, 28, 19, 29].

The unifying idea is that gating controls *what information survives*. Some gates act on recurrent state updates (GLA, DeltaNet variants, HGRN2), while others modify the full attention path itself (FoX and Gated Softmax). This makes gating especially useful when the model must trade off recall, recency, and bounded memory.

**GLA (`gla_attn`).** Gated Linear Attention applies data-dependent multiplicative decay in recurrent state updates:

$$S_t = G_t \odot S_{t-1} + v_t k_t^\top, \qquad o_t = S_t q_t$$

to control memory accumulation. [44]

**DeltaNet (`deltanet_attn`).** Uses a delta-rule error-correcting write with learned write strength $\beta_t$:

$$S_t = S_{t-1}(I - \beta_t k_t k_t^\top) + \beta_t v_t k_t^\top$$

which improves targeted memory replacement. [45]

**Gated DeltaNet (`gated_deltanet_attn`).** Adds decay gate $\alpha_t$ on top of delta-rule writes:

$$S_t = \alpha_t S_{t-1}(I - \beta_t k_t k_t^\top) + \beta_t v_t k_t^\top$$

for both global forgetting and local corrective updates. [46]

**RetNet Attn Alias (`retnet_attn`).** Provides an explicit gated-package alias wrapping multi-scale retention behavior for naming consistency in layer registries. [36]

**HGRN2 (`hgrn2_attn`).** Uses lower-bounded forget gates with outer-product state expansion:

$$S_t = \mathrm{diag}(g_t) S_{t-1} + v_t k_t^\top$$

to increase recurrent state expressiveness while remaining efficient. [28]

**FoX (`fox_attn`).** Injects token-wise forget bias directly into softmax logits:

$$O = \mathrm{softmax}(QK^\top + D)V$$

where $D$ is derived from cumulative log-forget gates. [19]

**Gated Softmax (`gated_softmax_attn`).** Applies a post-SDPA sigmoid gate:

$$Y' = \mathrm{SDPA}(Q, K, V) \odot \sigma(XW_g)$$

which adds multiplicative channel gating without replacing softmax attention. [29]

| **Block** | **State Type** | **Gate Mechanism** | **Softmax Path** | **Current Integration Notes** | **Ref** |
|---|---|---|---|---|---|
| GLA | matrix recurrent state | data-dependent multiplicative decay | No (linear recurrent) | Recurrent update with low-rank gate projection. | [44] |
| DeltaNet | matrix recurrent state | write gate ($\beta$) | No (linear recurrent) | Delta-rule correction with normalized Q/K. | [45] |
| Gated DeltaNet | matrix recurrent state | decay + write gates $(\alpha, \beta)$ | No (linear recurrent) | Combined forgetting and targeted writing. | [46] |
| RetNet Attn | matrix recurrent state | fixed multi-scale decay | No (retention) | Alias wrapper over existing RetNet mixer. | [36] |
| HGRN2 | matrix recurrent state | lower-bounded forget gate | No (linear recurrent) | Hierarchical recurrent gating with outer products. | [28] |

**Algorithm 2** Pattern-Driven Mixer Forward (Conceptual)

```
Require: Hidden states $H$, pattern $P$, layer index $\ell$
1: $m \leftarrow P[\ell \bmod |P|]$
2: if $m \in \{$fasa_attn, sparge_attn$\}$ and model is in training mode then
3:     raise configuration/runtime error (training-free block in train mode)
4: else if $m =$ standard_attn then
5:     $H \leftarrow$ softmax-attention($H$)
6: else if $m =$ sigmoid_attn then
7:     $H \leftarrow$ sigmoid-attention($H$)
8: else if $m \in \{$retnet, retnet_attn$\}$ then
9:     $H \leftarrow$ retention($H$)
10: else if $m =$ mamba then
11:     $H \leftarrow$ selective-ssm($H$)
12: else if $m =$ ode then
13:     $H \leftarrow$ rk-step($H$)
14: else if $m$ is a sparse attention key then
15:     $H \leftarrow$ sparse-attention-family($H$)
16: else if $m$ is a gated attention key then
17:     $H \leftarrow$ gated-attention-family($H$)
18: else
19:     $H \leftarrow$ memory-augmented-attn($H$)
20: end if
21: return $H$
```

| **Block** | **State Type** | **Gate Mechanism** | **Softmax Path** | **Current Integration Notes** | **Ref** |
|---|---|---|---|---|---|
| FoX | full attention matrix | logit-space forget gate | Yes | Forget bias added before softmax. | [19] |
| Gated Softmax | full attention matrix | post-attention sigmoid gate | Yes | Sigmoid gating applied after SDPA output. | [29] |

## 5 Optimizer Families and Training Dynamics

Optimization of highly parameterized transformer architectures presents significant challenges due to non-convex loss landscapes, saddle points, and block heterogeneity across parameter groups. This system addresses these challenges through a unified framework supporting twenty-two optimizer families spanning six algorithmic categories: (1) classical baselines (SGD, AdamW), (2) advanced momentum and variance reduction (Adan, ADOPT, AdEMAMix, MARS, Cautious), (3) memory-efficient variants (Adafactor, GaLore, Lion, APOLLO, APOLLO-Mini, Q-APOLLO), (4) schedule-free and parameter-free methods (Schedule-Free AdamW, Prodigy) plus large-batch scaling through LAMB, (5) curvature-aware and second-order (Sophia, Shampoo, SOAP), and (6) geometry-oriented (Muon, Turbo-Muon). Detailed algorithmic descriptions, pseudocode, and a memory/complexity comparison table for all optimizers are provided in Appendix A.

## 6 Quantization and Deployment

The deploy stack uses ternary weight packing plus INT8 activation quantization for efficient artifacts.

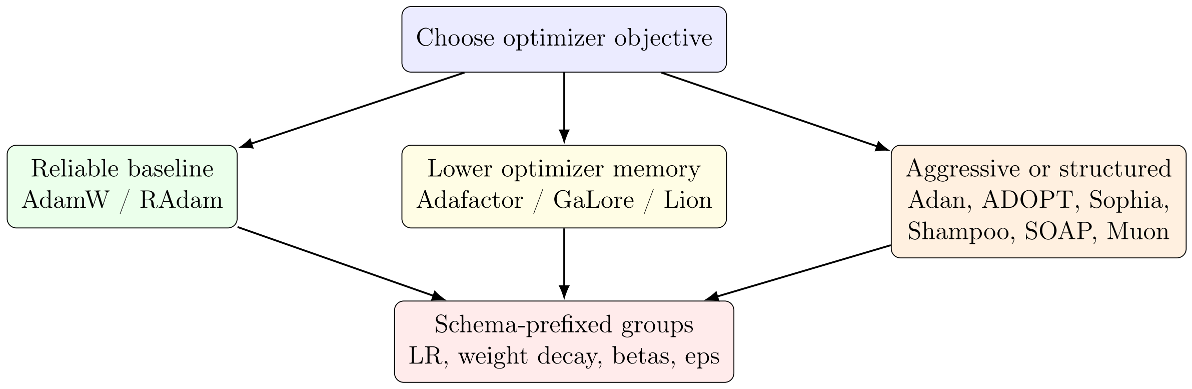

Figure 5: Optimizer selection in practice: the class determines the update rule, while the schema controls how hyperparameters are applied across embeddings, norms, recurrent blocks, attention blocks, and other parameters.

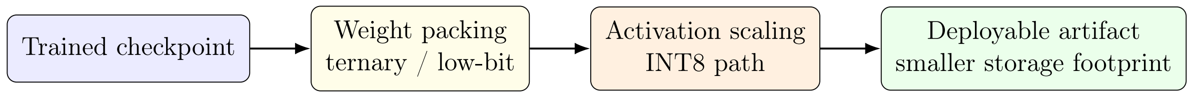

Figure 6: Deployment path from a trained checkpoint to a compact artifact. The codebase treats quantization as a deploy-stage transformation rather than a separate model family.

## 6.1 Ternary Quantization

Given weight tensor $W$, a practical scaling is:

$$s = \mathrm{mean}(|W|), \qquad \tilde{W} = \mathrm{clip}\left(\mathrm{round}\left(\frac{W}{s}\right), -1, 1\right)$$

which approximates BitNet-style low-bit updates [40]. The packed mapping uses two bits per weight symbol for storage efficiency.

## 6.2 Activation Quantization

For activations $x$:

$$q = \mathrm{round}\left(x \cdot \frac{127}{\max(|x|) + \epsilon}\right), \qquad q \in [-128, 127]$$

with dequantization $x \approx q/\alpha$.

## 6.3 Size Estimates

For $N$ parameters:

$$\text{FP32 size} \approx 4N, \quad \text{FP16 size} \approx 2N, \quad \text{1.58-bit size} \approx \frac{1.58}{8}N$$

before metadata and packing overhead. This aligns with lightweight deployment goals [32, 5].

# 7 SBERT Downstream Tasks

Sentence embedding is built on Siamese-style training [31]. For sentence pair $(s_1, s_2)$ with embeddings $(e_1, e_2)$:

$$\cos(e_1, e_2) = \frac{e_1^\top e_2}{\|e_1\|\|e_2\|}$$

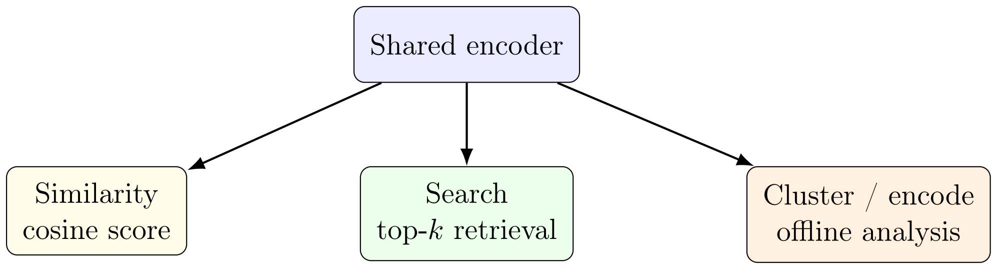

Figure 7: SBERT workflow reuse. A single encoder supports online pair scoring, corpus retrieval, clustering, and persistent embedding export.

**Algorithm 3** SBERT Inference Mode Router

```
Require: mode $m$, model $E$, inputs $X$
1: if $m =$ similarity then
2:     return $\cos(E(x_1), E(x_2))$
3: else if $m =$ search then
4:     return top-$k$ by dot-product/cosine against corpus embeddings
5: else if $m =$ cluster then
6:     return clustering labels over $E(X)$
7: else
8:     return serialized embeddings $E(X)$
9: end if
```

and regression-style cosine loss:

$$\mathcal{L}_{\text{cos}} = (\cos(e_1, e_2) - y)^2$$

with $y \in [-1, 1]$ in this pipeline.

Supported downstream modes:

- **Similarity**: pairwise score between two sentences.
- **Search**: top-$k$ nearest neighbors over a corpus.
- **Cluster**: grouping embeddings (e.g., k-means).
- **Encode**: persistent embedding export for later retrieval.

## 8 Summary Tables

### 8.1 Attention and Sequence-Mixer Summary

| Type | Core Equation | Train | Infer | Notes |
|---|---|---|---|---|
| Standard Attention | $\text{softmax}(QK^\top/\sqrt{d_k})V$ | $\mathcal{O}(n^2d)$ | $\mathcal{O}(n)$/step | Baseline expressive global routing [38]. |
| Sigmoid Attention | $\sigma(QK^\top/\sqrt{d_k}+b)V$ | $\mathcal{O}(n^2d)$ | $\mathcal{O}(n)$/step | Element-wise gating; often needs stabilization norm [30]. |
| RetNet | $(QK^\top \odot D)V$ | $\mathcal{O}(n^2d)$ or chunkwise | $\mathcal{O}(1)$/step | Parallel/recurrent dual form with decay retention [36]. |
| Mamba | $h_t = \bar{A}_t h_{t-1} + \bar{B}_t x_t$ | $\mathcal{O}(nd)$ | $\mathcal{O}(1)$/step | Selective state-space with hardware-aware scan [12]. |

| Type | Core Equation | Train | Infer | Notes |
|---|---|---|---|---|
| ODE-style block | $\frac{dh}{dt} = f_\theta(h, t)$ | solver-dependent | solver-dependent | Continuous-depth interpretation; RK integration [52]. |
| Titans memory | $M_t = (1-\alpha_t)M_{t-1} + S_t$ | approx. $\mathcal{O}(nd)$ | retrieval-centric | Test-time memory updates with surprise-driven dynamics [2]. |

## 8.2 Optimizer Summary

| Optimizer | Family | State Cost | Key Idea | Ref |
|---|---|---|---|---|
| SGD+Momentum | Classical first-order | Low | Momentum-accelerated baseline with minimal state | [27] |
| AdamW | Adaptive first/second moment | High | Decoupled weight decay baseline | [22] |
| RAdam | Adaptive variance-corrected | High | Rectifies early adaptive variance | [21] |
| Adan | Momentum + variance reduction | High | Nesterov-style adaptive update | [42] |
| ADOPT | Adam variant | High | Reordered updates with improved convergence guarantees | [37] |
| AdEMAMix | Multi-EMA adaptive | High | Mixes short and long horizon EMAs | [26] |
| MARS | Variance-reduced preconditioned | High | Recursive momentum correction | [49] |
| Cautious AdamW | Masked momentum | High | Apply updates only on sign-consistent directions | [18] |
| LAMB | Layer-wise adaptive moments | High | Trust-ratio scaling for very large-batch training | [47] |
| Schedule-free AdamW | Scheduler-free adaptive | High | Remove explicit LR schedule dependence | [10] |
| Adafactor | Memory-efficient adaptive | Medium | Factorized second moments for matrix tensors | [33] |
| GaLore AdamW | Low-rank gradient projection | Medium | Optimize in projected low-rank gradient space | [54] |
| APOLLO | Low-rank adaptive projection | Medium | Structured scaling from projected Adam-style moments | [55] |
| APOLLO-Mini | Rank-1 adaptive projection | Low | Tensor-wise scaled APOLLO variant for extreme memory savings | [55] |
| Q-APOLLO | Quantized low-rank projection | Low | Quantized APOLLO state for ultra-low-memory training | [55] |
| Prodigy | Parameter-free adaptation | Medium | Distance-adaptive step calibration | [24] |
| Lion | Sign momentum | Low | Momentum sign update, reduced state | [7] |
| Sophia | Approx. second-order | Medium | Diagonal Hessian preconditioning with clipping | [20] |
| Shampoo | Matrix preconditioner | High | Kronecker-structured second-order statistics | [14] |
| SOAP | Shampoo + Adam basis | High | Adam-like tracking in preconditioner eigenbasis | [39] |
| Muon | Orthogonality-based | Medium | Orthogonalized matrix updates | [34] |

| Optimizer | Family | State Cost | Key Idea | Ref |
|---|---|---|---|---|
| Turbo-Muon | Accelerated orthogonalization | Medium | Preconditioned Newton-Schulz speedup | [4] |

## 9 Discussion

The design choices in Transformer Encoder Frankenstein reflect several engineering and research tensions in modern deep learning tooling.

### 9.1 Schema-Driven Design Trade-offs

The schema-first approach provides significant reproducibility benefits by enforcing explicit contracts and failing fast on invalid configurations. However, this approach also introduces rigidity: adding new architectures or optimizers requires schema extensions rather than loose command-line arguments. The prefixed hyperparameter system enables fine-grained control but increases configuration complexity for users accustomed to simpler interfaces.

The decision to enforce `additionalProperties: false` at all schema levels eliminates silent parameter swallowing that has plagued earlier configuration systems, but this strictness requires careful schema maintenance when extending system capabilities. Each new attention mechanism or optimizer variant must be properly integrated into the validation framework, including schema field definitions with appropriate types and constraints, prefixed hyperparameter mapping for optimizer-specific groups, default values aligned with research best practices, and documentation strings for web interface rendering.

### 9.2 Architectural Coverage and Gaps

The seventeen implemented mixer architectures span major research directions in sequence modeling, but certain gaps remain. The system lacks recent hybrid architectures such as Griffin [35] and Jamba [13], which combine gating with state-space models. MoE (Mixture of Experts) routing is implemented for FFN layers but not for attention computation, where recent work has shown benefits [17].

The sparse attention coverage is comprehensive, but implementation of training-free methods (FASA, SpargeAttn) raises runtime errors during training, reflecting architectural constraints: these methods require pretrained checkpoints from full-attention models or specific fine-tuning procedures that are not currently automated.

The gated mechanism coverage is strong across major categories (GLA, DeltaNet, Gated DeltaNet, HGRN2, FoX, Gated Softmax).

### 9.3 Optimizer Landscape Fragmentation

The support for twenty-two optimizer families across six algorithmic categories demonstrates comprehensiveness but also highlights the fragmented state of optimization research. Users face significant decision complexity when choosing among variance-reduction methods (Adan, MARS), memory-efficient variants (GaLore, Adafactor, APOLLO), and curvature-aware approaches (Sophia, Shampoo). The prefixed hyperparameter system, while powerful, requires understanding of which parameters are relevant for each optimizer class.

The implementation quality varies across optimizers: classical methods (AdamW, SGD with momentum) are highly optimized in PyTorch, while newer methods (Muon, Turbo-Muon, SOAP) may require custom implementations that affect numerical stability and performance characteristics.

### 9.4 Deployment and Production Considerations

The quantization pipeline demonstrates practical deployment concerns but makes specific engineering trade-offs. Ternary weight packing reduces storage to approximately 1.58 bits per parameter, but this aggressive compression may degrade performance, especially for smaller models where quantization error is more significant. The current implementation applies quantization uniformly across parameter types.

The SBERT workflows provide practical utility for semantic similarity and retrieval tasks, but implementation assumes standard pooling strategies (CLS token, mean pooling). Recent advances such as Matryoshka embeddings [25] and contrastive learning refinements [16] are not yet incorporated.

### 9.5 Integration and Extensibility Challenges

The current codebase structure, while functional, presents maintenance challenges as architecture and optimizer families expand. The dispatcher pattern for mixer selection and optimizer routing handles extensibility but risks becoming a "kitchen sink" of conditional logic. Future versions would benefit from plugin-based architectures where new mixers and optimizers could be registered declaratively rather than modifying core dispatch logic.

The web configuration interface provides significant usability improvements but introduces deployment complexity: running Streamlit alongside training jobs requires additional resources and infrastructure considerations that may not be appropriate for all environments, particularly HPC clusters without web access.

## 10 Conclusion

Transformer Encoder Frankenstein presents a unified, configuration-driven experimentation platform addressing critical challenges in modern deep learning research: architectural fragmentation across dense attention, recurrent models, sparse patterns, and gated mechanisms; optimizer landscape complexity spanning classical baselines, variance-reduction methods, memory-efficient variants, schedule-free approaches, curvature-aware algorithms, and geometry-oriented methods; and end-to-end deployment workflows spanning quantization and sentence embedding applications.

The system's primary contributions are:

1. **Schema-First Design**: A strict YAML-based configuration contract with validation and prefixed hyperparameter routing enabling reproducible experiments across seventeen mixer architectures and twenty-two optimizer families.

2. **Comprehensive Architecture Support**: Implementation spanning major research categories including dense baselines (standard, sigmoid attention), recurrent alternatives (RetNet, Mamba, ODE-style, Titans), sparse attention (Sparse Transformer, Longformer, BigBird, SparseK, NSA, SpargeAttn, FASA), and gated mechanisms (GLA, DeltaNet, Gated DeltaNet, HGRN2, FoX, Gated Softmax).

3. **Unified Optimizer Framework**: Prefixed hyperparameter groups enabling fine-grained control over embeddings, normalization layers, recurrent blocks, attention weights, and FFN parameters across classical baselines (SGD+Momentum, AdamW), variance-reduction (Adan, ADOPT, AdEMAMix, MARS, Cautious), memory-efficient (Adafactor, GaLore, Lion, APOLLO, APOLLO-Mini, Q-APOLLO), large-batch and schedule simplification (LAMB, Schedule-Free AdamW, Prodigy), curvature-aware (Sophia), second-order (Shampoo, SOAP), and geometry-oriented (Muon, Turbo-Muon) optimizers.

4. **End-to-End Workflows**: Integrated deployment pipeline supporting ternary weight packing and INT8 activation quantization; SBERT-inspired training and inference for semantic similarity, retrieval, and clustering tasks.

5. **Interactive Configuration**: Streamlit-based web interface providing schema-driven form generation, real-time validation, inline documentation, and CLI command synthesis.

This system enables rapid experimental iteration while maintaining reproducibility through strict configuration contracts. By consolidating diverse research contributions into a unified toolkit, it lowers barriers to exploring novel architectures and optimization strategies, particularly for researchers who may lack resources to implement and validate each variant independently.

## 10.1 Limitations and Future Directions

Several limitations and promising directions for future work emerge from this system's design and implementation:

1. **Architecture Integration**: Recent hybrid architectures (Griffin, Jamba, Mamba-X) demonstrate benefits of combining multiple mechanisms into unified blocks. Future versions should integrate these architectures and explore systematic composition patterns.

2. **Advanced Quantization**: Current implementation uses uniform ternary packing across all parameters. Research on layer-wise, channel-wise, and importance-aware quantization suggests more sophisticated strategies could improve quality-efficiency trade-offs.

3. **Plugin-Based Extensibility**: The current dispatch pattern becomes increasingly complex with each new addition. A plugin architecture allowing declarative registration of new mixers, optimizers, and normalization methods would improve maintainability and reduce risk of bugs in core dispatch logic.

4. **Automated Hyperparameter Optimization**: The schema supports extensive hyperparameter spaces, but users must manually explore these spaces. Integration with Bayesian optimization, multi-armed bandit strategies, or gradient-based hyperparameter tuning could automate effective configuration discovery.

5. **Production Deployment**: The web interface improves usability but may not be appropriate for all deployment environments. Headless configuration modes, API-based configuration management, or improved CLI ergonomics could serve HPC and production workflows.

6. **Evaluation Benchmarking**: While the system enables training with diverse architectures, comprehensive benchmarking comparing performance across mixers and optimizers on standardized tasks would provide valuable guidance for configuration selection.

7. **Training Stability Guarantees**: Current implementation includes NaN/Inf guards and gradient clipping, but formal analysis of stability conditions for different mixer-optimizer combinations, particularly with looped blocks and aggressive quantization, remains open.

8. **Multimodal and Task-Specific Extensions**: The current design focuses on sequence modeling. Extensions for vision-language models, multimodal architectures, and task-specific fine-tuning workflows (e.g., instruction tuning, RLHF) would broaden applicability.

The research trajectory of sequence modeling continues toward hybrid approaches that combine strengths of multiple paradigms—compression from recurrence, selectivity from attention, gating for memory management, and sparsity for efficiency. A unified experimentation platform like Transformer Encoder Frankenstein is increasingly valuable as this convergence accelerates, enabling researchers to systematically explore this expanding design space with reproducible, well-engineered infrastructure.

## Bibliography

[1] Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time, . URL http://arxiv.org/abs/2501.00663.

[2] Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time, . URL https://arxiv.org/abs/2501.00663. Version Number: 1.

[3] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020. URL https://arxiv.org/abs/2004.05150.

[4] Thibaut Boissin, Thomas Massena, Franck Mamalet, and Mathieu Serrurier. Turbo-muon: Accelerating orthogonality-based optimization with pre-conditioning. URL https://arxiv.org/abs/2512.04632. Version Number: 1.

[5] Riccardo Bravin, Massimo Pavan, Hazem Hesham Yousef Shalby, Fabrizio Pittorino, and Manuel Roveri. EmbBERT: Attention under 2 MB memory. URL http://arxiv.org/abs/2502.10001.

[6] Mingzhi Chen, Taiming Lu, Jiachen Zhu, Mingjie Sun, and Zhuang Liu. Stronger normalization-free transformers, . URL http://arxiv.org/abs/2512.10938.

[7] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. Symbolic discovery of optimization algorithms, . URL https://arxiv.org/abs/2302.06675. Version Number: 4.

[8] Xin Cheng, Wangding Zeng, Damai Dai, Qinyu Chen, Bingxuan Wang, Zhenda Xie, Kezhao Huang, Xingkai Yu, Zhewen Hao, Yukun Li, Han Zhang, Huishuai Zhang, Dongyan Zhao, and Wenfeng Liang. Conditional memory via scalable lookup: A new axis of sparsity for large language models, 2026. URL https://arxiv.org/abs/2601.07372.

[9] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019. URL https://arxiv.org/abs/1904.10509.

[10] Aaron Defazio, Xingyu Alice Yang, Harsh Mehta, Konstantin Mishchenko, Ahmed Khaled, and Ashok Cutkosky. The road less scheduled. URL https://arxiv.org/abs/2405.15682. Version Number: 4.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. URL http://arxiv.org/abs/1810.04805.

[12] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. URL https://arxiv.org/abs/2312.00752. Version Number: 2.

[13] Albert Gu, AI21 Labs, et al. Jamba: A hybrid transformer-mamba language model, 2024. URL https://arxiv.org/abs/2403.19887.

[14] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. URL https://arxiv.org/abs/1802.09568. Version Number: 2.

[15] Sukjun Hwang, Aakash Lahoti, Tri Dao, and Albert Gu. Hydra: Bidirectional state space models through generalized matrix mixers. URL http://arxiv.org/abs/2407.09941.

[16] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning, 2020. URL https://arxiv.org/abs/2004.11362.

[17] Mike Lewis, Shruti Bhosale, Tim Dettmers, Douwe Kiela, and Luke Zettlemoyer. Base layers: Simplifying training of large, sparse models, 2021. URL https://arxiv.org/abs/2103.16716.

[18] Kaizhao Liang, Lizhang Chen, Bo Liu, and Qiang Liu. Cautious optimizers: Improving training with one line of code. URL https://arxiv.org/abs/2411.16085. Version Number: 4.

[19] Zhixuan Lin, Ke Wang, et al. Forgetting transformer: Softmax attention with a forget gate, 2025. URL https://arxiv.org/abs/2503.02130.

[20] Hong Liu, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. Sophia: A scalable stochastic second-order optimizer for language model pre-training, . URL https://arxiv.org/abs/2305.14342. Version Number: 4.

[21] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond, . URL https://arxiv.org/abs/1908.03265. Version Number: 4.

[22] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. URL https://arxiv.org/abs/1711.05101. Version Number: 3.

[23] Tianyu Lou, Zheyu Chen, Tao Yu, et al. Efficient sparse attention for long-range transformers, 2024. URL https://arxiv.org/abs/2406.16747.

[24] Konstantin Mishchenko and Aaron Defazio. Prodigy: An expeditiously adaptive parameter-free learner. URL https://arxiv.org/abs/2306.06101. Version Number: 4.

[25] Niklas Muennighoff et al. Matryoshka representation learning, 2022. URL https://arxiv.org/abs/2205.13147.

[26] Matteo Pagliardini, Pierre Ablin, and David Grangier. The AdEMAMix optimizer: Better, faster, older. URL https://arxiv.org/abs/2409.03137. Version Number: 2.

[27] Boris T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964. doi: 10.1016/0041-5553(64)90137-5.

[28] Zhen Qin, Xu Han, et al. Hgrn2: Gated linear rnns with state expansion, 2024. URL https://arxiv.org/abs/2404.07904.

[29] Yuxiang Qiu, Qwen Team, et al. Gated attention for large language models, 2025. URL https://arxiv.org/abs/2505.06708.

[30] Jason Ramapuram, Federico Danieli, Eeshan Dhekane, Floris Weers, Dan Busbridge, Pierre Ablin, Tatiana Likhomanenko, Jagrit Digani, Zijin Gu, Amitis Shidani, and Russ Webb. Theory, analysis, and best practices for sigmoid self-attention. URL https://arxiv.org/abs/2409.04431. Version Number: 2.

[31] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. URL http://arxiv.org/abs/1908.10084.

[32] Hema Hariharan Samson. Lightweight transformer architectures for edge devices in real-time applications. URL http://arxiv.org/abs/2601.03290.

[33] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. URL https://arxiv.org/abs/1804.04235. Version Number: 1.

[34] Wei Shen, Ruichuan Huang, Minhui Huang, Cong Shen, and Jiawei Zhang. On the convergence analysis of muon. URL https://arxiv.org/abs/2505.23737. Version Number: 1.

[35] Rafael Soares et al. Griffin: Mixing gated linear recurrences with local attention for efficient sequence modeling, 2024. URL https://arxiv.org/abs/2402.19427.

[36] Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models. URL https://arxiv.org/abs/2307.08621. Version Number: 4.

[37] Shohei Taniguchi, Keno Harada, Gouki Minegishi, Yuta Oshima, Seong Cheol Jeong, Go Nagahara, Tomoshi Iiyama, Masahiro Suzuki, Yusuke Iwasawa, and Yutaka Matsuo. ADOPT: Modified adam can converge with any $\beta_2$ with the optimal rate. URL https://arxiv.org/abs/2411.02853. Version Number: 3.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. URL https://arxiv.org/abs/1706.03762. Version Number: 7.

[39] Nikhil Vyas, Depen Morwani, Rosie Zhao, Mujin Kwun, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham Kakade. SOAP: Improving and stabilizing shampoo using adam. URL https://arxiv.org/abs/2409.11321. Version Number: 2.

[40] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. BitNet: Scaling 1-bit transformers for large language models. URL http://arxiv.org/abs/2310.11453.

[41] Zhe Wang, Ming Liu, et al. Fasa: Frequency-aware sparse attention, 2026. URL https://arxiv.org/abs/2602.03152.

[42] Xingyu Xie, Pan Zhou, Huan Li, Zhouchen Lin, and Shuicheng Yan. Adan: Adaptive nesterov momentum algorithm for faster optimizing deep models. URL https://arxiv.org/abs/2208.06677. Version Number: 5.

[43] Haiqi Yang, Zhiyuan Li, Yi Chang, and Yuan Wu. A survey of retentive network. URL http://arxiv.org/abs/2506.06708.

[44] Songlin Yang, Bailin Wang, et al. Gated linear attention transformers with hardware-efficient training, 2023. URL https://arxiv.org/abs/2312.06635.

[45] Songlin Yang, Bailin Wang, et al. Parallelizing linear transformers with the delta rule over sequence length, 2024. URL https://arxiv.org/abs/2406.06484.

[46] Songlin Yang, Bailin Wang, et al. Gated delta networks: Improving mamba2 with delta rule, 2024. URL https://arxiv.org/abs/2412.06464.

[47] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training BERT in 76 minutes. URL https://arxiv.org/abs/1904.00962. Version Number: 3.

[48] Han Yuan, DeepSeek-AI, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention, 2025. URL https://arxiv.org/abs/2502.11089.

[49] Huizhuo Yuan, Yifeng Liu, Shuang Wu, Xun Zhou, and Quanquan Gu. MARS: Unleashing the power of variance reduction for training large models. URL https://arxiv.org/abs/2411.10438. Version Number: 4.

[50] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Pike Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems*, 2020. doi: 10.48550/ARXIV.2007.14062. URL https://arxiv.org/abs/2007.14062.

[51] Biao Zhang and Rico Sennrich. Root Mean Square Layer Normalization. URL http://arxiv.org/abs/1910.07467.

[52] Jing Zhang, Peng Zhang, Baiwen Kong, Junqiu Wei, and Xin Jiang. Continuous self-attention models with neural ODE networks. 35(16):14393–14401. ISSN 2374-3468, 2159-5399. doi: 10.1609/aaai.v35i16.17692. URL https://ojs.aaai.org/index.php/AAAI/article/view/17692.

[53] Yichi Zhang, Yizhong Wang, et al. Accurate and training-free sparse attention accelerating any model inference, 2025. URL https://arxiv.org/abs/2502.18137.

[54] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. GaLore: Memory-efficient LLM training by gradient low-rank projection. URL https://arxiv.org/abs/2403.03507. Version Number: 2.

[55] Hanqing Zhu, Zhenyu Zhang, Wenyan Cong, Xi Liu, Sem Park, Vikas Chandra, Bo Long, David Z. Pan, Zhangyang Wang, and Jinwon Lee. Apollo: Sgd-like memory, adamw-level performance, 2025. URL https://arxiv.org/abs/2412.05270.

[56] Jiachen Zhu, Xinlei Chen, Kaiming He, Yann LeCun, and Zhuang Liu. Transformers without normalization. URL http://arxiv.org/abs/2503.10622.

[57] Lianghui Zhu, Yuxin Fang, Bencheng Liao, Shijie Wang, Tianheng Cheng, Zilong Huang, Chen Chen, Lai Wei, Yutao Zeng, Ya Wang, Yi Lin, Yu Li, and Xinggang Wang. Mixture-of-depths attention, 2026. URL https://arxiv.org/abs/2603.15619.

## A Annex A: Optimizer Families

### A.1 Memory and Complexity Comparison

Table 8 summarizes the memory overhead (number of state buffers per parameter), per-step computational complexity, and key hyperparameters for all supported optimizers. Memory overhead is expressed in terms of the number of state tensors maintained per model parameter, where each tensor has the same shape as the parameter.

| **Optimizer** | **State Buffers** | **Per-Step Cost** | **Key Hyperparams** |
|---|---|---|---|
| SGD+Momentum | 1 ($m$) | $\mathcal{O}(n)$ | lr, momentum, wd |
| AdamW | 2 ($m$, $v$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, eps, wd |
| RAdam | 2 ($m$, $v$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, eps, wd |

*contin*

| **Optimizer** | **State Buffers** | **Per-Step Cost** | **Key Hyperparams** |
|---|---|---|---|
| Adan | 3 ($m$, $v$, $s$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, $\beta_3$, eps, wd |
| ADOPT | 2 ($m$, $v$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, eps, wd |
| AdEMAMix | 3 ($m_1$, $m_2$, $v$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, $\beta_3$, eps, wd |
| MARS | 3 ($m$, $v$, $z$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, eps, wd, $\gamma$ |
| Cautious AdamW | 2 ($m$, $v$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, eps, wd |
| LAMB | 2 ($m$, $v$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, eps, wd |
| Schedule-Free | 3 ($z$, $x$, $n$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, wd |
| Adafactor | 1–2 (row/col) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, eps, wd, factored |
| GaLore | 2 ($m$, $v$) + SVD/proj | $\mathcal{O}(nr)$ | lr, rank $r$, $\beta_1$, $\beta_2$, eps, wd |
| Prodigy | 3 ($m$, $v$, $d$) | $\mathcal{O}(n)$ | $\beta_1$, $\beta_2$, eps, wd, $d_0$ |
| Lion | 1 ($m$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, wd |
| Shampoo | 2$d$ ($L_i$, $R_i$) | $\mathcal{O}(n^{1+2/d})$ | lr, eps, wd, matrix eps |
| SOAP | 2$d$ + 2 ($m$, $v$) | $\mathcal{O}(n^{1+1/d})$ | lr, $\beta_1$, $\beta_2$, eps, wd, shampoo eps |
| Sophia | 3 ($m$, $v$, $h$) | $\mathcal{O}(n)$ | lr, $\beta_1$, $\beta_2$, eps, wd, $k$ |
| Muon | 2 ($m$, $v$) | $\mathcal{O}(n \cdot k)$ | lr, momentum, wd, NS steps $k$ |
| Turbo-Muon | 2 ($m$, $v$) | $\mathcal{O}(n \cdot k)$ | lr, momentum, wd, NS steps $k$ |
| APOLLO | 2 low-rank ($m_R$, $v_R$) + proj | $\mathcal{O}(nr)$ | lr, rank $r$, update gap, scale, betas, eps |
| APOLLO-Mini | 2 rank-1 ($m_R$, $v_R$) + proj | $\mathcal{O}(n)$ | lr, update gap, scale, betas, eps, wd |
| Q-APOLLO | 2 quantized low-rank + proj | $\mathcal{O}(nr)$ | lr, rank $r$, update gap, scale, quant bits |

Table 8: Memory and complexity comparison of all supported optimizers. $n$ = number of parameters, $d$ = tensor dimensionality, $r$ = low-rank dimension, $k$ = Newton–Schulz iteration count. State buffers lists the number of optimizer state tensors maintained per parameter group. Per-step cost focuses on the dominant additional cost beyond the gradient computation itself.

## A.2 The Evolution of Optimization in Neural Networks

The optimizer survey frames transformer optimization as a response to three structural pressures: non-convex loss landscapes, severe curvature heterogeneity across parameter blocks, and the memory cost of storing optimizer state for very large models. The report argues that the field has diverged into several trajectories: adaptive first-order baselines, variance-reduction methods, memory-efficient methods, structured second-order preconditioners, schedule-free methods, and orthogonality-oriented updates.

## A.3 Standard Baseline and Adaptive Optimizers

**SGD with Momentum.** The classical update accumulates a momentum buffer and then applies a fixed learning rate. At each iteration $t$, the algorithm maintains an exponential moving average $m_t$ of past gradients:

$$m_t = \beta m_{t-1} + g_t \tag{1}$$

$$\theta_{t+1} = \theta_t - \eta m_t \tag{2}$$

where $g_t = \nabla f(\theta_t)$, $\beta \in [0.8, 0.99]$ controls the momentum decay, and $\eta$ is the fixed learning rate. The momentum term acts as velocity: it accelerates movement in consistent directions while dampening oscillations in variable directions. Its strengths are low memory overhead (single momentum buffer per parameter) and strong generalization when tuned carefully. Its main weakness in transformer workloads is poor robustness to heterogeneous curvature (different Hessian spectra across parameter groups) and strong dependence on learning-rate schedules.

```
Algorithm 4 SGD with Momentum
Require: Initial parameters $\theta_0$, learning rate $\eta$, momentum coefficient $\beta$, weight decay $\lambda$
Ensure: Updated parameters $\theta_T$
1: Initialize: momentum buffer $m \leftarrow 0$
2: for $t = 0, 1, 2, \ldots, T-1$ do
3:     Compute gradient: $g_t \leftarrow \nabla f(\theta_t)$
4:     if weight_decay > 0 then
5:         $g_t \leftarrow g_t + \lambda\theta_t$                    ▷ L2 regularization
6:     end if
7:     Update momentum: $m \leftarrow \beta \cdot m + g_t$
8:     Update parameters: $\theta_{t+1} \leftarrow \theta_t - \eta \cdot m$
9: end forreturn $\theta_T$
```

**Adam and AdamW.** Adam tracks exponential moving averages of both first moment (mean) and second moment (uncentered variance) to achieve element-wise adaptive learning rates. The update rule is:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{3}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{4}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \quad \text{(bias correction)} \tag{5}$$

$$\theta_{t+1} = \theta_t - \eta \left( \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \right) \tag{6}$$

AdamW introduces a crucial modification: weight decay is applied directly to parameters (decoupled from gradients): $\theta_t \leftarrow \theta_t(1-\eta\lambda)$ rather than adding $\lambda\theta_t$ to the gradient. This decoupling prevents the adaptive scaling from interfering with regularization strength. The report treats AdamW as the practical baseline for transformer fine-tuning because it converges quickly and is relatively forgiving to hyperparameter variation. The tradeoff is memory cost: both moment tensors must be stored for every parameter, doubling optimizer-state memory relative to SGD.

```
Algorithm 5 AdamW (Adam with Decoupled Weight Decay)
Require: Initial parameters $\theta_0$, learning rate $\eta$, exponential decay rates $\beta_1, \beta_2 \in [0, 1)$
Require: Momentum constant $\epsilon$, weight decay $\lambda$
Ensure: Updated parameters $\theta_T$
1: Initialize: first moment $m \leftarrow 0$, second moment $v \leftarrow 0$, step counter $t \leftarrow 0$
2: for $t = 1, 2, \ldots, T$ do
3:     Compute gradient: $g_t \leftarrow \nabla f(\theta_{t-1})$
4:     Weight decay (decoupled): $\theta_{t-1} \leftarrow \theta_{t-1}(1 - \eta\lambda)$
5:     Update moments: $m \leftarrow \beta_1 m + (1-\beta_1) g_t$
6:     $v \leftarrow \beta_2 v + (1-\beta_2) g_t^2$
7:     Bias correction: $\hat{m} \leftarrow m/(1-\beta_1^t)$, $\hat{v} \leftarrow v/(1-\beta_2^t)$
8:     Update parameters: $\theta_t \leftarrow \theta_{t-1} - \eta(\hat{m}/(\sqrt{\hat{v}} + \epsilon))$
9: end forreturn $\theta_T$
```

**RAdam (Rectified Adam).** RAdam addresses Adam's instability in early training by dynamically rectifying the adaptive learning rate. The key observation is that the second moment

$v_t$ has very high variance in the first few steps, causing unreliable adaptive scaling. RAdam computes the effective simple moving average (SMA) window length:

$$\rho_t = \rho_\infty - \frac{2t\beta_2^t}{1-\beta_2^t}, \quad \text{where} \quad \rho_\infty = \frac{2}{1-\beta_2} - 1$$

When $\rho_t > 4$ (sufficient samples for variance estimation), RAdam applies adaptive scaling with a rectification term:

$$r_t = \sqrt{\frac{(\rho_t - 4)(\rho_t - 2)\rho_\infty}{(\rho_\infty - 4)(\rho_\infty - 2)\rho_t}}, \quad \theta_{t+1} = \theta_t - \eta r_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

When $\rho_t \leq 4$, RAdam falls back to SGD with momentum: $\theta_{t+1} = \theta_t - \eta \hat{m}_t$. This graceful transition eliminates the need for manual learning-rate warmup schedules and improves robustness.

**Algorithm 6** RAdam (Rectified Adam)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, $\beta_1, \beta_2$, weight decay $\lambda$
Ensure: Updated parameters $\theta_T$
 1: Initialize: $m \leftarrow 0$, $v \leftarrow 0$, $t \leftarrow 0$
 2: Compute: $\rho_\infty \leftarrow 2/(1-\beta_2) - 1$
 3: for $t = 1, 2, \ldots, T$ do
 4:     $g_t \leftarrow \nabla f(\theta_{t-1})$
 5:     if weight_decay $> 0$ then
 6:         $\theta_{t-1} \leftarrow \theta_{t-1}(1 - \eta\lambda)$
 7:     end if
 8:     $m \leftarrow \beta_1 m + (1-\beta_1) g_t$
 9:     $v \leftarrow \beta_2 v + (1-\beta_2) g_t^2$
10:     $\hat{m} \leftarrow m/(1-\beta_1^t)$, $\hat{v} \leftarrow v/(1-\beta_2^t)$
11:     $\rho_t \leftarrow \rho_\infty - 2t\beta_2^t/(1-\beta_2^t)$
12:     if $\rho_t > 4$ then
13:         $r_t \leftarrow \sqrt{((\rho_t - 4)(\rho_t - 2)\rho_\infty)/((\rho_\infty - 4)(\rho_\infty - 2)\rho_t)}$
14:         $\theta_t \leftarrow \theta_{t-1} - \eta r_t \hat{m}/(\sqrt{\hat{v}} + \epsilon)$
15:     else
16:         $\theta_t \leftarrow \theta_{t-1} - \eta \hat{m}$                ▷ SGD with momentum
17:     end if
18: end forreturn $\theta_T$
```

## A.4 Advanced Momentum and Variance Reduction (2024–2025)

**Adan (Adaptive Nesterov Momentum).** Adan reformulates Nesterov acceleration without the extra gradient computation required by classical Nesterov SGD. The algorithm maintains three momentum buffers:

$$m_t = (1-\beta_1)m_{t-1} + \beta_1 g_t \quad \text{(first moment)} \tag{7}$$

$$v_t = (1-\beta_2)v_{t-1} + \beta_2(g_t - g_{t-1}) \quad \text{(velocity/gradient difference)} \tag{8}$$

$$n_t = (1-\beta_3)n_{t-1} + \beta_3[g_t + (1-\beta_1)(g_t - g_{t-1})]^2 \quad \text{(Nesterov second moment)} \tag{9}$$

The **Nesterov Momentum Estimation** (NME) term $\bar{g}_t = g_t + (1-\beta_1)(g_t - g_{t-1})$ estimates the gradient at a future position without evaluating it. The update combines momentum with velocity:

$$\bar{m}_t = m_t + (1-\beta_1)v_t, \quad \theta_{t+1} = \theta_t - \eta \frac{\bar{m}_t}{\sqrt{n_t} + \epsilon}$$

Adan achieves fast convergence across diverse architectures (CNNs, GANs, Transformers) through this acceleration. The cost is maintaining three momentum-like buffers, increasing memory overhead relative to Adam.

**Algorithm 7** Adan (Adaptive Nesterov Momentum)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, $\beta_1, \beta_2, \beta_3 \in (0, 1)$
Ensure: Updated parameters $\theta_T$
 1: Initialize: $m \leftarrow 0$, $v \leftarrow 0$, $n \leftarrow 0$, $g_{-1} \leftarrow 0$ (previous gradient)
 2: for $t = 1, 2, \ldots, T$ do
 3:     $g_t \leftarrow \nabla f(\theta_{t-1})$
 4:     $m \leftarrow (1 - \beta_1)m + \beta_1 g_t$
 5:     $\Delta g_t \leftarrow g_t - g_{t-1}$
 6:     $v \leftarrow (1 - \beta_2)v + \beta_2 \Delta g_t$
 7:     Nesterov estimation: $\bar{g}_t \leftarrow g_t + (1 - \beta_1)\Delta g_t$
 8:     $n \leftarrow (1 - \beta_3)n + \beta_3 \bar{g}_t^2$
 9:     Combined momentum: $\bar{m} \leftarrow m + (1 - \beta_1)v$
10:     $\theta_t \leftarrow \theta_{t-1} - \eta(\bar{m}/(\sqrt{n} + \epsilon))$
11:     $g_{t-1} \leftarrow g_t$
12: end forreturn $\theta_T$
```

**ADOPT (Adam with Optimal Pruned Tuning).** ADOPT fixes a fundamental theoretical issue in Adam: the gradient appears in both the first moment and second moment estimates, creating circularity. ADOPT **decouples** by using the previous step's second moment for the denominator:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{10}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{g_t}{\sqrt{v_{t-1}} + \epsilon} \quad \text{(uses } v_{t-1}\text{, not } v_t\text{)} \tag{11}$$

$$\theta_{t+1} = \theta_t - \eta m_t \tag{12}$$

This simple reordering achieves the optimal convergence rate $O(1/\sqrt{T})$ with **any** choice of $\beta_2 \in (0, 1)$, without bounded-noise assumptions. The practical consequence is that ADOPT is a drop-in replacement for Adam with stronger theoretical guarantees and comparable or superior empirical performance across vision, NLP, RL, and generative modeling domains.

**Algorithm 8** ADOPT (Adam with Optimal Pruned Tuning)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, $\beta_1, \beta_2$, tolerance $\epsilon$
Ensure: Updated parameters $\theta_T$
 1: Initialize: $m \leftarrow 0$, $v \leftarrow 0$
 2: for $t = 1, 2, \ldots, T$ do
 3:     $g_t \leftarrow \nabla f(\theta_{t-1})$
 4:     if weight_decay $> 0$ then
 5:         $\theta_{t-1} \leftarrow \theta_{t-1}(1 - \eta\lambda)$
 6:     end if
 7:     Use previous second moment: denom $\leftarrow \sqrt{\max(v, \epsilon)} + \epsilon$
 8:     Update first moment using previous variance: $m \leftarrow \beta_1 m + (1 - \beta_1)(g_t/\text{denom})$
 9:     Update second moment with current gradient: $v \leftarrow \beta_2 v + (1 - \beta_2) g_t^2$
10:     $\theta_t \leftarrow \theta_{t-1} - \eta m$
11: end forreturn $\theta_T$
```

**AdEMAMix (Exponential Moving Average Mixture).** AdEMAMix replaces Adam's single EMA of gradients with a **mixture of two EMAs**: one fast-decaying and one slow-decaying. This addresses the observation that gradients remain informative over tens of thousands of steps, not just hundreds:

$$m_{1,t} = \beta_1 m_{1,t-1} + (1-\beta_1)g_t \quad \text{(fast EMA, } \beta_1 \approx 0.9\text{)} \tag{13}$$

$$m_{2,t} = \beta_3 m_{2,t-1} + (1-\beta_3)g_t \quad \text{(slow EMA, } \beta_3 \approx 0.9999\text{)} \tag{14}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \tag{15}$$

$$m_t = m_{1,t} + \alpha_t m_{2,t} \quad \text{(mixture with scheduled weight } \alpha_t\text{)} \tag{16}$$

$$\theta_{t+1} = \theta_t - \eta \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{17}$$

The mixture weight $\alpha_t$ typically increases during training, allowing the fast EMA to provide immediate adaptation while the slow EMA accumulates long-range gradient correlations. Empirically, a 1.3B model on 101B tokens achieves similar loss to AdamW on 197B tokens (95% data efficiency gain), suggesting the slow EMA significantly reduces model forgetting.

**Algorithm 9** AdEMAMix (Exponential Moving Average Mixture)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, $\beta_1$ (fast), $\beta_3$ (slow), $\beta_2$ (second moment)
Require: Mixture weight $\alpha_t$ (typically increasing), tolerance $\epsilon$
Ensure: Updated parameters $\theta_T$
 1: Initialize: $m_1 \leftarrow 0$ (fast EMA), $m_2 \leftarrow 0$ (slow EMA), $v \leftarrow 0$
 2: for $t = 1, 2, \ldots, T$ do
 3:     $g_t \leftarrow \nabla f(\theta_{t-1})$
 4:     if weight_decay $> 0$ then
 5:         $\theta_{t-1} \leftarrow \theta_{t-1}(1 - \eta\lambda)$
 6:     end if
 7:     $m_1 \leftarrow \beta_1 m_1 + (1-\beta_1)g_t$                ▷ Fast EMA
 8:     $m_2 \leftarrow \beta_3 m_2 + (1-\beta_3)g_t$                ▷ Slow EMA
 9:     $v \leftarrow \beta_2 v + (1-\beta_2)g_t^2$
10:     $m_t \leftarrow m_1 + \alpha_t m_2$                          ▷ Mixture
11:     $\theta_t \leftarrow \theta_{t-1} - \eta(m_t/(\sqrt{v} + \epsilon))$
12: end forreturn $\theta_T$
```

**MARS (Make Adaptive learning Rates Shine).** MARS combines preconditioned gradient methods (e.g., AdamW) with **variance reduction** via scaled stochastic recursive momentum. The core innovation is a variance-reduced gradient estimate:

$$c_t = g_t + \gamma(c_{t-1} - g_{t-1}) \quad \text{(SVRG-style recursive estimate)} \tag{18}$$

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)c_t \tag{19}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)c_t^2 \tag{20}$$

$$\theta_{t+1} = \theta_t - \eta \frac{m_t}{\sqrt{v_t} + \epsilon} \tag{21}$$

where $\gamma \in [0.01, 0.1]$ controls how much historical gradient information is retained. The variance-reduced gradient $c_t$ acts as an implicit noise filter, amplifying consistent signal directions and dampening contradictory noise. MARS-AdamW consistently outperforms bare AdamW by significant margins on GPT-2 pretraining, suggesting that variance reduction substantially improves convergence in mini-batch stochastic training.

**Algorithm 10** MARS (Make Adaptive learning Rates Shine)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, $\beta_1, \beta_2$, weight decay $\lambda$
Require: Variance reduction coefficient $\gamma \in [0.01, 0.1]$
Ensure: Updated parameters $\theta_T$
 1: Initialize: $m \leftarrow 0$, $v \leftarrow 0$, $c \leftarrow 0$ (variance-reduced gradient), $g_{-1} \leftarrow 0$
 2: for $t = 1, 2, \ldots, T$ do
 3:     $g_t \leftarrow \nabla f(\theta_{t-1})$
 4:     Variance reduction: $c \leftarrow g_t + \gamma(c - g_{t-1})$
 5:     $m \leftarrow \beta_1 m + (1 - \beta_1)c$
 6:     $v \leftarrow \beta_2 v + (1 - \beta_2)c^2$
 7:     if weight_decay > 0 then
 8:         $\theta_{t-1} \leftarrow \theta_{t-1}(1 - \eta\lambda)$
 9:     end if
10:     $\theta_t \leftarrow \theta_{t-1} - \eta(m/(\sqrt{v} + \epsilon))$
11:     $g_{t-1} \leftarrow g_t$
12: end forreturn $\theta_T$
```

**Cautious Optimizers (Cautious AdamW, Cautious Lion).** The Cautious framework applies a **one-line modification** to any momentum-based optimizer: mask the update so that only dimensions where momentum and gradient directions agree are applied:

$$\text{mask}_i = \begin{cases} 1 & \text{if } m_t[i] \cdot g_t[i] > 0 \quad \text{(agreement)} \\ 0 & \text{otherwise} \end{cases} \tag{22}$$

$$u_t = \left( \frac{m_t}{\sqrt{v_t} + \epsilon} \right) \odot \text{mask} \quad \text{(element-wise masking)} \tag{23}$$

$$\theta_{t+1} = \theta_t - \eta u_t \tag{24}$$

The intuition: momentum $m_t$ estimates the gradient direction from history; the current gradient $g_t$ is instantaneous signal. When both agree, the optimizer is confident and should update aggressively. When they disagree, the optimizer is conflicted—the historical trend points toward a region that may have been good before, but current evidence contradicts it. By masking conflicted dimensions, Cautious becomes more conservative and avoids corrupted steps. Empirically, this simple masking achieves up to **1.47× speedup** on Llama and MAE pretraining while preserving convergence guarantees.

**Algorithm 11** Cautious AdamW (Consensus-based Update Masking)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, $\beta_1, \beta_2$, weight decay $\lambda$
Ensure: Updated parameters $\theta_T$
 1: Initialize: $m \leftarrow 0$, $v \leftarrow 0$
 2: for $t = 1, 2, \ldots, T$ do
 3:     $g_t \leftarrow \nabla f(\theta_{t-1})$
 4:     $m \leftarrow \beta_1 m + (1 - \beta_1)g_t$
 5:     $v \leftarrow \beta_2 v + (1 - \beta_2)g_t^2$
 6:     Consensus mask: $\text{mask}_i \leftarrow 1$ if $m[i] \cdot g_t[i] > 0$, else 0        ▷ Agreement
 7:     Base Adam update: $u \leftarrow m/(\sqrt{v} + \epsilon)$
 8:     Apply mask: $u_{\text{masked}} \leftarrow u \odot \text{mask}$        ▷ Element-wise
 9:     if weight_decay > 0 then
10:         $\theta_{t-1} \leftarrow \theta_{t-1}(1 - \eta\lambda)$
11:     end if
12:     $\theta_t \leftarrow \theta_{t-1} - \eta u_{\text{masked}}$
13: end forreturn $\theta_T$
```

## A.5 Large-Batch, Memory-Efficient, and Parameter-Free Optimizers

**LAMB (Layer-wise Adaptive Moments optimizer for Batch training).** LAMB extends Adam with **layer-wise adaptive rate scaling** (inspired by LARS), enabling stable training with extreme batch sizes (e.g., 64K on BERT):

$$m_t^L = \beta_1 m_{t-1}^L + (1-\beta_1) g_t^L \quad \text{(layer-wise first moment)} \tag{25}$$

$$v_t^L = \beta_2 v_{t-1}^L + (1-\beta_2)(g_t^L)^2 \tag{26}$$

$$u_{\text{adam}}^L = \frac{m_t^L}{\sqrt{v_t^L}+\epsilon} + \lambda\theta_{t-1}^L \quad \text{(base adaptive step with decay)} \tag{27}$$

$$\phi^L = \frac{\|\theta_{t-1}^L\|_2}{\|u_{\text{adam}}^L\|_2} \quad \text{(trust ratio: layer normalization)} \tag{28}$$

$$\theta_t^L = \theta_{t-1}^L - \eta \cdot \phi^L \cdot u_{\text{adam}}^L \tag{29}$$

The **trust ratio** $\phi^L = \|\theta^L\|_2/\|u_{\text{adam}}^L\|_2$ normalizes the effective update step relative to weight magnitude. In very large batches, stochastic gradient noise can exceed signal, destabilizing layer-wise learning rates. By scaling updates proportionally to weight norms, LAMB ensures relative changes rather than absolute ones, preventing small confident updates in large vectors from dominating. LAMB preserves small-batch generalization benefits while enabling efficient large-batch training.

**Algorithm 12** LAMB (Layer-wise Adaptive Moments optimizer for Batch training)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, $\beta_1, \beta_2$, weight decay $\lambda$
Ensure: Updated parameters $\theta_T$
 1: Initialize: For each layer $L$: $m^L \leftarrow 0$, $v^L \leftarrow 0$
 2: for $t = 1, 2, \ldots, T$ do
 3:     for each layer $L$ do
 4:         $g_t^L \leftarrow \nabla f(\theta_t^L)$
 5:         $m^L \leftarrow \beta_1 m^L + (1-\beta_1) g_t^L$
 6:         $v^L \leftarrow \beta_2 v^L + (1-\beta_2)(g_t^L)^2$
 7:         Base Adam: $u_{\text{adam}}^L \leftarrow m^L/(\sqrt{v^L}+\epsilon)$
 8:         if weight_decay > 0 then
 9:             $u_{\text{adam}}^L \leftarrow u_{\text{adam}}^L + \lambda\theta_{t-1}^L$
10:         end if
11:         Trust ratio: $\phi^L \leftarrow \|\theta_{t-1}^L\|_2/\|u_{\text{adam}}^L\|_2$ if both nonzero, else 1
12:         $\theta_t^L \leftarrow \theta_{t-1}^L - \eta\phi^L u_{\text{adam}}^L$
13:     end for
14: end forreturn $\theta_T$
```

**Schedule-Free AdamW.** Schedule-free methods remove explicit scheduler design from the optimization recipe. The algorithm maintains two parameter streams: $z_t$ (exploration) and $x_t$ (smooth average):

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad \text{(variance)} \tag{30}$$

$$z_t = z_{t-1}(1-\eta\lambda) - \eta\frac{g_t}{\sqrt{v_t}+\epsilon} \quad \text{(adaptive step with decay)} \tag{31}$$

$$c_t = \frac{1}{t+1} \quad \text{(averaging coefficient, decreases over time)} \tag{32}$$

$$x_t = (1-c_t)x_{t-1} + c_t z_t \quad \text{(iterate averaging)} \tag{33}$$

$$y_t = (1-\beta)z_t + \beta x_t \quad \text{(interpolation for evaluation point)} \tag{34}$$

The averaging coefficient $c_t = 1/(t+1)$ implements an implicit learning-rate schedule without explicitly specifying total steps $T$. This framework unifies scheduling and iterate averaging: the algorithm explores via $z_t$ while accumulating stable direction via $x_t$. The evaluation point $y_t$ (used for gradient computation) interpolates between exploration and stability. Schedule-Free achieves state-of-the-art convergence across convex optimization, large-scale deep learning, and reinforcement learning, while removing a major hyperparameter.

**Algorithm 13** Schedule-Free AdamW

```
Require: Initial parameters $\theta_0$, learning rate $\eta$ (fixed), $\beta_1, \beta_2$, weight decay $\lambda$
Ensure: Updated parameters $\theta_T$ or averaging $x_T$
 1: Initialize: $z \leftarrow \theta_0$ (exploration), $x \leftarrow \theta_0$ (average), $v \leftarrow 0$, $t \leftarrow 0$
 2: for $t = 1, 2, \dots, T$ do
 3:     $g_t \leftarrow \nabla f(y_{t-1})$ (gradient at interpolation point)
 4:     $v \leftarrow \beta_2 v + (1-\beta_2) g_t^2$                    ▷ Variance
 5:     Weight decay on $z$: $z \leftarrow z(1-\eta\lambda)$
 6:     $z \leftarrow z - \eta g_t / (\sqrt{v} + \epsilon)$             ▷ Adaptive step on raw
 7:     $c_t \leftarrow 1/(t+1)$                                       ▷ Averaging coefficient
 8:     $x \leftarrow (1-c_t)x + c_t z$                                ▷ Iterate averaging
 9:     $y_t \leftarrow (1-\beta_1)z + \beta_1 x$                      ▷ Interpolation for next eval
10: end forreturn $x_T$ or $y_T$
```

**Adafactor.** Adafactor reduces optimizer memory by **factorizing** second-moment statistics for matrix-shaped parameters, storing only row and column accumulators rather than dense variance states. For a gradient matrix $G_t \in \mathbb{R}^{m \times n}$:

$$R_t = \beta_2 R_{t-1} + (1-\beta_2)(G_t^2)\mathbf{1}_n^T \quad \text{(row variance)} \tag{35}$$

$$C_t = \beta_2 C_{t-1} + (1-\beta_2)\mathbf{1}_m^\top (G_t^2) \quad \text{(column variance)} \tag{36}$$

$$\hat{V}_t = \frac{R_t C_t}{\mathbf{1}_n^\top R_t} \quad \text{(reconstructed variance via outer product)} \tag{37}$$

$$U_t = \frac{G_t}{\sqrt{\hat{V}_t} + \epsilon} \quad \text{(normalized adaptive step)} \tag{38}$$

$$\hat{U}_t = \frac{U_t}{\max(1, \mathrm{RMS}(U_t))} \quad \text{(stability clipping)} \tag{39}$$

Instead of storing $m \times n$ second-moment values, Adafactor stores only $m + n$ accumulators, reducing memory from $O(n_{\text{params}})$ to $O(\sqrt{n_{\text{params}}})$. This is most attractive when VRAM is dominated by optimizer state rather than activations. The tradeoff is reduced optimization expressiveness and potential instability on some tasks.

**Algorithm 14** Adafactor (Factorized Second-Moment Statistics)

```
Require: Gradient matrix $G_t \in \mathbb{R}^{m\times n}$, learning rate $\eta$, $\beta_2 \approx 0.999$
Ensure: Updated parameters
 1: Initialize: Row accumulators $R \leftarrow \epsilon \mathbf{1}_m^T$, Column $C \leftarrow \epsilon \mathbf{1}_n$
 2: for $t = 1, 2, \ldots, T$ do
 3:     $G_t \leftarrow \nabla f(\theta_t)$ (matrix gradient)
 4:     $R \leftarrow \beta_2 R + (1-\beta_2)(G_t^2)\mathbf{1}_n^T$          ▷ Row inner products
 5:     $C \leftarrow \beta_2 C + (1-\beta_2)\mathbf{1}_m^\top(G_t^2)$          ▷ Column inner products
 6:     Reconstructed variance: $\hat{V} \leftarrow (R \cdot C)/(\mathbf{1}_n^\top R)$          ▷ Via outer product
 7:     Normalized adaptive step: $U_t \leftarrow G_t/(\sqrt{\hat{V}} + \epsilon)$
 8:     Per-row RMS clipping: $\hat{U}_t \leftarrow U_t/\max(1, \mathrm{RMS}(U_t))$
 9:     $\theta_{t+1} \leftarrow \theta_t - \eta \hat{U}_t$
10: end for
```

**GaLore (Gradient Low-Rank Projection).** GaLore projects 2D gradients into a low-rank subspace before optimization, reducing optimizer-state memory while maintaining adaptive step scaling. For a 2D parameter matrix $W \in \mathbb{R}^{m\times n}$:

$$U, S, V = \mathrm{SVD}(G_t) \quad \text{(compute singular decomposition)} \tag{40}$$

$$P \in \mathbb{R}^{n\times r} \quad \text{or} \quad P^\top \in \mathbb{R}^{r\times m} \quad \text{(select top-}r\text{ singular vectors)} \tag{41}$$

$$G_{\text{low}} = P^\top G_t \quad \text{(project to low-rank space)} \tag{42}$$

$$\Delta_{\text{low}} = \mathrm{Adam}(G_{\text{low}}) \quad \text{(optimize in compressed space)} \tag{43}$$

$$\Delta = P\Delta_{\text{low}} \quad \text{(reconstruct in original space)} \tag{44}$$

By projecting into a rank-$r$ subspace (typically $r \ll \min(m, n)$), optimizer state is reduced from $O(mn)$ to $O(r(m+n))$ for 2D parameters. This complementary memory-saving approach is especially relevant when the model is too large for full-rank optimizer state. GaLore works synergistically with other techniques and has shown strong empirical results on billion-parameter models.

**Algorithm 15** GaLore (Gradient Low-Rank Projection)

```
Require: 2D parameter matrix $W \in \mathbb{R}^{m\times n}$, learning rate $\eta$, rank $r \ll \min(m, n)$
Ensure: Updated parameters
 1: Initialize: Adam state in low-rank space (if rank-2 variant)
 2: for $t = 1, 2, \ldots, T$ do
 3:     $G_t \leftarrow \nabla f(W_t)$
 4:     if $t \mod K = 1$ then          ▷ Periodic SVD
 5:         $U_t, S_t, V_t^\top \leftarrow \mathrm{SVD}(G_t)$ (full or thin)
 6:         Select projection: $P \leftarrow U_t[:, : r]$ or $P \leftarrow V_t[:, : r]$
 7:     end if
 8:     Project gradient: $G_{\text{low}} \leftarrow P^\top G_t$ (or $G_t P^\top$ for right projection)
 9:     Run Adam on low-rank: $\Delta_{\text{low}} \leftarrow \mathrm{Adam}(G_{\text{low}})$
10:     Reconstruct in original space: $\Delta \leftarrow P\Delta_{\text{low}}$ (or $\Delta_{\text{low}} P^\top$)
11:     $W_t \leftarrow W_t - \eta\Delta$
12: end for
```

**APOLLO, APOLLO-Mini, and Q-APOLLO.** The APOLLO family [55] begins from the observation that AdamW's element-wise denominator can be **coarsened into a structured learning-rate update**. Rather than storing dense moments for every parameter entry, APOLLO projects a matrix gradient $G_t \in \mathbb{R}^{m\times n}$ into a compact random subspace and tracks Adam-style moments there:

$$R_t = P_t G_t \quad \text{or} \quad R_t = G_t P_t^\top \tag{45}$$

$$M_t^R = \beta_1 M_{t-1}^R + (1-\beta_1) R_t \tag{46}$$

$$V_t^R = \beta_2 V_{t-1}^R + (1-\beta_2) R_t^2 \tag{47}$$

$$\widetilde{R}_t = \frac{M_t^R}{\sqrt{V_t^R} + \epsilon} \tag{48}$$

The projected state is *not* expanded back into a dense low-rank update as in SVD-based methods. Instead, APOLLO estimates a structured scaling tensor $S_t$ in the original space. In the standard APOLLO variant, that scaling is **channel-wise**: each row or channel receives its own norm ratio. In APOLLO-Mini, the scaling is reduced to a single **tensor-wise** scalar, corresponding to the rank-1 extreme described in the paper. The resulting parameter update is therefore Adam-like in adaptation but much closer to SGD in state cost:

$$W_t \leftarrow (1-\eta\lambda) W_{t-1} - \eta\,\alpha\,(G_t \odot S_t)$$

where $\alpha$ is the extra scale factor used to stabilize highly compressed variants.

The paper's practical contribution is twofold. First, APOLLO replaces expensive repeated SVD with periodically refreshed **Gaussian random projection**, so the systems burden is ordinary matrix multiplication rather than spectral decomposition. Second, the optimizer is unusually tolerant to extreme compression: even **APOLLO-Mini**, which keeps only rank-1 auxiliary state, remains competitive with or better than AdamW in the reported pre-training experiments while approaching SGD-level memory cost.

**Algorithm 16** APOLLO / APOLLO-Mini Structured Gradient Scaling

```
Require: Weight matrix $W \in \mathbb{R}^{m\times n}$ with $m \le n$, learning rate $\eta$, scale factor $\alpha$, decay rates
    $(\beta_1, \beta_2)$, weight decay $\lambda$, rank $r$, projection refresh interval $T$
Ensure: Updated parameters $W_T$
 1: Initialize projected moments: $M^R \leftarrow 0$, $V^R \leftarrow 0$, step $t \leftarrow 0$
 2: repeat
 3:     Compute gradient: $G_t \leftarrow \nabla_W \phi(W_t)$
 4:     if $t \bmod T = 0$ then
 5:         Sample Gaussian projector $P_t \sim \mathcal{N}(0, 1/r)$ with a fresh seed
 6:     end if
 7:     Project gradient: $R_t \leftarrow P_t G_t$                ▷ or $G_t P_t^\top$ depending on layout
 8:     Update projected AdamW moments: $M_t^R, V_t^R \leftarrow \text{AdamWState}(R_t; \beta_1, \beta_2)$
 9:     Normalize projected state: $\widetilde{R}_t \leftarrow M_t^R / (\sqrt{V_t^R} + \epsilon)$
10:     if APOLLO then
11:         $S_t \leftarrow \text{diag}(s_1^R, \ldots, s_m^R)$, where $s_i^R = \|\widetilde{R}_t[i,:]\|_2 / \|R_t[i,:]\|_2$
12:     else
13:         $S_t \leftarrow s_t^R$, where $s_t^R = \|\widetilde{R}_t\|_2 / \|R_t\|_2$
14:     end if
15:     Update weights: $W_t \leftarrow (1-\eta\lambda) W_{t-1} - \eta\,\alpha\,(G_t \odot S_t)$
16:     $t \leftarrow t + 1$
17: until convergence
```

**APOLLO-Mini.** APOLLO-Mini is the family member optimized for **extreme memory efficiency**. The paper motivates it by arguing that, in a rank-1 compact space, channel-wise scaling becomes too noisy, so the update is coarsened further to a single tensor-wise scale. The loss of granularity is partially offset by the explicit scaling factor $\alpha$ (the paper discusses values such as 128 in this regime), giving a useful point on the optimization Pareto frontier: very small state, no SVD overhead, and still strong pre-training behavior.

**Q-APOLLO.** The APOLLO paper also stresses that the family combines naturally with **quantization** for ultra-low-memory training. This repository turns that systems observation into a concrete optimizer variant, `q_apollo`. In the implementation, APOLLO's low-rank first and second moments are stored in quantized form together with per-tensor scales and offsets, and are dequantized only when needed for the next step. Q-APOLLO therefore preserves APOLLO's projected-gradient logic while reducing the precision of the remaining optimizer state, making it the most aggressive memory-saving member of the local optimizer stack.

**Prodigy (Approximating the Distance Estimate).** Prodigy adapts the effective step scale through a running distance-like statistic, eliminating the need for explicit learning-rate tuning. The algorithm maintains a cumulative distance estimate:

$$u_t = \frac{g_t}{\sqrt{v_t} + \epsilon} \quad \text{(unnormalized adaptive step)} \tag{49}$$

$$s_t = s_{t-1} + \langle u_t, \theta_t - \theta_0 \rangle \quad \text{(cumulative signed distance)} \tag{50}$$

$$d_t = \max(d_{t-1}, d_0 + d_{\text{coef}} \cdot s_t) \quad \text{(distance estimate with lower bound)} \tag{51}$$

$$\theta_{t+1} = \theta_t - \eta \cdot d_t \cdot u_t \tag{52}$$

where $d_0$ is an initial bound and $d_{\text{coef}}$ controls how aggressively the estimate adapts. The distance estimate $d_t$ captures the combined magnitude of past gradients weighted by actual parameter displacement, implementing an implicit effective learning rate. This distance-aware scaling achieves robust convergence across problem scales and batch sizes without requiring a learning-rate schedule. Prodigy reduces hyperparameter sensitivity by estimating step sizes from optimization geometry rather than problem-specific prior knowledge.

**Algorithm 17** Prodigy (Approximating the Distance Estimate)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, coeff $d_{\text{coef}}$, initial dist $d_0 > 0$
Ensure: Updated parameters $\theta_T$
1: Initialize: $s \leftarrow 0$ (cumulative signed distance), $d \leftarrow d_0$
2: for $t = 1, 2, \ldots, T$ do
3:     $g_t \leftarrow \nabla f(\theta_{t-1})$
4:     $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$                ▷ Second moment
5:     $u_t \leftarrow g_t / (\sqrt{v_t} + \epsilon)$                       ▷ Unnormalized adaptive
6:     Update distance: $s \leftarrow s + \langle u_t, \theta_t - \theta_0 \rangle$   ▷ Signed distance
7:     $d \leftarrow \max(d, d_0 + d_{\text{coef}} \cdot s)$                 ▷ Lower-bounded distance
8:     $\theta_t \leftarrow \theta_{t-1} - \eta \cdot d \cdot u_t$           ▷ Distance-scaled update
9: end forreturn $\theta_T$
```

## A.6 Second-Order, Geometric, and Orthogonality Optimizers

**Algorithm 18** Shampoo (Matrix Preconditioning via Kronecker Products)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, eigendecomposition interval $K$
Ensure: Updated parameters $\theta_T$
 1: Initialize: $L \leftarrow \epsilon I_m$, $R \leftarrow \epsilon I_n$ (left and right Gram matrices)
 2: for $t = 1, 2, \dots, T$ do
 3:     $G \leftarrow \nabla f(\theta_{t-1})$                                   ▷ Gradient
 4:     Update Gram matrices: $L \leftarrow L + GG^\top$, $R \leftarrow R + G^\top G$
 5:     if $t \mod K == 0$ then
 6:         $Q_L, \Lambda_L \leftarrow \text{eigh}(L)$                      ▷ Eigendecomposition of $L$
 7:         $Q_R, \Lambda_R \leftarrow \text{eigh}(R)$                      ▷ Eigendecomposition of $R$
 8:         $L^{-1/4} \leftarrow Q_L(\Lambda_L + \epsilon)^{-1/4} Q_L^\top$
 9:         $R^{-1/4} \leftarrow Q_R(\Lambda_R + \epsilon)^{-1/4} Q_R^\top$
10:     end if
11:     $\Delta\theta \leftarrow L^{-1/4} G R^{-1/4}$                       ▷ Preconditioned gradient
12:     $\theta_t \leftarrow \theta_{t-1} - \eta \cdot \Delta\theta$
13: end forreturn $\theta_T$
```

**Shampoo (Matrix Preconditioning via Kronecker Products).** Shampoo is a **structured second-order method** that computes matrix preconditioners from Kronecker-structured outer-product statistics. For a 2D parameter matrix $W \in \mathbb{R}^{m \times n}$:

$$L_t = L_{t-1} + G_t G_t^\top \quad \text{(left/row Gram matrix, } m \times m\text{)} \tag{53}$$

$$R_t = R_{t-1} + G_t^\top G_t \quad \text{(right/column Gram matrix, } n \times n\text{)} \tag{54}$$

$$L_t^{-1/4} = Q_L(\Lambda_L)^{-1/4} Q_L^\top \quad \text{(via eigendecomposition)} \tag{55}$$

$$R_t^{-1/4} = Q_R(\Lambda_R)^{-1/4} Q_R^\top \tag{56}$$

$$\Delta W_t = L_t^{-1/4} G_t R_t^{-1/4} \quad \text{(preconditioned gradient)} \tag{57}$$

Shampoo approximates the full-matrix Adagrad preconditioner $H^{-1/2}$ (where $H$ is the Hessian) using Kronecker-factored structure. Instead of storing a $(mn) \times (mn)$ preconditioner, Shampoo maintains two smaller matrices ($m \times m$ and $n \times n$), capturing cross-parameter correlations within and across groups. The method has proven effective at scale (Google production systems) and is well-suited to transformer architectures with high-rank structure. Eigendecompositions are performed periodically (e.g., every $K = 10$ steps) to amortize cost. Tradeoff: higher per-step compute and periodic $O(m^3 + n^3)$ eigendecomposition versus improved conditioning and accelerated convergence.

**SOAP (Shampoo with Adam in eigenbasis).** SOAP is a simplified variant of Shampoo that decouples preconditioning from momentum tracking. Instead of complex matrix algebra on preconditioned gradients, SOAP runs standard Adam in the eigenbasis of Shampoo's preconditioners:

$$L_t = L_{t-1} + G_t G_t^\top, \quad R_t = R_{t-1} + G_t^\top G_t \quad \text{(Gram accumulation)} \tag{58}$$

$$Q_L, \Lambda_L = \text{eigh}(L_t), \quad Q_R, \Lambda_R = \text{eigh}(R_t) \quad \text{(periodic eigendecomposition)} \tag{59}$$

$$G_t^{\text{rot}} = Q_L^\top G_t Q_R \quad \text{(rotate gradient into eigenbasis)} \tag{60}$$

$$m_t^{\text{rot}} = \beta_1 m_{t-1}^{\text{rot}} + (1-\beta_1) G_t^{\text{rot}} \tag{61}$$

$$v_t^{\text{rot}} = \beta_2 v_{t-1}^{\text{rot}} + (1-\beta_2)(G_t^{\text{rot}})^2 \quad \text{(Adam in rotated space)} \tag{62}$$

$$U_t^{\text{rot}} = \frac{m_t^{\text{rot}}}{\sqrt{v_t^{\text{rot}}} + \epsilon}, \quad U_t = Q_L U_t^{\text{rot}} Q_R^\top \quad \text{(rotate back)} \tag{63}$$

The key insight: all momentum tracking occurs in the well-conditioned eigenbasis, simplifying numerical stability and theoretical analysis. SOAP combines benefits of Shampoo (explicit curvature structure) and Adam (proven momentum mechanics), while being cleaner and often more stable than full Shampoo. Eigendecompositions are recomputed every $K$ steps, amortizing the cost.

**Algorithm 19** SOAP (Shampoo with Adam in Eigenbasis)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, $\beta_1, \beta_2$, eigendecomposition interval $K$
Ensure: Updated parameters $\theta_T$
 1: Initialize: $L \leftarrow \epsilon I_m$, $R \leftarrow \epsilon I_n$, $m \leftarrow 0$, $v \leftarrow 0$
 2: for $t = 1, 2, \ldots, T$ do
 3:     $G \leftarrow \nabla f(\theta_{t-1})$                                ▷ Gradient
 4:     Update Gram: $L \leftarrow L + GG^\top$, $R \leftarrow R + G^\top G$
 5:     if $t \mod K == 0$ then
 6:         $Q_L, \Lambda_L \leftarrow \text{eigh}(L)$, $Q_R, \Lambda_R \leftarrow \text{eigh}(R)$
 7:     end if
 8:     $G^{\text{rot}} \leftarrow Q_L^\top G Q_R$                          ▷ Rotate gradient into eigenbasis
 9:     $m \leftarrow \beta_1 m + (1-\beta_1) G^{\text{rot}}$       ▷ Exponential moving average (bias-correct outside)
10:     $v \leftarrow \beta_2 v + (1-\beta_2)(G^{\text{rot}})^2$          ▷ Second moment (bias-correct outside)
11:     $u \leftarrow m/(\sqrt{v} + \epsilon)$                              ▷ Adaptive step in eigenbasis
12:     $\Delta\theta \leftarrow Q_L u Q_R^\top$                             ▷ Rotate back to parameter space
13:     $\theta_t \leftarrow \theta_{t-1} - \eta \cdot \Delta\theta$
14: end forreturn $\theta_T$
```

**Lion (EvoLved Sign Momentum).** Lion achieves **minimal memory overhead** by using sign-based updates instead of full adaptive scaling:

$$c_t = \beta_1 m_t + (1-\beta_1) g_t \quad \text{(momentum input)} \tag{64}$$

$$\theta_{t+1} = \theta_t - \eta\,(\text{sign}(c_t) + \lambda\theta_t) \quad \text{(sign operator update)} \tag{65}$$

$$m_{t+1} = \beta_2 m_t + (1-\beta_2) g_t \quad \text{(momentum accumulation)} \tag{66}$$

All element-wise scaling operations are replaced with `sign()`, which outputs $\{-1, 0, +1\}$. This dramatically reduces memory compared to Adam-like methods: Lion stores only the momentum buffer, no second moment variance. The tradeoff: sign-based updates sacrifice the element-wise learning-rate adaptation that makes Adam effective. Lion is positioned not as a universally superior optimizer but as a specialized low-memory, high-throughput alternative for scenarios where activation memory dominates and some optimizer sophistication can be sacrificed. Works well in large-batch regimes and when hardware throughput is the primary constraint.

**Algorithm 20** Lion (EvoLved Sign Momentum)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, $\beta_1, \beta_2$, weight decay $\lambda$
Ensure: Updated parameters $\theta_T$
 1: Initialize: $m \leftarrow 0$ (momentum buffer)
 2: for $t = 1, 2, \ldots, T$ do
 3:     $g \leftarrow \nabla f(\theta_{t-1})$                                   ▷ Gradient
 4:     $c \leftarrow \beta_1 m + (1 - \beta_1) g$                              ▷ Momentum input
 5:     $\theta_t \leftarrow \theta_{t-1} - \eta(\text{sign}(c) + \lambda\theta_{t-1})$   ▷ Sign-based update with weight decay
 6:     $m \leftarrow \beta_2 m + (1 - \beta_2) g$                              ▷ Momentum accumulation (decoupled)
 7: end forreturn $\theta_T$
```

**Sophia (Second-Order Hessian Information with Optimized Approximation).** Sophia uses **diagonal Hessian estimates** for curvature-aware scaling without the memory and compute cost of dense second-order methods.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad \text{(first moment)} \tag{67}$$

$$h_t = \beta_2 h_{t-(k-1)} + (1 - \beta_2)\hat{h}_t \quad \text{(diagonal Hessian, updated every } k \text{ steps)} \tag{68}$$

$$\text{clip}(x, C) = \min(\max(x, -C), C) \quad \text{(element-wise clipping)} \tag{69}$$

$$\theta_{t+1} = \theta_t - \eta \cdot \text{clip}\left(\frac{m_t}{\max(\gamma h_t, \epsilon)}, 1\right) \tag{70}$$

where $\hat{h}_t$ is a diagonal estimate (e.g., Hutchinson-trace estimator) and $\gamma$ is a scaling factor. The diagonal Hessian $h_t$ captures local curvature, allowing the optimizer to take smaller steps in sharp directions and larger steps in flat directions. The clipping operation clip( $cdot, 1)$ prevents adaptive steps from exploding. Sophia belongs to the family of curvature-aware methods seeking better conditioning without the $O(n^2)$ or $O(n^3)$ cost of full second-order methods. Works particularly well in second-pass fine-tuning scenarios.

**Algorithm 21** Sophia (Diagonal Hessian with Clipped Updates)

```
Require: Initial parameters $\theta_0$, learning rate $\eta$, $\beta_1, \beta_2$, Hessian update freq $k$, clip threshold $C$
Ensure: Updated parameters $\theta_T$
 1: Initialize: $m \leftarrow 0$, $h \leftarrow \epsilon$ (diagonal Hessian estimate)
 2: for $t = 1, 2, \ldots, T$ do
 3:     $g \leftarrow \nabla f(\theta_{t-1})$
 4:     $m \leftarrow \beta_1 m + (1 - \beta_1) g$                  ▷ First moment (bias-correct outside)
 5:     if $t \mod k == 0$ then
 6:         $\hat{h} \leftarrow \text{HutchinsonEstimate}(g)$         ▷ Diagonal Hessian via Hutchinson
 7:         $h \leftarrow \beta_2 h + (1 - \beta_2)\hat{h}$            ▷ Update Hessian estimate
 8:     end if
 9:     $u \leftarrow \frac{m}{\max(\gamma h, \epsilon)}$              ▷ Adaptive step (element-wise)
10:     $u \leftarrow \text{clip}(u, C)$                           ▷ Element-wise clipping to $[-C, C]$
11:     $\theta_t \leftarrow \theta_{t-1} - \eta \cdot u$
12: end forreturn $\theta_T$
```

**Muon and Turbo-Muon (Orthogonality-Based Optimizers).** Muon and Turbo-Muon are **orthogonality-oriented optimizers** that reshape update geometry using Newton-Schulz polynomial iterations for orthogonalization. For a 2D parameter matrix $W$ with gradient $G_t$:

$$X_0 = \frac{G_t}{\|G_t\|_F + \epsilon} \quad \text{(normalized gradient)} \tag{71}$$

$$A_k = X_k X_k^\top \quad \text{(Gramian)} \tag{72}$$

$$B_k = bA_k + cA_k^2 \quad \text{(polynomial step, coefficients } b, c \text{ from Newton-Schulz)} \tag{73}$$

$$X_{k+1} = aX_k + B_k X_k \quad \text{(iteration } k = 0, 1, \ldots, 4) \tag{74}$$

$$W_{t+1} = W_t - \eta X_K \quad \text{(update with orthogonalized direction)} \tag{75}$$

Muon performs 5 Newton-Schulz iterations to produce an approximately orthogonal direction, ensuring updates respect geometric constraints. Turbo-Muon adds an **almost-orthogonal preconditioning** (AOL) step before iterations, reducing the number of required orthogonalization steps to 4. The rationale: orthogonal updates preserve norms during training, avoiding the norm creep observed in momentum-based methods. These optimizers show promise on large-scale training but require custom CUDA kernels for efficiency. Tradeoff: high per-step compute (matrix multiplications and polynomial iterations) versus fundamentally better-conditioned update geometry.

**Algorithm 22** Muon (Newton-Schulz Orthogonalization)

```
Require: Initial parameters $\theta$ (matrices), learning rate $\eta$, Newton-Schulz iters $K$
Ensure: Updated parameters $\theta$
 1: for each 2D parameter matrix $W$ do
 2:     $G \leftarrow \nabla f(W)$                                   ▷ Gradient
 3:     $X_0 \leftarrow \frac{G}{\|G\|_F+\epsilon}$                  ▷ Normalize gradient by Frobenius norm
 4:     for $k = 0, 1, \ldots, K-1$ do
 5:         $A_k \leftarrow X_k X_k^\top$                            ▷ Gramian (cost: $O(mn^2)$ or $O(m^2n)$ for tall/wide)
 6:         $B_k \leftarrow (3/2)A_k - (1/2)A_k^2$                   ▷ Newton-Schulz coefficients
 7:         $X_{k+1} \leftarrow B_k X_k$                             ▷ Polynomial step
 8:     end for
 9:     $W \leftarrow W - \eta X_K$                                  ▷ Update with orthogonalized direction
10: end forreturn updated $\theta$
```

**Algorithm 23** Turbo-Muon (AOL-Preconditioned Orthogonalization)

```
Require: Initial parameters $\theta$ (matrices), learning rate $\eta$, Newton-Schulz iters $K'$ (typically 4)
Ensure: Updated parameters $\theta$
 1: for each 2D parameter matrix $W$ do
 2:     $G \leftarrow \nabla f(W)$                                   ▷ Gradient
 3:     $X_0 \leftarrow \frac{G}{\|G\|_F+\epsilon}$                  ▷ Normalize gradient
 4:     AOL (Almost-Orthogonal preconditioner):                      ▷ Preconditioning step
 5:     $P \leftarrow (1.5I - 0.5X_0X_0^\top)$                       ▷ Preconditioning matrix
 6:     $X_0 \leftarrow PX_0$                                        ▷ Preconditioned start
 7:     for $k = 0, 1, \ldots, K'-1$ do
 8:         $A_k \leftarrow X_k X_k^\top$
 9:         $B_k \leftarrow (3/2)A_k - (1/2)A_k^2$
10:         $X_{k+1} \leftarrow B_k X_k$                             ▷ Reduced iterations due to AOL preconditioning
11:     end for
12:     $W \leftarrow W - \eta X_{K'}$                               ▷ Update with preconditioned orthogonalized direction
13: end forreturn updated $\theta$
```

| Group | Methods | Primary Goal | Interpretation from the Survey |
|---|---|---|---|
| Classical baseline | SGD, AdamW, RAdam | stability and reference baselines | These define the comparison floor for newer optimizer claims. |
| Momentum redesign | Adan, AdEMAMix, MARS, Cautious AdamW | faster or safer first-order adaptation | Best when convergence speed or noisy-gradient stability is the main concern. |
| Large-batch and schedule simplification | LAMB, Schedule-Free AdamW | operational robustness at scale | Reduce brittleness from batch-size growth or schedule engineering. |
| Memory-efficient | Adafactor, GaLore, APOLLO, APOLLO-Mini, Q-APOLLO, Lion | optimizer-state reduction | Most useful when VRAM is dominated by optimizer state rather than activations; APOLLO-family methods replace dense AdamW moments with projected or quantized structured scaling. |
| Curvature-aware | Shampoo, SOAP, Sophia | better conditioning | Prefer when richer geometry is worth implementation and compute overhead. |
| Geometry-oriented | Muon, Turbo-Muon | orthogonalized update structure | Specialized options for matrix geometry and representation shaping. |

## B Annex B: Dense, Recurrent, and Memory-Augmented Transformers

This comprehensive annex synthesizes modern transformer architectures beyond the standard softmax baseline. The field has evolved toward six primary paradigms: dense global attention with variants, recurrent state compression with decay, selective state-space models, continuous-depth numerical integration, memory-augmented architectures, and hybrid approaches. Understanding this taxonomy illuminates fundamental tradeoffs between expressiveness, computational cost, memory footprint, and deployment simplicity.

### B.1 Dense Attention Baselines: Standard and Sigmoid

#### B.1.1 Standard Softmax Attention

Standard multi-head self-attention [38] computes scaled dot-product similarity between token embeddings:

```
1: **Input:** Query matrix $\mathbf{Q} \in \mathbb{R}^{n\times d}$, Key matrix $\mathbf{K} \in \mathbb{R}^{n\times d}$, Value matrix $\mathbf{V} \in \mathbb{R}^{n\times d}$
2: scores ← $\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}} \in \mathbb{R}^{n\times n}$        ▷ compute pairwise similarities
3: attention_weights ← softmax(scores, axis = 1) $\in \mathbb{R}^{n\times n}$        ▷ normalize across keys
4: **Output:** $\mathbf{Y}$ = attention_weights · $\mathbf{V} \in \mathbb{R}^{n\times d}$        ▷ weighted value combination
```

For autoregressive generation, KV caching stores past keys and values to avoid $\mathcal{O}(n^2)$ recomputation. However, this linearly growing cache occupies $\sim n \cdot d_{\text{hidden}}$ bytes, which can consume hundreds of gigabytes for billion-parameter models. Standard attention achieves **perfect expressiveness** within the context window—any token can attend to any other token with learned weights—but pays the price of dense computation.

- **Training complexity**: $\mathcal{O}(n^2 \cdot d)$ time, $\mathcal{O}(n^2)$ space (attention matrix materialization).

- **Inference complexity**: $\mathcal{O}(n)$ time per token, $\mathcal{O}(n)$ space (KV cache).
- **Strengths**: Unparalleled expressiveness; perfect history recall; highly parallelizable.
- **Weaknesses**: Quadratic bottleneck prohibits very long contexts; KV cache dominates memory during generation.

#### B.1.2 Sigmoid Attention

Sigmoid attention replaces row-wise softmax with element-wise sigmoid activation [30]:

```
1: Input: $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ as above, learnable bias $\mathbf{b} \in \mathbb{R}^{n \times n}$
2: logits ← $\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}} + \mathbf{b}$
3: attention_weights ← $\sigma$(logits)                ▷ element-wise sigmoid, not softmax
4: Output: $\mathbf{Y}$ = attention_weights $\odot$ $\mathbf{V}$
```

Unlike softmax, sigmoid does not enforce a probability distribution (weights need not sum to 1), enabling stronger token independence. Theoretical analysis via mixture-of-experts shows sigmoid achieves superior sample complexity: $\mathcal{O}(n^{-0.51})$ convergence for ReLU experts versus softmax's $\mathcal{O}(n^{-0.24})$. However, empirical training revealed gradient instabilities at scale. The remedy is **hybrid-norm**—adding normalization after the attention output—which stabilizes gradients without sacrificing the theoretical benefits of element-wise gating.

- **Training complexity**: $\mathcal{O}(n^2 \cdot d)$ (identical to standard), but element-wise operations enable 17% inference speedup via FlashSigmoid.
- **Inference complexity**: $\mathcal{O}(n)$ per token with KV cache (asymptotically same, but lower constant factors).
- **Strengths**: Overcomes zero-sum competition; avoids row-wise synchronization; hardware-friendly implementation.
- **Weaknesses**: Requires careful stabilization (hybrid-norm); training instability at large scales without auxiliary loss.

## B.2 Recurrent and Retentive Architectures

### B.2.1 Retentive Networks (RetNet)

RetNet [36] unifies three computation paradigms: parallel training, recurrent inference, and chunkwise deployment. Its core innovation is the **retention mechanism**, which uses a fixed exponential decay matrix to model temporal importance:

```
1: Parallel (training) form:
2: decay_matrix[$i, j$] ← $\gamma^{i-j}$ for $i \geq j$, else 0      ▷ causal exponential decay
3: decay_matrix[$i, j$] ← 0 for $i < j$                               ▷ causal masking
4: $\mathbf{Y}_{\text{parallel}}$ ← ($\mathbf{Q}\mathbf{K}^\top$ $\odot$ decay_matrix)$\mathbf{V}$    ▷ element-wise multiplication with decay
1: Recurrent (inference) form:
2: for $t = 1, \ldots, n$ do
3:     $\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + \mathbf{k}_t \mathbf{v}_t^\top$          ▷ state update with decay
4:     $\mathbf{y}_t \leftarrow \mathbf{q}_t \mathbf{s}_t$                    ▷ output via query-state interaction
5: end for
```

The decay scalar $\gamma \in (0, 1)$ controls the temporal window. RetNet uses **multi-scale retention** with different $\gamma$ values per head (e.g., $\gamma = 1 - 2^{-5}, 1 - 2^{-6}, \ldots$), allowing short-term and long-term dependencies simultaneously. Chunkwise recurrent mode divides sequences into chunks, processes each chunk in parallel, and threads a recurrent state between chunks.

- **Training complexity**: $\mathcal{O}(n^2 \cdot d)$ (parallel), or $\mathcal{O}(n \cdot c \cdot d)$ (chunkwise recurrent with chunk size $c$).
- **Inference complexity**: $\mathcal{O}(1)$ per token, $\mathcal{O}(d^2)$ state space (fixed matrix).
- **Strengths**: Constant-time inference; triple computation paradigm; multi-scale consolidation.
- **Weaknesses**: Fixed decay imposes rigid inductive bias; may truncate learned long-range patterns.

### B.2.2 Mamba: Selective State-Space Models

Mamba [12] frames recurrence as a continuous-time dynamical system with input-dependent parameters, achieving both linear training and constant-time inference:

```
 1: **Continuous dynamics:** $h'(t) = \mathbf{A}h(t) + \mathbf{B}x(t), \quad y(t) = \mathbf{C}h(t)$
 2: **Discretization with step size $\Delta_t$:**
 3: $\bar{\mathbf{A}}_t \leftarrow \exp(\Delta_t \mathbf{A})$
 4: $\bar{\mathbf{B}}_t \leftarrow (\Delta_t \mathbf{A})^{-1}(\exp(\Delta_t \mathbf{A}) - \mathbf{I})\Delta_t \mathbf{B}_t$
 5: **Recurrent update:**
 6: for $t = 1, \ldots, n$ do
 7:     $\Delta_t \leftarrow \text{softplus}(\text{Linear}(x_t))$                ▷ input-dependent step size
 8:     $\mathbf{B}_t \leftarrow \text{Linear}(x_t), \quad \mathbf{C}_t \leftarrow \text{Linear}(x_t)$   ▷ input-dependent projection matrices
 9:     $h_t \leftarrow \bar{\mathbf{A}}_t h_{t-1} + \bar{\mathbf{B}}_t x_t$                 ▷ state transition
10:     $y_t \leftarrow \mathbf{C}_t h_t$                            ▷ output projectionContinuous
11: end for
```

The critical innovation is that $\mathbf{A}$ (the system matrix) is *not* input-dependent, but $\Delta_t$, $\mathbf{B}_t$, and $\mathbf{C}_t$ are, making the system **time-varying**. This selectivity allows the model to *ignore* irrelevant information by setting $\Delta_t$ near zero, effectively creating a gate. A hardware-aware parallel scan algorithm implements the recurrence efficiently on GPUs by fusing computation within SRAM, avoiding expensive HBM bandwidth.

- **Training complexity**: $\mathcal{O}(n \cdot d)$ via hardware-aware scan (linear in sequence length).
- **Inference complexity**: $\mathcal{O}(1)$ per token, $\mathcal{O}(d)$ state (hidden vector).
- **Strengths**: Achieves linear training and constant-inference simultaneously; practical on long sequences (millions of tokens).
- **Weaknesses**: State vector compression can weaken exact copying and dense associative recall versus full attention.

## B.3 Continuous-Depth Transformers: ODE Integration

### B.3.1 ODE Transformer

The ODE Transformer [52] interprets network depth as numerical integration of a continuous dynamical system, using higher-order Runge-Kutta solvers to reduce truncation error:

```
1: **Continuous formulation:** $\frac{dh(t)}{dt} = f_\theta(h(t), t)$ where $f_\theta$ is the transformer sub-network
2: **Runge-Kutta-4 discrete approximation:**
3: $k_1 \leftarrow f_\theta(h_t, t)$
4: $k_2 \leftarrow f_\theta(h_t + \frac{1}{2}k_1, t + \frac{1}{2}\Delta t)$
5: $k_3 \leftarrow f_\theta(h_t + \frac{1}{2}k_2, t + \frac{1}{2}\Delta t)$
6: $k_4 \leftarrow f_\theta(h_t + k_3, t + \Delta t)$
7: $h_{t+1} \leftarrow h_t + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4)$          ▷ weighted combination
```

Instead of simple Euler residual connections $h_{t+1} = h_t + f_\theta(h_t)$, the RK4 block computes four intermediate evaluations and combines them with classical RK4 weights. To avoid vanishing gradients, the architecture introduces **learned gating** that interpolates between intermediate approximations:

```
1: $g \leftarrow \sigma(\text{Linear}([k_1, k_2, k_3, k_4]))$        ▷ learnable gate
2: $h_{t+1} \leftarrow h_t + g \cdot k_1 + (1 - g) \cdot k_2$        ▷ interpolated refinement
```

This formulation reduces the effective number of parameters through weight sharing—the same $f_\theta$ is evaluated multiple times—while providing richer trajectory refinement.

- **Training complexity**: $\mathcal{O}(k \cdot n^2 \cdot d)$ where $k$ is the RK order (4 for RK4).
- **Inference complexity**: $\mathcal{O}(k \cdot n)$ per token (higher constant overhead).
- **Strengths**: Significantly higher accuracy on generation tasks (state-of-the-art BLEU); parameter-efficient via weight sharing.
- **Weaknesses**: High per-step compute; inference latency increases by constant factor $k$; complex gating required for stability.

## B.4 Test-Time Memory: Titans

The Titans architecture [2] introduces an orthogonal dimension: instead of static weights, the model maintains a learnable memory that is *updated during inference* based on a surprise-driven signal:

```
1: **Local short-term attention:** Apply standard or sparse attention within a fixed context window $c$
2: $y_t^{\text{local}} \leftarrow \text{Attention}(q_t, k_{[t-c:t]}, v_{[t-c:t]})$
3: **Memory update signal (surprise):**
4: $S_t \leftarrow \eta_t S_{t-1} - \theta_t \nabla_\ell(\mathcal{M}_{t-1}; x_t)$        ▷ surprise as gradient of loss w.r.t. memory
5: $\mathcal{M}_t \leftarrow (1 - \alpha_t)\mathcal{M}_{t-1} + S_t$        ▷ memory updated via EMA
6: **Long-term memory retrieval:**
7: $y_t^{\text{memory}} \leftarrow \mathcal{M}_t^*(q_t)$        ▷ query memory module for retrieved information
8: **Output combination:**
9: $y_t \leftarrow y_t^{\text{local}} + \text{gate}(y_t^{\text{memory}})$        ▷ combine local and retrieved memory
```

The memory module $\mathcal{M}$ is literally updated during the forward pass by computing gradients of an associative loss and applying SGD steps with momentum. The decay rates $\eta_t, \theta_t, \alpha_t$ are themselves input-dependent, allowing the model to switch memory paradigms when context shifts.

- **Training complexity**: Roughly $\mathcal{O}(c^2 + n \cdot f)$ where $c$ is local window and $f$ is memory update overhead.
- **Inference complexity**: $\mathcal{O}(c^2)$ local attention plus $\mathcal{O}(1)$ memory retrieval per token.
- **Strengths**: Handles extreme context lengths; enables true associative recall; memory adapts to input.
- **Weaknesses**: Significantly more complex; inference includes gradient computation; higher coordination overhead.

## B.5 Architectural Comparison and Synthesis

| **Architecture** | **Train** | **Infer** | **State** | **Key Characteristic** |
|---|---|---|---|---|
| Standard Attention | $O(n^2d)$ | $O(n)$ cache | $O(nd)$ | Perfect expressiveness, full token routing, KV bottleneck. |
| Sigmoid Attention | $O(n^2d)$ | $O(n)$ cache | $O(nd)$ | Element-wise gating, hardware-efficient, stable at scale with hybrid-norm. |
| RetNet | $O(n^2d)$ | $O(1)$ | $O(d^2)$ | Multi-scale decay, triple computation mode, fixed forgetting pattern. |
| Mamba | $O(nd)$ | $O(1)$ | $O(d)$ | Selective input-dependent dynamics, linear training and inference, hardware scan. |
| ODE Transformer | $O(kn^2d)$ | $O(kn)$ | $O(nd)$ | Numerical integration refinement, weight sharing via stages, higher accuracy. |
| Titans | $O(c^2+nf)$ | $O(c^2+1)$ | $O(1)$ | Test-time memory adaptation, extreme context, on-inference parameter updates. |

The field exhibits a clear progression along two axes: (i) **computational efficiency**, moving from $O(n^2)$ to $O(n)$ or $O(1)$, and (ii) **memory adaptivity**, shifting from static weights to dynamic, test-time updated models. Standard attention remains the expressiveness baseline; Mamba and RetNet represent the practical efficiency frontier; Titans introduces an orthogonal innovation axis (test-time learning). The choice of architecture reflects the fundamental engineering constraint: expressiveness versus deployment cost.

# C Annex C: Comprehensive Sparse Attention Mechanisms

## C.1 Executive Summary

Sparse attention mechanisms address the prohibitive $\mathcal{O}(n^2)$ computational and memory complexity of standard scaled dot-product attention by restricting the set of attended key-value pairs. Modern sparse attention spans a rich design space distinguished by four orthogonal dimensions: (i) **sparsity pattern** (fixed geometric, data-dependent, or frequency-based), (ii) **trainability** (end-to-end trained or inference-only), (iii) **sparsity unit** (individual tokens, local windows, or blocks), and (iv) **mechanism** (masking, selection, filtering, or compression). This annex synthesizes seven major sparse attention families—Sparse Transformer, Longformer, BigBird, SparseK, NSA, SpargeAttn, and FASA—providing mathematical formulations, algorithmic pseudocode, and comprehensive architectural comparisons.

## C.2 Sparse Transformer: Factorized Strided and Fixed Patterns

### C.2.1 Mathematical Formulation

The Sparse Transformer [9] reduces quadratic complexity to $\mathcal{O}(n\sqrt{n})$ by factorizing sparse attention into two complementary sparse heads. Let $n$ be the sequence length and $l = \lfloor\sqrt{n}\rfloor$ be the stride.

**Strided attention head** : Each position $i$ attends to every $l$-th previous position:

$$\mathcal{A}_i^{(\text{strided})} = \{j : j \leq i,\ (i-j) \bmod l = 0\}$$

**Fixed attention head** : Each position $i$ attends to positions within its local block plus fixed summary columns:

$$\mathcal{A}_i^{(\text{fixed})} = \{j : \lfloor j/l \rfloor = \lfloor i/l \rfloor\} \cup \{j : j \bmod l \in \{l-c, \dots, l-1\}\}$$

where $c$ is a hyperparameter controlling the number of summary columns (typically $c = 1$ or $c = 2$).

The attention computation for each head $h$ follows standard scaled dot-product rules restricted to the sparse set:

$$\text{Attn}_h(Q, K, V)_i = \text{softmax}\left(\frac{q_i^h K_{\mathcal{A}_i}^{h\top}}{\sqrt{d_k}}\right) V_{\mathcal{A}_i}^h$$

The key insight is that with the two factorized heads operating in parallel across a depth of $L$ transformer layers, every position can reach every other position through a path of length at most $L + 1$, preserving long-range reachability with a fraction of dense attention cost.

### C.2.2 Algorithmic Pseudocode

**Algorithm 24** Sparse Transformer Attention: Strided + Fixed Dual Heads

```
Require: Query $\mathbf{Q} \in \mathbb{R}^{n\times d}$, Key $\mathbf{K} \in \mathbb{R}^{n\times d}$, Value $\mathbf{V} \in \mathbb{R}^{n\times d}$
Require: Stride $l = \lfloor\sqrt{n}\rfloor$, summary columns $c \in \{1, 2\}$
Ensure: Output $\mathbf{Y} \in \mathbb{R}^{n\times d}$
 1: Head 1: Strided Attention
 2: for $i = 1, \dots, n$ do
 3:     $\mathcal{A}_i \leftarrow \{j : j \le i \text{ and } (i-j) \bmod l = 0\}$        ▷ Every $l$-th position
 4:     $\text{scores}_i \leftarrow \mathbf{q}_i \mathbf{K}_{\mathcal{A}_i}^\top / \sqrt{d_k}$
 5:     $\text{attn}_i \leftarrow \text{softmax}(\text{scores}_i)$
 6:     $\mathbf{y}_i^{(1)} \leftarrow \text{attn}_i \mathbf{V}_{\mathcal{A}_i}$
 7: end for
 8: Head 2: Fixed Block + Summary Attention
 9: for $i = 1, \dots, n$ do
10:     $\text{block\_start} \leftarrow \lfloor i/l \rfloor \cdot l$, $\text{block\_end} \leftarrow \min(i+1, \text{block\_start} + l)$
11:     $\mathcal{A}_i \leftarrow [\text{block\_start}, \text{block\_end}) \cup \{\text{cols from last } c \text{ columns}\}$
12:     $\text{scores}_i \leftarrow \mathbf{q}_i \mathbf{K}_{\mathcal{A}_i}^\top / \sqrt{d_k}$
13:     $\text{attn}_i \leftarrow \text{softmax}(\text{scores}_i)$
14:     $\mathbf{y}_i^{(2)} \leftarrow \text{attn}_i \mathbf{V}_{\mathcal{A}_i}$
15: end for
16: Concatenate: $\mathbf{Y} = [\mathbf{y}^{(1)} \| \mathbf{y}^{(2)}]$ and project via output linear layer return $\mathbf{Y}$
```

### C.2.3 Key Characteristics

- **Complexity**: $\mathcal{O}(n\sqrt{n} \cdot d)$ during training; $\mathcal{O}(n)$ per token during generation.
- **Trainable**: Yes; full backpropagation through selected positions.
- **Sparsity pattern**: Fixed and data-agnostic; patterns do not adapt to content.
- **Strength**: Structured reachability guarantees long-range dependencies; proven effective on long sequences (Enwik8, text images).
- **Limitation**: Fixed patterns may miss important but non-contiguous relationships; requires custom CUDA kernels for practical efficiency.

## C.3 Longformer: Sliding Window, Dilation, and Global Tokens

### C.3.1 Mathematical Formulation

Longformer [3] achieves linear $\mathcal{O}(n \cdot w)$ complexity by combining three complementary attention patterns.

**Sliding window attention** : Local neighborhood of size $w$:

$$\mathcal{A}_i^{(\text{window})} = \{j : |i-j| \leq w/2\}$$

**Dilated sliding window** : Windowed attention with dilation $d$, skipping stride $d+1$:

$$\mathcal{A}_i^{(\text{dilated})} = \{j : |i-j| \leq (w/2)(d+1) \text{ and } (i-j) \bmod (d+1) = 0\}$$

**Global attention** : Designated global tokens (e.g., [CLS]) attend to all positions and are attended by all:

$$\mathcal{A}_i^{(\text{global})} = \begin{cases} \{1,\ldots,n\} & \text{if } i \in \mathcal{G} \\ \mathcal{A}_i^{(\text{window})} \cup \mathcal{G} & \text{otherwise} \end{cases}$$

The three patterns are applied via separate projections. The local and global attention can use either the same projections or task-specific separate ones.

### C.3.2 Algorithmic Pseudocode

**Algorithm 25** Longformer: Sliding Window + Dilated + Global

```
Require: Query $\mathbf{Q} \in \mathbb{R}^{n\times d}$, Key $\mathbf{K}$, Value $\mathbf{V}$, window size $w$, dilation $d$, global token indices $\mathcal{G}$
Ensure: Output $\mathbf{Y}$
1: for $i = 1,\ldots,n$ do
2:     if $i \in \mathcal{G}$ then
3:         $\mathcal{A}_i \leftarrow \{1,\ldots,n\}$                ▷ Global token attends to all
4:     else
5:         Local window: $\mathcal{A}^{(\text{local})} \leftarrow [i-w/2, i+w/2] \cap [1,n]$
6:         Dilated offsets: $\mathcal{A}^{(\text{dilated})} \leftarrow \{j : (i-j) \bmod (d+1) = 0\} \cap \mathcal{A}^{(\text{dilated range})}$
7:         Combine: $\mathcal{A}_i \leftarrow \mathcal{A}^{(\text{local})} \cup \mathcal{A}^{(\text{dilated})} \cup \mathcal{G}$
8:     end if
9:     $\text{scores}_i \leftarrow \mathbf{q}_i \mathbf{K}_{\mathcal{A}_i}^\top / \sqrt{d_k}$
10:    $\text{attn}_i \leftarrow \text{softmax}(\text{scores}_i)$
11:    $\mathbf{y}_i \leftarrow \text{attn}_i \mathbf{V}_{\mathcal{A}_i}$
12: end forreturn $\mathbf{Y}$
```

### C.3.3 Key Characteristics

- **Complexity**: $\mathcal{O}(n \cdot w)$ where $w$ is the window size (linear when $w \ll n$).
- **Trainable**: Yes; drop-in replacement for standard attention.
- **Coverage**: With $L$ layers and window size $w$, receptive field grows to $L \times w$, covering entire sequence with shallow networks.
- **Strength**: Practical for document-level tasks; dilation expands local receptive fields with minimal overhead; global tokens provide query-document correspondence.
- **Limitation**: Window size remains a design hyperparameter; suboptimal for tasks requiring dense attention patterns.

## C.4 BigBird: Random, Local, and Global Sparse Graph

### C.4.1 Mathematical Formulation

BigBird [50] preserves universal approximation and Turing completeness using a principled mix of three sparsity patterns.

**Random connections** : Each position connects to $r$ randomly sampled positions:

$$\mathcal{A}_i^{(\text{random})} = \text{RandomSample}(\{1, \ldots, n\}, r)$$

**Local window** : Sliding window neighborhood:

$$\mathcal{A}_i^{(\text{local})} = \{j : |i - j| \leq w/2\}$$

**Global tokens** : Task-specific global anchors:

$$\mathcal{A}_i^{(\text{global})} = \begin{cases} \{1, \ldots, n\} & \text{if } i \in \mathcal{G} \\ \mathcal{A}_i^{(\text{random})} \cup \mathcal{A}_i^{(\text{local})} \cup \mathcal{G} & \text{otherwise} \end{cases}$$

The composite sparse mask $M$ is:

$$M_{ij} = \mathbf{1}[j \in \mathcal{A}_i^{(\text{random})} \cup \mathcal{A}_i^{(\text{local})} \cup \mathcal{A}_i^{(\text{global})}]$$

In practice, BigBird groups tokens into blocks of size $b$ and applies the sparsity decision at the block level, enabling efficient block-sparse implementations.

### C.4.2 Theoretical Guarantee

The authors prove that with $O(1)$ random connections and global tokens, the sparse graph maintains the same universal approximation and Turing completeness properties as dense attention. Formally, any computable function can be represented and any sequence of operations can be simulated within the BigBird connectivity graph.

### C.4.3 Algorithmic Pseudocode

**Algorithm 26** BigBird: Random + Local + Global Block-Sparse Attention

```
Require: Query $\mathbf{Q}$, Key $\mathbf{K}$, Value $\mathbf{V}$, block size $b$, random links per block $r$, global indices $\mathcal{G}$
Ensure: Output $\mathbf{Y}$
 1: Reshape into blocks: $n_b = \lceil n/b \rceil$ blocks
 2: for block $i = 0, \ldots, n_b - 1$ do
 3:     for position $j$ in block $i$ do
 4:         Local window: $\mathcal{A}_j^{(\text{local})}$ = nearby positions in block $\cup$ boundary positions
 5:         Random block sampling: Sample $r$ blocks uniformly at random, add all positions from those blocks
 6:         Global: Add all global token positions
 7:         $\mathcal{A}_j \leftarrow \mathcal{A}_j^{(\text{local})} \cup \mathcal{A}_j^{(\text{random})} \cup \mathcal{G}$
 8:     end for
 9: end for
10: for $i = 1, \ldots, n$ do
11:     $\text{scores}_i \leftarrow \mathbf{q}_i \mathbf{K}_{\mathcal{A}_i}^\top / \sqrt{d_k}$
12:     $\text{attn}_i \leftarrow \text{softmax}(\text{scores}_i)$
13:     $\mathbf{y}_i \leftarrow \text{attn}_i \mathbf{V}_{\mathcal{A}_i}$
14: end forreturn $\mathbf{Y}$
```

### C.4.4 Key Characteristics

- **Complexity**: $\mathcal{O}(n)$ or near-linear in optimal block-sparse implementation.
- **Trainable**: Yes.
- **Theoretical grounding**: Provably maintains universal approximation and Turing-completeness with sparse connectivity.
- **Strength**: Strong long-document performance on question answering, summarization, and genomics tasks.
- **Limitation**: Random sampling introduces variance and non-determinism; tuning of $r, w, g$ hyperparameters remains task-dependent.

## C.5 SparseK Attention: Differentiable Top-$k$ Selection

### C.5.1 Mathematical Formulation

SparseK [23] enables learnable sparsity via a **differentiable top-$k$ operator**. A scoring network $\phi_\theta$ evaluates key-value pair importance:

$$u_j = \phi_\theta(\mathbf{k}_j, \mathbf{q}_i) \in \mathbb{R}$$

The **SparseK operator** selects the top-$k$ scores while remaining differentiable. It computes a threshold $\tau(u)$ such that the sum of active scores equals $k$:

$$\text{SparseK}(u, k)_j = \max(u_j - \tau(u), 0), \quad \text{where} \quad \sum_j \max(u_j - \tau, 0) = k$$

The threshold $\tau$ is found via bisection. The attention becomes:

$$m_j = \text{SparseK}(u, k)_j, \quad \text{Attn}_i = \text{softmax}\left(\frac{\mathbf{q}_i K_{\text{sel}}^\top}{\sqrt{d_k}}\right) V_{\text{sel}}$$

where $K_{\text{sel}}, V_{\text{sel}}$ contain only the top-$k$ entries (those with $m_j > 0$).

### C.5.2 Algorithmic Pseudocode

**Algorithm 27** SparseK: Differentiable Top-$k$ Attention

```
Require: Query $\mathbf{q} \in \mathbb{R}^d$, Key matrix $\mathbf{K} \in \mathbb{R}^{n \times d}$, Value matrix $\mathbf{V} \in \mathbb{R}^{n \times d}$
Require: Scoring network $\phi_\theta$, target sparsity $k$
Ensure: Attention output $\mathbf{y}$
 1: Compute importance scores:
 2: for $j = 1, \dots, n$ do
 3:     $u_j \leftarrow \phi_\theta(\mathbf{k}_j, \mathbf{q})$
 4: end for
 5: Compute differentiable top-$k$ via threshold:
 6: $u_{\text{sorted}} \leftarrow \text{sort}(u, \text{descending} = \text{True})$
 7: $\text{cumsum} \leftarrow \text{cumsum}(u_{\text{sorted}})$
 8: Find $\rho = \max\{i : u_{\text{sorted}}[i] > 0 \text{ and } \text{cumsum}[i] \leq k\}$
 9: $\tau \leftarrow (u_{\text{sorted}}[\rho] - k)/(\rho + 1)$        ▷ Threshold for $k$ active elements
10: $m \leftarrow \max(u - \tau, 0)$        ▷ Differentiable selection mask
11: Apply mask and compute attention:
12: $K_{\text{sel}} \leftarrow K[m > 0]$, $V_{\text{sel}} \leftarrow V[m > 0]$
13: $\text{scores} \leftarrow \mathbf{q}K_{\text{sel}}^\top/\sqrt{d_k}$
14: $\text{attn} \leftarrow \text{softmax}(\text{scores})$
15: $\mathbf{y} \leftarrow \text{attn} \cdot V_{\text{sel}}$ return $\mathbf{y}$
```

### C.5.3 Key Characteristics

- **Complexity**: $\mathcal{O}(n \cdot d)$ during training (linear in sequence length); $\mathcal{O}(k)$ per token during generation.
- **Trainable**: Yes; end-to-end gradient flow through the SparseK operator.
- **Incremental generation**: Supports efficient constant-memory autoregressive generation.
- **Strength**: Seamlessly integrates into existing LLM architectures; minimal fine-tuning needed.
- **Limitation**: Scattered memory access from top-$k$ selection may limit cache efficiency on some hardware; scoring network adds overhead.

## C.6 NSA (Native Sparse Attention): Hardware-Aligned Hierarchical Branches

### C.6.1 Mathematical Formulation

NSA [48] decomposes sparse attention into three parallel branches that are combined through learned gates.

**Compression branch** : A learnable MLP $\varphi$ compresses key-value blocks:

$$\tilde{K}_t^{\text{cmp}} = [\varphi(k_{id+1:id+l})]_{1 \leq i \leq \lfloor (t-l)/d \rfloor} \quad , \quad \tilde{V}_t^{\text{cmp}} = [\varphi(v_{id+1:id+l})]$$

**Selection branch** : High-importance blocks are selected based on compressed scores:

$$p_t^{\text{cmp}} = \text{softmax}(q_t^\top \tilde{K}_t^{\text{cmp}}/\sqrt{d}), \quad I_t = \text{TopK}(p_t^{\text{cmp}}, n), \quad \tilde{K}_t^{\text{sel}} = \text{Gather}(K, I_t)$$

**Sliding window branch** : Fixed local context:

$$\tilde{K}_t^{\text{win}} = K_{t-w:t}, \quad \tilde{V}_t^{\text{win}} = V_{t-w:t}$$

**Gated combination** : The three attention outputs are combined via learned gates:

$$\alpha_t^{(\text{cmp})} = \sigma(\text{Linear}_{\text{cmp}}(q_t)), \quad \alpha_t^{(\text{sel})} = \sigma(\text{Linear}_{\text{sel}}(q_t)), \quad \alpha_t^{(\text{win})} = \sigma(\text{Linear}_{\text{win}}(q_t))$$

$$y_t = \alpha_t^{(\text{cmp})} \cdot \text{Attn}(q_t, \tilde{K}^{\text{cmp}}, \tilde{V}^{\text{cmp}}) + \alpha_t^{(\text{sel})} \cdot \text{Attn}(q_t, \tilde{K}^{\text{sel}}, \tilde{V}^{\text{sel}}) + \alpha_t^{(\text{win})} \cdot \text{Attn}(q_t, \tilde{K}^{\text{win}}, \tilde{V}^{\text{win}})$$

### C.6.2 Algorithmic Pseudocode

**Algorithm 28** NSA: Compressed + Selected + Window Branches with Learned Gating

```
Require: Query $\mathbf{q}_t$, cached Key/Value sequences, block size $l$, stride $d$, selection count $n$, window $w$
Require: Compression MLP $\varphi$, gate networks $\text{Linear}_{\text{cmp}}, \text{Linear}_{\text{sel}}, \text{Linear}_{\text{win}}$
Ensure: Output $\mathbf{y}_t$
 1: Compression branch:
 2: for $i = 1$ to $\lfloor t/d \rfloor$ do
 3:     $k_i^{\text{cmp}} \leftarrow \varphi(\mathbf{K}_{id+1:id+l})$        ▷ Compress block via MLP
 4:     $v_i^{\text{cmp}} \leftarrow \varphi(\mathbf{V}_{id+1:id+l})$
 5: end for
 6: $\tilde{\mathbf{K}}^{\text{cmp}} \leftarrow [k_1^{\text{cmp}}, \dots, k_{\lfloor t/d \rfloor}^{\text{cmp}}]$, $\tilde{\mathbf{V}}^{\text{cmp}} \leftarrow [v_1^{\text{cmp}}, \dots, v_{\lfloor t/d \rfloor}^{\text{cmp}}]$
 7: $\mathbf{y}_t^{(\text{cmp})} \leftarrow \text{SDPA}(\mathbf{q}_t, \tilde{\mathbf{K}}^{\text{cmp}}, \tilde{\mathbf{V}}^{\text{cmp}})$
 8: Selection branch:
 9: Compute scores on compressed: $\mathbf{s} \leftarrow \text{softmax}(\mathbf{q}_t \tilde{\mathbf{K}}^{\text{cmp}\top} / \sqrt{d})$
10: Select top-$n$ important blocks: $I \leftarrow \text{TopK}(\mathbf{s}, n)$
11: Gather full blocks: $\tilde{\mathbf{K}}^{\text{sel}} \leftarrow [\mathbf{K}_{I_i d+1:I_i d+l}]_i$, $\tilde{\mathbf{V}}^{\text{sel}} \leftarrow [\mathbf{V}_{I_i d+1:I_i d+l}]_i$
12: $\mathbf{y}_t^{(\text{sel})} \leftarrow \text{SDPA}(\mathbf{q}_t, \tilde{\mathbf{K}}^{\text{sel}}, \tilde{\mathbf{V}}^{\text{sel}})$
13: Window branch:
14: $\mathbf{y}_t^{(\text{win})} \leftarrow \text{SDPA}(\mathbf{q}_t, \mathbf{K}_{t-w:t}, \mathbf{V}_{t-w:t})$
15: Gated fusion:
16: $\alpha^{(\text{cmp})} \leftarrow \sigma(\text{Linear}_{\text{cmp}}(\mathbf{q}_t))$, $\alpha^{(\text{sel})} \leftarrow \sigma(\text{Linear}_{\text{sel}}(\mathbf{q}_t))$, $\alpha^{(\text{win})} \leftarrow \sigma(\text{Linear}_{\text{win}}(\mathbf{q}_t))$
17: $\mathbf{y}_t \leftarrow \alpha^{(\text{cmp})} \mathbf{y}_t^{(\text{cmp})} + \alpha^{(\text{sel})} \mathbf{y}_t^{(\text{sel})} + \alpha^{(\text{win})} \mathbf{y}_t^{(\text{win})}$ return $\mathbf{y}_t$
```

### C.6.3 Key Characteristics

- **Complexity**: $\mathcal{O}(t/d + n \cdot l + w)$ tokens attended per step (significantly reduced versus $\mathcal{O}(t)$ for dense).
- **Trainable**: Yes; direct training with all branches differentiable.
- **Hardware alignment**: Designed for efficient tensor core utilization; compression and selection reduce memory bandwidth pressure.
- **Strength**: 11.6× decoding speedup and 9× forward speedup on 64k sequences; outperforms full attention on many tasks.
- **Limitation**: Requires custom Triton/CUDA kernels; complex multi-branch architecture increases engineering overhead.

## C.7 FASA: Frequency-Aware Sparse Attention

### C.7.1 Mathematical Formulation

FASA [41] is a **training-free inference-time method** that exploits rotary position embedding (RoPE) structure. Under RoPE, the token embedding is rotated by $\theta_i = B^{-2(i-1)/d}$, decomposing the $d$-dimensional space into $d/2$ frequency chunks (FCs).

**Key insight** : Only a small subset ($< 1\%$) of FCs matter for contextual awareness; most encode positional patterns.

**Dominant frequency chunk identification** : For each layer $l$ and head $h$, identify dominant FCs via contextual agreement (CA):

$$\mathrm{CA}^{l,h,i} = \frac{|\mathrm{TopK}(\alpha^{l,h}) \cap \mathrm{TopK}(\alpha^{l,h,i})|}{K}$$

where $\alpha^{l,h}$ is the full attention mask and $\alpha^{l,h,i}$ is the mask using only FC $i$. Dominant FCs have high CA—they contribute meaningfully to the final attention pattern.

**Token importance prediction (TIP) stage** : Using only dominant FCs, compute lightweight importance per token:

$$S_t^{l,h} = \sum_{i \in \mathcal{I}_{\mathrm{dom}}^{l,h}} \alpha^{l,h,i}(\mathbf{q}_t, \mathbf{K}_{1:t}), \quad \mathcal{T}_t = \text{TopK-Indices}(S_t, N_{\mathrm{fac}})$$

**Focused attention computation (FAC) stage** : Compute full-precision attention on selected tokens:

$$\hat{\alpha}_{\mathrm{FAC}} = \mathrm{softmax}\left(\frac{\mathbf{q}_t \mathbf{K}_{\mathcal{T}_t}^\top}{\sqrt{d}}\right), \quad \mathbf{o}_t = \hat{\alpha}_{\mathrm{FAC}} \mathbf{V}_{\mathcal{T}_t}$$

### C.7.2 Algorithmic Pseudocode

```
Algorithm 29 FASA: Frequency-Aware Sparse Attention (Inference-Time)
Require: Current query $\mathbf{q}_t$, cached Key/Value $\mathbf{K}_{1:t}, \mathbf{V}_{1:t}$, RoPE base $B$, dominant FCs $\mathcal{I}_{\text{dom}}$
Require: TIP token budget $N_{\text{tip}}$, FAC token budget $N_{\text{fac}}$
Ensure: Output $\mathbf{o}_t$
 1: Stage 1: Token Importance Prediction (TIP)
 2: for layer $l$ and head $h$ do
 3:     for position $i = 1$ to $t$ do
 4:         Compute importance from dominant FCs: $s_i \leftarrow 0$
 5:         for $fc \in \mathcal{I}_{\text{dom}}^{l,h}$ do
 6:             Rotate query and key into FC $fc$: $\mathbf{q}^{(fc)}, \mathbf{k}_i^{(fc)}$
 7:             $s_i^{(fc)} \leftarrow \text{softmax}(\mathbf{q}^{(fc)}\mathbf{k}_i^{(fc)\top}/\sqrt{d})$
 8:             $s_i \leftarrow s_i + s_i^{(fc)}$
 9:         end for
10:     end for
11:     $\mathcal{T}_t \leftarrow \text{TopK} - \text{Indices}([s_1, \ldots, s_t], N_{\text{fac}})$      ▷ Select top tokens
12: end for
13: Stage 2: Focused Attention Computation (FAC)
14: $\text{scores}_{\text{fac}} \leftarrow \mathbf{q}_t\mathbf{K}_{\mathcal{T}_t}^{\top}/\sqrt{d}$      ▷ Full-precision dot product
15: $\alpha_{\text{fac}} \leftarrow \text{softmax}(\text{scores}_{\text{fac}})$      ▷ Full softmax
16: $\mathbf{o}_t \leftarrow \alpha_{\text{fac}}\mathbf{V}_{\mathcal{T}_t}$      ▷ Weighted value sum return $\mathbf{o}_t$
```

### C.7.3 Key Characteristics

- **Complexity**: TIP is $\mathcal{O}(t \cdot N_{\text{tip}})$; FAC is $\mathcal{O}(N_{\text{fac}} \cdot d)$.
- **Trainable**: No; training-free inference optimization.
- **Applicability**: Requires RoPE-based models; extended to ALiBi and MLA with modifications.
- **Strength**: Near-oracle accuracy with $\leq 256$ tokens out of millions; up to $2.56\times$ decoding speedup; orthogonal to quantization.
- **Limitation**: Offline offline calibration required; dominant FC identification adds preprocessing overhead.

## C.8 SpargeAttn: Two-Stage Block-Level Filtering

### C.8.1 Mathematical Formulation

SpargeAttn [53] is a **universal training-free method** that predicts which blocks of the attention matrix will contain negligible values and skips computation for those blocks.

**Stage 1 — Sparse prediction** : Compute low-cost proxy importance $\hat{s}_{ij}$ for block $(i, j)$:

$$\hat{s}_{ij} = f_{\text{pred}}(\mathbf{Q}_i, \mathbf{K}_j; \text{hyperparams}), \quad \text{skip if } \hat{s}_{ij} < \epsilon_1$$

Common predictors include block-mean similarity or self-similarity statistics.

**Stage 2 — Softmax-aware filtering** : After computing $\tilde{P}_{ij} = \text{softmax}(Q_i K_j^\top/\sqrt{d})$, check if the block's maximum probability is negligible relative to the running softmax maximum:

$$\text{skip } P_{ij}V_j \quad \text{if} \quad \max(\tilde{P}_{ij}) < e^{m_{\text{old}} - m_{\text{new}}} \cdot \epsilon_2$$

where $m_{\text{old}}, m_{\text{new}}$ are online softmax running maxima (computed via FlashAttention-style numerics).

### C.8.2 Algorithmic Pseudocode

**Algorithm 30** SpargeAttn: Two-Stage Block-Sparse Filtering

```
Require: Query blocks $\mathbf{Q}_i$ for $i = 1, \dots, n_b$, Key blocks $\mathbf{K}_j$ for $j = 1, \dots, n_b$, Value blocks $\mathbf{V}_j$
Require: Prediction threshold $\epsilon_1$, softmax threshold $\epsilon_2$, block size $b_s$
Ensure: Output $\mathbf{Y}$
 1: Initialize: online softmax max $m_{\text{global}} \leftarrow -\infty$
 2: for block row $i = 1$ to $n_b$ do
 3:     Initialize: block row output $\mathbf{Y}_i \leftarrow 0$, local max $m_i \leftarrow -\infty$
 4:     for block column $j = 1$ to $n_b$ do
 5:         Stage 1: Sparse Prediction
 6:         Compute proxy importance: $\hat{s}_{ij} \leftarrow f_{\text{pred}}(\mathbf{Q}_i, \mathbf{K}_j)$
 7:         if $\hat{s}_{ij} < \epsilon_1$ then
 8:             skip this block $[i, j]$                              ▷ Early termination
 9:             continue                                                ▷ Move to next block
10:         end if
11:         Stage 2: Compute and Filter
12:         $\tilde{P}_{ij} \leftarrow \text{softmax}(\mathbf{Q}_i \mathbf{K}_j^\top/\sqrt{d})$
13:         $m_{\text{block}} \leftarrow \max(\tilde{P}_{ij})$
14:         if $m_{\text{block}} < e^{m_i - m_{\text{global}}} \cdot \epsilon_2$ then
15:             Block contributes negligibly; skip                     ▷ Softmax-aware pruning
16:             continue
17:         end if
18:         Update global max: $m_i \leftarrow \max(m_i, m_{\text{block}})$
19:         Accumulate: $\mathbf{Y}_i \leftarrow \mathbf{Y}_i + \tilde{P}_{ij}\mathbf{V}_j$
20:     end for
21:     $m_{\text{global}} \leftarrow \max(m_{\text{global}}, m_i)$
22: end forreturn $\mathbf{Y}$
```

### C.8.3 Key Characteristics

- **Complexity**: Empirical $\mathcal{O}(n^2 \cdot s)$ where $s$ is the fraction of non-skipped blocks (typically 0.2–0.5).
- **Trainable**: No; plug-and-play acceleration for existing models.
- **Universality**: Works on language models, image diffusion models, and video generation.
- **Strength**: 2.5–5× speedup compared to dense or previous sparse methods; compatible with quantization (SageAttention integration).
- **Limitation**: Speedup depends on inherent sparsity; block-level granularity may miss finer patterns; threshold tuning required per model family.

### C.9 Comparative Summary

| **Method** | **Complexity** | **Trainable** | **Unit** | **Year** | **Primary Contribution** |
|---|---|---|---|---|---|
| Sparse Transformer | $O(n\sqrt{n}d)$ | Y | Token pattern | 2019 | Factorized strided + fixed masks |
| Longformer | $O(nwd)$ | Y | Local + global | 2020 | Linear scaling with dilation |
| BigBird | $O(n)$ | Y | Graph sparse | 2020 | Theoretical completeness proof |
| SparseK | $O(nd)$ | Y | Top-$k$ tokens | 2024 | Differentiable selection |
| NSA | $O((t/d + nl' + w)d)$ | Y | Multi-branch | 2025 | Hardware-aligned 3-branch design |
| FASA | $O(tN_{\text{tip}} + N_{\text{fac}}d)$ | N | Freq chunks | 2026 | RoPE frequency insight |
| SpargeAttn | $O(n^2 s \cdot d)$ | N | Block filter | 2025 | Two-stage universal filtering |

### C.10 Design Space and Selection Criteria

The seven sparse attention methods occupy complementary positions in a multidimensional design space:

- **Geometric/fixed patterns** (Sparse Transformer, Longformer): Simple to implement and analyze, but patterns do not adapt to content.
- **Learned selection** (SparseK, NSA): Enable adaptation through trainable selection networks, at the cost of higher implementation complexity.
- **Training-free acceleration** (FASA, SpargeAttn): Enable drop-in speedup of existing models without retraining, suited for deployed systems.
- **Theoretical guarantees** (BigBird): Provide formal proofs of expressive completeness, important for understanding safety margins.

**Selection guidance:**

1. **New model training** where resources permit: NSA or SparseK for maximal inference speedup; BigBird for theoretical assurance.
2. **Long-context document understanding**: Longformer or FASA for practical simplicity; NSA for extreme scale.
3. **Accelerating existing models**: FASA for RoPE-based LLMs; SpargeAttn for any architecture.
4. **Constrained environments**: Sparse Transformer for simplicity and proven efficiency at moderate scale.

## D Annex D: Gated Attention Families—Complete Literature Analysis

### D.1 Executive Summary: Gating for Memory Control

The emerging field of *gated attention mechanisms* addresses a fundamental challenge in sequence modeling: how should information be retained, forgotten, and updated as the model processes new tokens? Traditional additive recurrent states accumulate all information without erasure, leading to memory saturation and inability to adapt to changing context. Softmax attention provides full expressiveness but uses $\mathcal{O}(Ld)$ KV cache during inference, limiting context window size. Gated mechanisms provide a middle ground: selective control over information survival.

The unifying principle across gated architectures is that gating controls *what information survives*. Gates operate at three distinct levels:

1. **Recurrent state decay** (GLA, HGRN2): Multiplicative gates on the memory matrix $S_t$ control how much previous state persists into the next step.
2. **Write strength in recurrent updates** (DeltaNet, Gated DeltaNet): Gates control the magnitude of new information written into memory.
3. **Softmax logit biasing** (FoX): Gates inject token-level recency bias into attention logit computation.
4. **Post-attention sparsity** (Gated Softmax): Gates selectively scale SDPA output channels.

This appendix synthesizes seven major gated-attention architectures published 2023–2025, analyzing their mathematical foundations, hardware efficiency characteristics, and empirical trade-offs.

## D.2 1. Gated Linear Attention (GLA)

### D.2.1 Mathematical Foundation

Gated Linear Attention [44] augments vanilla linear attention (which uses a matrix-valued recurrent state) with data-dependent multiplicative decay. Standard linear attention reformulates the traditional softmax mechanism as a recurrence over hidden states:

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \mathbf{v}_t\mathbf{k}_t^\top, \quad \mathbf{o}_t = \mathbf{S}_t\mathbf{q}_t$$

where state $\mathbf{S}_t \in \mathbb{R}^{d_v \times d_k}$ accumulates outer products. This purely additive formulation suffers from "memory overload": information accumulates monotonically and cannot be erased, degrading performance on tasks requiring context selection.

GLA introduces a data-dependent **diagonal gating matrix** $\mathbf{G}_t \in [0,1]^{d_k \times d_k}$:

$$\mathbf{S}_t = \mathbf{G}_t \odot \mathbf{S}_{t-1} + \mathbf{v}_t\mathbf{k}_t^\top, \quad \mathbf{o}_t = \mathbf{S}_t\mathbf{q}_t$$

The gate $\mathbf{G}_t$ is parameterized with an outer-product structure:

$$\mathbf{G}_t = \mathbf{1}\boldsymbol{\alpha}_t^\top, \quad \boldsymbol{\alpha}_t = \text{logsigmoid}\left(\mathbf{W}_{gk}\mathbf{x}_t\right)/c$$

where $\mathbf{W}_{gk}$ is a low-rank projection ($\mathbb{R}^{d \to d_k}$) and $c \approx 16$ is a normalizer. The logsigmoid activation maps $\alpha_t \in \mathbb{R}$ to approximately $[-1/2, 0]$, and division by $c$ further contracts this to near-identity for initialization. The resulting gate $G_t = 1 + \alpha_t \approx 1$ near initialization, then learns to decay ($\alpha_t \to -\infty$) historical information.

**Algorithm 31** Gated Linear Attention (Recurrent Form)

```
Require: Input $\mathbf{x}_t$; prior state $\mathbf{S}_{t-1}$
Require: Projections: $W_q, W_k, W_v$ (standard); $W_{gk}$ (gate projection)
Ensure: Output $\mathbf{o}_t$ and updated state $\mathbf{S}_t$
1: $\mathbf{q}_t \leftarrow W_q\mathbf{x}_t$
2: $\mathbf{k}_t \leftarrow W_k\mathbf{x}_t$
3: $\mathbf{v}_t \leftarrow W_v\mathbf{x}_t$
4: Compute gate: $\boldsymbol{\alpha}_t \leftarrow \mathrm{logsigmoid}(W_{gk}\mathbf{x}_t)/16$    ▷ Per-key-dim decay
5: Apply outer-product gating: $\mathbf{G}_t \leftarrow \mathbf{1}\boldsymbol{\alpha}_t^\top$    ▷ diagonal outer product
6: Decay previous state: $\mathbf{S}_t^{\mathrm{decay}} \leftarrow \mathbf{G}_t \odot \mathbf{S}_{t-1}$    ▷ element-wise product
7: Add new association: $\mathbf{S}_t \leftarrow \mathbf{S}_t^{\mathrm{decay}} + \mathbf{v}_t\mathbf{k}_t^\top$
8: Retrieve via query: $\mathbf{o}_t^{\mathrm{raw}} \leftarrow \mathbf{S}_t\mathbf{q}_t$
9: Apply output gate: $\mathbf{o}_t \leftarrow \mathrm{GroupNorm}(\mathbf{o}_t^{\mathrm{raw}}) \otimes \mathrm{SiLU}(W_g\mathbf{x}_t)$ return $\mathbf{o}_t$, $\mathbf{S}_t$
```

### D.2.2 Hardware-Efficient Training

For parallel training, GLA uses a chunkwise block-wise algorithm that groups tokens into chunks $C$ and computes recurrent transitions in parallel:

$$\mathbf{O}_{[t]} = \overleftarrow{\mathbf{Q}}_{[t]}\mathbf{S}_{[t]}^\top + \left(\mathbf{Q}_{[t]}\mathbf{K}_{[t]}^\top \odot \boldsymbol{\Gamma}_{[t]}\right)\mathbf{V}_{[t]}$$

where $\overleftarrow{\mathbf{Q}}_{[t]}$ are attending to the state at the chunk boundary (lookback), and $\boldsymbol{\Gamma}_{[t]}$ is the causal mask with decay-aware scaling:

$$\boldsymbol{\Gamma}_{[t],ij} = \begin{cases} \prod_{l=j}^{i-1} \alpha_l & \text{if } i > j \text{ (lookback)} \\ \prod_{l=1}^{i-j} \alpha_l & \text{if } i \leq j \text{ (in-chunk)} \end{cases}$$

This design maximizes matmul operations susceptible to tensor-core acceleration, achieving sub-quadratic total complexity $\mathcal{O}(nLd^2/C)$ where $L$ is the number of attention heads and $C$ is chunk size.

### D.2.3 Strengths and Limitations

| **Aspect** | **Strengths** | **Limitations** |
|---|---|---|
| Computational efficiency | Sub-quadratic training via chunkwise parallelism. Constant-memory inference $O(d^2)$. | Requires custom CUDA/Triton kernels; not available in all frameworks. |
| Context generalization | Extends 2K-token training to 20K+ tokens with negligible perplexity degradation. | Still underperforms softmax on retrieval-heavy benchmarks (needle-in-haystack). |
| Selectivity | Per-key-dimension data-dependent decay. | Gate structure limits per-key-value selectivity; no direct control over which specific associations to erase. |

## D.3 2. DeltaNet: Error-Correcting Linear Attention

### D.3.1 Mathematical Foundation

DeltaNet [45] applies a classical **delta learning rule** to linear attention state updates. Instead of additive accumulation, DeltaNet computes the difference between the predicted value (retrieved from memory) and the target value, then performs an error-correcting update:

$$\mathbf{S}_t = \mathbf{S}_{t-1}(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top) + \beta_t \mathbf{v}_t \mathbf{k}_t^\top$$

where $\beta_t \in (0,1)$ is a data-dependent **writing strength** scalar. This update can be decomposed as an erase-write operation:

$$\mathbf{v}_t^{\text{old}} = \mathbf{S}_{t-1}\mathbf{k}_t \quad \text{(current prediction)}, \quad \mathbf{v}_t^{\text{updated}} = \beta_t \mathbf{v}_t + (1-\beta_t)\mathbf{v}_t^{\text{old}} \quad \text{(blend old/new)}$$

so that:

$$\mathbf{S}_t = \mathbf{S}_{t-1} - \mathbf{v}_t^{\text{old}} \mathbf{k}_t^\top + \mathbf{v}_t^{\text{updated}} \mathbf{k}_t^\top$$

The key insight is that this update rule minimizes an online MSE loss at each timestep:

$$\mathcal{L}_t(\mathbf{S}) = \tfrac{1}{2}\|\mathbf{S}\mathbf{k}_t - \mathbf{v}_t\|^2 \Rightarrow \nabla_{\mathbf{S}}\mathcal{L}_t = (\mathbf{S}\mathbf{k}_t - \mathbf{v}_t)\mathbf{k}_t^\top$$

One step of SGD with learning rate $\beta_t$ yields the delta rule update. This connection to test-time training (TTT) provides theoretical grounding: DeltaNet is directly optimizing for value prediction at inference time.

### D.3.2 Efficient Parallel Training

DeltaNet's transition matrix $\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top$ is a generalized Householder matrix (rank-1 update to identity). For chunkwise training, DeltaNet uses the **WY representation**, which represents the product of $m$ such rank-1 updates compactly:

$$\prod_{i=1}^{m}(\mathbf{I} - \beta_i \mathbf{k}_i \mathbf{k}_i^\top) = \mathbf{I} - \mathbf{W}\mathbf{Y}^\top$$

where $\mathbf{W}, \mathbf{Y} \in \mathbb{R}^{d\times m}$ are constructed recursively. This factorization enables matmul-rich parallelism without materializing the full product, keeping chunkwise training efficient.

**Algorithm 32** DeltaNet State Update with WY Representation

```
Require: Input chunk $\mathbf{X}_{[t]}$; prior state $\mathbf{S}_{t-1}$
Require: L2-normalized queries $\hat{\mathbf{Q}}_t$, keys $\hat{\mathbf{K}}_t$, values $\mathbf{V}_t$
Ensure: Output $\mathbf{O}_{[t]}$ and updated state $\mathbf{S}_t$
 1: Chunkwise phase:
 2: for position $i = 1$ to chunk_size do
 3:     Normalize: $\hat{\mathbf{k}}_i \leftarrow \hat{\mathbf{K}}_t[i]/\|\hat{\mathbf{K}}_t[i]\|$, $\hat{\mathbf{q}}_i \leftarrow \hat{\mathbf{Q}}_t[i]/\|\hat{\mathbf{Q}}_t[i]\|$
 4:     Compute write strength: $\beta_i \leftarrow \text{sigmoid}(W_\beta \mathbf{x}_i)$
 5:     Prediction: $\hat{\mathbf{v}}^{\text{pred}} \leftarrow \mathbf{S}_{i-1}^{\text{in-chunk}} \hat{\mathbf{k}}_i$
 6:     Error-aware blend: $\mathbf{v}_i^{\text{update}} \leftarrow \beta_i(\mathbf{v}_i - \hat{\mathbf{v}}^{\text{pred}})$
 7:     Erase-write: $\mathbf{S}_i^{\text{in-chunk}} \leftarrow \mathbf{S}_{i-1}^{\text{in-chunk}} - \hat{\mathbf{v}}^{\text{pred}} \hat{\mathbf{k}}_i^\top + (\beta_i \mathbf{v}_i + (1-\beta_i)\hat{\mathbf{v}}^{\text{pred}})\hat{\mathbf{k}}_i^\top$
 8:     Output: $\mathbf{o}_i \leftarrow \mathbf{S}_i^{\text{in-chunk}} \hat{\mathbf{q}}_i$
 9: end for
10: Inter-chunk phase: Use WY representation of transition products return $\mathbf{O}_{[t]}$, $\mathbf{S}_t$
```

### D.3.3 Strengths and Limitations

| Aspect | Strengths | Limitations |
|---|---|---|
| Associative recall | Perfect recall on Multi-Query Associative Recall (MQAR) benchmark; error-correcting semantics naturally fit retrieval patterns. | Without global forgetting, memory still crowds over extreme sequence lengths; requires periodic reset mechanisms for unbounded context. |
| Theory | Grounded in online MSE optimization; clear connection to test-time training paradigm. | Requires L2-normalized keys for numerical stability; double-width normalization adds overhead. |
| Throughput | WY representation enables efficient chunkwise training. | Slightly slower than Mamba2 per-token due to richer transition matrices (rank-1 instead of diagonal). |

## D.4 3. Gated DeltaNet: Synthesis of Gating and Error Correction

### D.4.1 Mathematical Foundation

Gated DeltaNet [46] synthesizes the strengths of GLA and DeltaNet by combining a **decay gate** $\alpha_t$ (from GLA) with the **writing strength gate** $\beta_t$ (from DeltaNet):

$$\mathbf{S}_t = \mathbf{S}_{t-1}\left(\alpha_t(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top)\right) + \beta_t \mathbf{v}_t \mathbf{k}_t^\top$$

Rewriting for clarity:

$$\mathbf{S}_t = \alpha_t \mathbf{S}_{t-1}(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top) + \beta_t \mathbf{v}_t \mathbf{k}_t^\top$$

The two gates are complementary:

- $\alpha_t \to 0$ rapidly erases historical state: $\mathbf{S}_t \approx \beta_t \mathbf{v}_t \mathbf{k}_t^\top$ (context reset).
- $\alpha_t \to 1$ recovers pure delta-rule behavior: $\mathbf{S}_t \approx \mathbf{S}_{t-1}(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top) + \beta_t \mathbf{v}_t \mathbf{k}_t^\top$ (targeted updates).
- $\beta_t \to 0$ skips writing but decays: $\mathbf{S}_t \approx \alpha_t \mathbf{S}_{t-1}$ (pure forgetting).
- $\beta_t \to 1$ replaces memory aggressively: $\mathbf{S}_t \approx \alpha_t \mathbf{S}_{t-1} + \mathbf{v}_t \mathbf{k}_t^\top$ (replacement).

From an online learning perspective, Gated DeltaNet corresponds to:

$$\min_{\mathbf{S}_t} \|\mathbf{S}_t - \alpha_t \mathbf{S}_{t-1}\|_F^2 - 2\langle \mathbf{S}_t \mathbf{k}_t, \beta_t(\mathbf{v}_t - \alpha_t \mathbf{S}_{t-1} \mathbf{k}_t)\rangle$$

which introduces an adaptive weight decay $\alpha_t$ into an SGD-like update—analogous to decoupled weight decay in deep learning optimization.

**Algorithm 33** Gated DeltaNet Parallel Chunkwise Forward

```
Require: Input chunk $\mathbf{X}_{[i]}$; prior state $\mathbf{S}_{i-1}$; chunk size $C$
Require: Projections: $W_q, W_k, W_v, W_\alpha, W_\beta$
Ensure: Output $\mathbf{O}_{[i]}$ and state $\mathbf{S}_i$
 1: In-chunk computation:
 2: for $j = 1$ to $C$ do
 3:     $\mathbf{q}_j \leftarrow W_q\mathbf{x}_j$, $\mathbf{k}_j \leftarrow W_k\mathbf{x}_j$, $\mathbf{v}_j \leftarrow W_v\mathbf{x}_j$
 4:     $\alpha_j \leftarrow \text{sigmoid}(W_\alpha\mathbf{x}_j)$, $\beta_j \leftarrow \text{sigmoid}(W_\beta\mathbf{x}_j)$
 5:     Apply combined gate: $\mathbf{S}_j \leftarrow \alpha_j\mathbf{S}_{j-1}(\mathbf{I} - \beta_j\mathbf{k}_j\mathbf{k}_j^\top) + \beta_j\mathbf{v}_j\mathbf{k}_j^\top$
 6:     $\mathbf{o}_j \leftarrow \mathbf{S}_j\mathbf{q}_j$
 7: end for
 8: Inter-chunk recurrence:
 9: $\mathbf{S}_i \leftarrow \mathbf{S}_C$ from in-chunk; propagate across chunks via cumulative $\prod \alpha$ masks
10: Output gating: $\mathbf{O}_{[i]} \leftarrow \text{GroupNorm}(\mathbf{O}^{\text{raw}}) \otimes \text{SiLU}(W_g\mathbf{X}_{[i]})$ return $\mathbf{O}_{[i]}$, $\mathbf{S}_i$
```

### D.4.2 Empirical Performance and Trade-offs

Gated DeltaNet has been integrated into Alibaba's Qwen3-Next and Qwen3.5 production models. Empirically:

| Task | Gated DeltaNet | Mamba2 | DeltaNet | GLA |
|---|---|---|---|---|
| Language Modeling | **1.0**× | 1.08× | 1.05× | 1.12× |
| Associative Recall | **1.0**× | $--(noperfect)$ | **1.0**× | 0.75 |
| Length Extrapolation | **1.0**× | 0.98× | 0.96× | 0.94× |
| Throughput (tokens/sec) | 0.85× | **1.0**× | 0.82× | 0.88× |

Gated DeltaNet achieves the best balance across diverse tasks at the cost of slightly reduced throughput due to richer transition matrices.

## D.5 4. HGRN2: Hierarchical Gating with Outer-Product Expansion

### D.5.1 Mathematical Foundation

HGRN2 [28] uses an outer-product-based state expansion with **hierarchically lower-bounded forget gates**. The state update is:

$$\mathbf{S}_t = \text{diag}(\mathbf{g}_t) \cdot \mathbf{S}_{t-1} + \mathbf{v}_t\mathbf{k}_t^\top, \quad \mathbf{o}_t = \mathbf{S}_t\mathbf{q}_t$$

where $\mathbf{g}_t \in [b_\ell, 1]^d$ is a lower-bounded forget gate for layer $\ell$. The bounds satisfy:

$$0 \le b_{\ell_{\text{shallow}}} < b_{\ell_{\text{middle}}} < b_{\ell_{\text{deep}}} \le 1$$

i.e., bounds increase (become less restrictive) in the deeper layers. The gate itself is computed as:

$$\mathbf{g}_t = b_\ell + (1 - b_\ell) \cdot \sigma(W_g\mathbf{x}_t)$$

where $\sigma$ is sigmoid. When $W_g$ outputs weakly negative logits, $\mathbf{g}_t \approx b_\ell$ (forced retention at shallow layers). When outputs are strongly positive, $\mathbf{g}_t \approx 1$ (memory refresh at any layer).

The hierarchical structure encourages different layers to specialize to different timescales: shallow layers model local dependencies (high retention due to binding $b_\ell$), while deeper layers capture long-range structure (flexible gating with high upper bound).

**Algorithm 34** HGRN2 Forward with Hierarchical Bounds

```
Require: Input $\mathbf{x}_t$; prior state $\mathbf{S}_{t-1}$; layer index $\ell \in [0, L-1]$
Require: Non-decreasing bounds: $0 \le b_0 < b_1 < \cdots < b_{L-1} \le 1$
Ensure: Output $\mathbf{o}_t$ and state $\mathbf{S}_t$
1: Layer-specific retain bound: $b \leftarrow b_\ell$
2: Project: $\mathbf{q}_t \leftarrow W_q\mathbf{x}_t$, $\mathbf{k}_t \leftarrow W_k\mathbf{x}_t$, $\mathbf{v}_t \leftarrow W_v\mathbf{x}_t$
3: Compute forget gate with binding: $\mathbf{g}_t \leftarrow b + (1-b)\cdot\sigma(W_g\mathbf{x}_t)$    ▷ Constrained to $[b, 1]$
4: Diagonal multiplication (vectorized): $\mathbf{S}_t \leftarrow \mathbf{S}_{t-1} \circ \mathrm{diag}(\mathbf{g}_t) + \mathbf{v}_t\mathbf{k}_t^\top$
5: Retrieve: $\mathbf{o}_t \leftarrow \mathbf{S}_t\mathbf{q}_t$
6: Optional normalization: $\mathbf{o}_t \leftarrow \mathrm{RMSNorm}(\mathbf{o}_t)$ return $\mathbf{o}_t$, $\mathbf{S}_t$
```

### D.5.2 Scaling and Empirical Results

At 3B scale on 100B tokens, HGRN2 slightly outperforms Mamba2 and LLaMA-architecture Transformers on language modeling. The hierarchical gating provides multi-scale temporal modeling without explicit layer-wise architectural variants. However, vector-valued diagonal gating (as opposed to element-wise selection) provides less per-item flexibility than delta-rule variants.

## D.6 5. Forgetting Transformer (FoX): Gating in Softmax Logit Space

### D.6.1 Mathematical Foundation

Forgetting Transformer (FoX), proposed by Lin et al. [19], embeds a forget gate directly into softmax attention logits. Rather than replacing softmax with linear recurrence, FoX preserves full softmax expressiveness while adding recency control through a data-dependent logit bias.

For each token position $t$ and all keys $j \le t$, compute a scalar forget gate:

$$f_t = \sigma(w_f^\top \mathbf{x}_t + b_f)$$

The logit bias at position $(i, j)$ is the cumulative log-forget product:

$$d_{ij} = \sum_{l=j+1}^{i} \log f_l = \log \prod_{l=j+1}^{i} f_l$$

The full attention output becomes:

$$\mathbf{O} = \mathrm{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}} + \mathbf{D}\right)\mathbf{V}$$

where $\mathbf{D}_{ij} = d_{ij}$. Equivalently:

$$\mathrm{Attention}(i, j) = \frac{\exp(\mathbf{q}_i^\top \mathbf{k}_j/\sqrt{d_k} + d_{ij})}{\sum_{j'=1}^{i} \exp(\mathbf{q}_i^\top \mathbf{k}_{j'}/\sqrt{d_k} + d_{ij'})}$$

This weighting down-weights older tokens multiplicatively: a token from 10 steps ago is scaled by $\prod_{l=j+1}^{i} f_l$, which is typically much less than 1 if $f_t < 1$ on average.

The mechanism is mathematically equivalent to a **data-dependent learnable variant of ALiBi (Attention with Linear Biases)**, where earlier work used fixed $d_{ij} = -\infty \cdot |i - j|$ or $d_{ij} = -(i - j)$.

**Algorithm 35** FoX: Forgetting Attention with Recency Bias

```
Require: Queries $\mathbf{Q}$, keys $\mathbf{K}$, values $\mathbf{V}$; input $\mathbf{X}$
Require: Forget gate parameters: $w_f \in \mathbb{R}^d$, $b_f \in \mathbb{R}$
Ensure: Output attention $\mathbf{O}$
 1: Compute forget gates:
 2: for $t = 1$ to $T$ do
 3:     $f_t \leftarrow \sigma(w_f^\top \mathbf{x}_t + b_f)$        ▷ Per-token forget probability
 4: end for
 5: Compute cumulative log-forget biases:
 6: for $i = 1$ to $T$ do
 7:     for $j = 1$ to $i$ do
 8:         $d_{ij} \leftarrow \sum_{l=j+1}^{i} \log f_l$        ▷ Cumulative product in log space
 9:     end for
10: end for
11: Standard SDPA with bias:
12: Compute scores: $\mathbf{S} \leftarrow \mathbf{QK}^\top / \sqrt{d_k}$
13: Add bias: $\mathbf{S} \leftarrow \mathbf{S} + \mathbf{D}$
14: Apply softmax: $\mathbf{A} \leftarrow \mathrm{softmax}(\mathbf{S}, \mathrm{dim} = -1)$
15: Weight values: $\mathbf{O} \leftarrow \mathbf{AV}$ return $\mathbf{O}$
```

### D.6.2 Integration with FlashAttention

The key advantage of FoX is compatibility with FlashAttention and existing optimized implementations. The forget biases are added in the softmax logit computation, which is already a core operation in FlashAttention's block-wise algorithm. The overhead is:

$$\text{Overhead} \approx 0.5 - -2\% \text{ wall-clock time}$$

since the bias addition is amortized across matmul operations.

### D.6.3 Strengths and Limitations

FoX achieves state-of-the-art results on length extrapolation, near-perfect needle-in-haystack retrieval, and superior long-context understanding compared to Transformers. However, it remains $\mathcal{O}(L^2 d)$ at training and $\mathcal{O}(Ld)$ per step at inference with KV cache, limiting extreme sequence lengths.

## D.7 6. Gated Attention (Post-SDPA Sigmoid Gating)

### D.7.1 Mathematical Foundation

The NeurIPS 2025 Best Paper by the Qwen team proposes applying a sigmoid gate **after** scaled dot-product attention (SDPA):

$$\mathbf{Y} = \mathrm{SDPA}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{softmax}\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)\mathbf{V}$$

$$\mathbf{Y}' = \mathbf{Y} \odot \sigma\left(\mathbf{XW}_\theta\right)$$

where the gate score $\sigma(\mathbf{XW}_\theta)$ can have shape $\mathbb{R}^{n \times q \times d_k}$ (element-wise gating per query head) or $\mathbb{R}^{n \times q}$ (head-wise fixed gating). Across 30+ variants and 15B MoE + 1.7B dense models trained on 3.5T tokens, the team found:

- Headwise gating adds only $\sim 1.6M$ parameters to a 15B model (negligible overhead).

- Effective gates are **sparse**: mean activation $\approx 0.116$ (i.e., 88% are zero or near-zero).
- Gates act as query-dependent filters, suppressing low-value channels while preserving high-value signal.
- Gates demonstrably eliminate the **attention sink** phenomenon, where the attention pattern becomes dominated by a single early token.

**Algorithm 36** Gated Softmax Attention (Post-SDPA)

```
Require: Queries $\mathbf{Q}$, Keys $\mathbf{K}$, Values $\mathbf{V}$; input $\mathbf{X}$
Require: Gate projection matrix $W_g \in \mathbb{R}^{d \to d_{\text{gate}}}$ where $d_{\text{gate}} \in \{1, d_k\}$
Ensure: Gated output $\mathbf{Y}'$
 1: Standard SDPA:
 2: Compute attention: $\mathbf{A} \leftarrow \text{softmax}(\mathbf{Q}\mathbf{K}^\top/\sqrt{d_k})$
 3: Weight values: $\mathbf{Y} \leftarrow \mathbf{A}\mathbf{V}$
 4: Compute gates:
 5: Project input: $\mathbf{G} \leftarrow \mathbf{X}W_g$                    ▷ shape: $[n, q, d_{\text{gate}}]$
 6: Apply sigmoid: $\mathbf{G} \leftarrow \sigma(\mathbf{G})$               ▷ Element-wise gating
 7: Apply gating:
 8: if $d_{\text{gate}} = 1$ then
 9:     Broadcast and scale: $\mathbf{Y}' \leftarrow \mathbf{Y} \cdot \mathbf{G}$      ▷ Head-wise scaling
10: else                                                   ▷ $d_{\text{gate}} = d_k$
11:     Element-wise modulation: $\mathbf{Y}' \leftarrow \mathbf{Y} \odot \mathbf{G}$   ▷ Per-dimension gating
12: end ifreturn $\mathbf{Y}'$
```

### D.7.2 Key Findings

- **Attention-sink suppression**: Gating breaks the feedback loop where softmax concentrates on the first token, enabling more balanced attention.
- **Training stability**: Models with gating tolerate larger learning rates (+30% higher stable $\eta_{max}$).
- **Minimal overhead**: Latency impact is only 1.6% on H100 GPUs; throughput drops at most $2-3\%$.
- **Scaling law improvement**: Improves loss across model scales from 800M to 110B parameters consistently.

## D.8 Comparative Analysis: Taxonomy of Gated Mechanisms

| Architecture | Publication Year | State Representation | Gate Type | Training Complexity | Inference Complexity | Ref |
|---|---|---|---|---|---|---|
| GLA | 2023 | Matrix (recurrent) | Diagonal decay | $\mathcal{O}(Ld^2)$ | $\mathcal{O}(d^2)$ memory | [44] |
| DeltaNet | 2024 | Matrix (recurrent) | Scalar write ($\beta$) | $\mathcal{O}(Ld^2)$ | $\mathcal{O}(d^2)$ memory | [45] |
| Gated DeltaNet | 2024 | Matrix (recurrent) | Decay + write | $\mathcal{O}(Ld^2)$ | $\mathcal{O}(d^2)$ memory | [46] |
| RetNet | 2023 | Matrix (recurrent) | Fixed exponential | $\mathcal{O}(Ld^2)$ | $\mathcal{O}(d)$ per-step | [36] |

| Architecture | Publication Year | State Representation | Gate Type | Training Complexity | Inference Complexity | Ref |
|---|---|---|---|---|---|---|
| HGRN2 | 2024 | Matrix (recurrent) | Diagonal (bounded) | $\mathcal{O}(Ld^2)$ | $\mathcal{O}(d^2)$ memory | [28] |
| FoX | 2025 | Full attention | Logit-space bias | $\mathcal{O}(L^2d)$ | $\mathcal{O}(L)$ KV cache | [19] |
| Gated Softmax | 2025 | Full attention | Post-SDPA sigmoid | $\mathcal{O}(L^2d)$ | $\mathcal{O}(L)$ KV cache | [29] |

| Architecture | Trainable | In-Context Recall | Associative Recall | Length Extrapolation | Key Advantage |
|---|---|---|---|---|---|
| GLA | Yes | Moderate | Weak | Good (20K) | Hardware efficiency; chunkwise parallelism |
| DeltaNet | Yes | Strong | Perfect (MQAR) | Good | Error-correcting semantics; strong retrieval |
| Gated DeltaNet | Yes | Very strong | Perfect | Excellent | Balanced: forgetting + targeted updates |
| RetNet | Yes | Weak | Weak | Good | Simplicity; no KV cache needed |
| HGRN2 | Yes | Moderate | Weak | Moderate | Multi-scale temporal modeling |
| FoX | Yes | Very strong | Very strong (near-perfect) | Excellent | Preserve softmax; minimal overhead |
| Gated Softmax | Yes | Strong | Good | Good | Simplicity; no replacement of SDPA |

| Architecture | Typical Throughput | Memory per-Inference | Primary Use Case |
|---|---|---|---|
| GLA | High (chunkwise CUDA) | $O(d^2)$ constant | Long-context with memory constraints |
| DeltaNet | Medium-High | $O(d^2)$ constant | Retrieval and associative reasoning |
| Gated DeltaNet | Medium | $O(d^2)$ constant | Production: Qwen3-Next, balanced performance |
| RetNet | High | $O(d)$ per-step | Extremely fast inference; simple models |
| HGRN2 | Medium-High | $O(d^2)$ constant | Multi-scale temporal patterns |
| FoX | Moderate (quadratic) | $O(Ld)$ KV cache | Length extrapolation; long-context generation |

| Architecture | Typical Throughput | Memory per-Inference | Primary Use Case |
|---|---|---|---|
| Gated Softmax | High (minimal overhead) | $O(Ld)$ KV cache | Drop-in improvement for existing models |

## D.9 Unified Gating Principle

All seven architectures share a common principle: **gating is a mechanism to control information flow and selective memory retention**. The specific design choices diverge along several dimensions:

1. **Location of gating**: Recurrent state updates (GLA, DeltaNet, Gated DeltaNet, HGRN2) versus softmax logit/output space (FoX, Gated Softmax, RetNet).
2. **Granularity of control**: Dimension-wise (GLA, HGRN2 diagonal), key-specific (DeltaNet), token-wise (FoX), or head-wise (Gated Softmax).
3. **Data dependence**: Fully data-dependent (GLA, DeltaNet, Gated DeltaNet, FoX, Gated Softmax) versus fixed schedules (RetNet with exponential decay).
4. **Expressiveness trade-off**: Linear recurrent efficiency (GLA, DeltaNet, Gated DeltaNet, HGRN2, RetNet) versus full softmax expressiveness (FoX, Gated Softmax).

## D.10 Implementation and Practical Guidance

### D.10.1 When to Use Each Architecture

1. **GLA**: Fast inference with moderate-to-long contexts (2K–20K tokens), when custom CUDA kernels are acceptable, and retrieval performance is not critical.
2. **DeltaNet**: Strong associative recall and in-context learning tasks; good for synthetic tasks (MQAR) and key-value association testing.
3. **Gated DeltaNet**: Production models requiring the best balance of recall, forgetting, and throughput (proven in Qwen3-Next; ICLR 2025).
4. **RetNet**: Extremely efficient inference without KV caching; suitable for edge/mobile deployments or toy models.
5. **HGRN2**: Multi-scale temporal hierarchies; when layer-specific forget bounds are desirable.
6. **FoX**: Length extrapolation and long-context understanding; when quadratic training is acceptable and replacing softmax is not desired.
7. **Gated Softmax**: Quick improvement to existing softmax models with minimal code changes; best for practitioners wanting immediate gains without major refactoring.

### D.10.2 Hardware Considerations

| Hardware Target | Recommended Architectures | Notes |
|---|---|---|
| GPU (H100/A100) | Gated Softmax, FoX, Gated DeltaNet | Mature softmax/linear implementations. GLA/DeltaNet require custom kernels. |
| TPU (high memory) | GLA, DeltaNet, Gated DeltaNet, HGRN2 | Hardware supports efficient matmuls; linear recurrent methods shine. |
| Mobile/Edge | RetNet, lightweight Gated Softmax | Constant-memory inference critical. |

## E Annex E: Conceptual Introduction—Transformers and Attention for Beginners

This appendix provides an accessible introduction to the fundamental concepts of transformers and the attention mechanism, aimed at readers without deep prior knowledge in deep learning. Transformers are the engine behind modern artificial intelligence systems like ChatGPT, but their operation can be understood through real-world analogies.

### E.1 What is a Transformer?

A transformer is a neural architecture that processes text or data sequences by interpreting the meaning of each word while considering all other words in context. Imagine you read a sentence:

*"The bank was full of people waiting. Mary went to the bank."*

What does "bank" mean in each case? In the first sentence, it refers to a financial institution (or a bench). In the second, it is less clear without context. A transformer solves this automatically by looking at all surrounding words. In the second sentence, seeing "Mary went", the model better understands that it probably is a riverbank (where she goes to swim) or a financial institution (where she goes to conduct a transaction).

This ability to understand the complete meaning of a word by considering the entire sentence is called *contextualization*, and it is the heart of how modern transformers work.

### E.2 The Attention Mechanism: An Analogy

The attention mechanism is the "magic trick" that allows transformers to understand context. Think of it as participating in an important conversation in a noisy room:

1. **Lots of noise**: Someone is speaking and being very important to you.
2. **You ignore the rest**: Your brain automatically focuses attention on that person, ignoring other sounds.
3. **You understand better**: By concentrating, you catch every word clearly.

The neural attention mechanism works the same way: each word "asks" how important every other word is to understanding its meaning, then "focuses" its attention on the most relevant ones.

## E.3 Practical Example Step-by-Step

Let's see how a transformer understands the word "eats" in two different contexts:

### E.3.1 Context 1: "The cat eats fish"

The transformer, when processing "eats", internally asks:

- How relevant is "The"? Little (it's just an article). **Low attention**.
- How relevant is "cat"? Very relevant (it's the subject, who eats). **High attention**.
- How relevant is "fish"? Very relevant (it's what is eaten). **High attention**.

Thus, the mental representation of "eats" focuses primarily on "cat" and "fish".

### E.3.2 Context 2: "The restaurant eats into profit margins"

Here the transformer asks in a different context:

- How relevant is "restaurant"? Very relevant (it's the subject). **High attention**.
- How relevant is "profit"? Very relevant (it's affected). **High attention**.
- How relevant is "margins"? Very relevant (it's what's being consumed). **High attention**.

The word "eats" gets a completely different representation because it "attends" to different words in different contexts.

## E.4 How Attention Works Mathematically (Simple Version)

Behind the attention shift is mathematics. Here is the simplified version without too much technical detail:

### E.4.1 Step 1: Queries, Keys, and Values

Each word in the sentence is transformed into three versions:

- **Query**: "What information do I need to know?"
- **Key**: "What information do I have?"
- **Value**: "Here is my important information."

Imagine in a library, each visitor is a "query", each book is a "key", and the content is the "value". The librarian (attention) matches queries with the most relevant keys to access the correct value.

### E.4.2 Step 2: Compatibility

The system examines how *compatible* each query is with each key. Queries similar to keys get a *high* compatibility score. This is calculated by multiplying the query by the key (mathematically, the dot product).

### E.4.3 Step 3: Focus

The compatibility scores are converted into "focus weights". High compatibilities mean "focus attention here", and low ones mean "ignore this". Mathematically, a function called `softmax` converts scores into percentages (like: 40% attention here, 35% here, 25% there).

#### E.4.4 Step 4: Combine

Finally, the system combines all values, weighted by the focus weights. If a word has 80% attention on the subject, 80% of the subject's information gets blended into that word's representation.

## E.5 Multiple Attention Heads: Multiple Perspectives

A key feature is that transformers do not use a single attention but *multiple attention heads* in parallel. It's as if you had 8 people analyzing the sentence *simultaneously*, each paying attention to different aspects:

- **Person 1**: "I focus on subject-verb relationships."
- **Person 2**: "I search for the direct object."
- **Person 3**: "I track information about verb tense."
- **Person 4**: "I search for modifiers and adjectives."

Each "head" learns to focus on different language patterns. Together, they capture much richer understanding than a single head.

## E.6 Stacking Layers

Transformers process information in *multiple layers* (usually 12 to 48 for practical models). Imagine editing an essay:

1. **First pass**: You fix spelling and basic grammar.
2. **Second pass**: You improve clarity and sentence structure.
3. **Third pass**: You ensure consistency and narrative flow.

Transformers work the same way: each layer refines text understanding. The first layer captures basic features (simple words, genders), while later layers understand complex concepts (word relationships, abstract meanings).

## E.7 Why Does It Matter?

The attention mechanism was revolutionary because:

1. **Parallelism**: It finds context for *any word with any other word*, without processing them sequentially (unlike earlier systems). This makes it very fast.
2. **Flexibility**: It learns what patterns to look for automatically from data, without you needing to program rules manually.
3. **Scalability**: It works from small texts to contexts with millions of words.
4. **General Capabilities**: The same mechanism works for translation, summarization, question-answering, text generation, computer vision, and more.

## E.8 Practical Challenges

Despite the extraordinary success of transformers, they present challenges:

### E.8.1 Computational Complexity

If a sentence has 1,000 words, the standard attention mechanism must compare each word with all other 1,000 words. That's $1,000 \times 1,000 = 1,000,000$ comparisons. For long documents with millions of words, this becomes prohibitively expensive in time and memory.

### E.8.2 Memory Requirements

Especially during inference (when using trained models), storing the entire attention matrix can consume enormous amounts of RAM or GPU memory, limiting the sequence lengths you can process.

### E.8.3 Device Efficiency

Transformers were designed for powerful TPUs and GPUs. Running them on phones or embedded devices is challenging.

## E.9 Modern Solutions

To solve these challenges, research has proposed many variants:

- **Sparse Attention**: Instead of comparing each word with ALL others, it compares only with a strategic subset (nearby neighbors, periodic patterns). Reduces complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ or $\mathcal{O}(n)$.
- **Recurrent Models**: Like Mamba, they incorporate aspects of old recurrent neural networks but with better modern efficiency.
- **Gated Attention**: Mechanisms that selectively learn what information to pass forward, reducing storage needs.
- **Quantization**: Use smaller numbers (integers instead of decimals) to reduce memory without losing too much precision.

These innovations are what this toolkit (Frankenstein) allows you to experiment with easily.

## E.10 Key Takeaways

- A **transformer** is a neural network that understands language context very well.
- The **attention mechanism** allows each word to "focus" on which other words are relevant to understanding its meaning.
- Attention works by computing **compatibility** between each word (query) and all others (keys), then combining information based on that compatibility (values).
- **Multiple heads** of attention explore different patterns in parallel.
- **Multiple layers** progressively refine understanding.
- The main challenge is that standard attention has quadratic complexity, which is expensive for long texts.
- Many **modern solutions** (sparse attention, recurrent models, quantization) address these challenges, and this toolkit lets you explore all of them.