LEARNABLE GRADIENT ACCUMULATION (LGA)

FORGEN AI
James D. Stevens Jr.
james@forgen.ai

ABSTRACT

Learnable Gradient Accumulation (LGA) introduces a general and interpretable meta-learning framework for memory- and context-aware optimization by parameterizing gradient state information through learnable transformations. By replacing static update rules with trainable memory and context dynamics, LGA enables gradient updates that explicitly encode how past and present information interact. LGA enables, among other things, for example, *grokking* acceleration for transformer architectures, particularly through learning, for a given task, how to treat gradients given a provided context and/or accumulated gradient state. LGA uses a learned gradient transformation function, which may take any of a variety of different forms, including any one or more of an affine or linear manner, gated or simple nonlinear mechanism, or through more complex learning mechanisms such as multilayer perceptrons (MLPs) and recurrent modules, for example. LGA, thus, utilizes enables utilizing past gradient states to learn how to snare out higher-level generalizations more quickly (enabling, *inter alia*, accelerated *grokking*) that are sought to be learned by the model. These gradient transformation parameters and model parameters may be learned through a bi-level training/optimization process.

I INTRODUCTION

Optimization in neural networks has evolved significantly, transitioning from simple stochastic gradient descent (SGD) to a diverse array of adaptive methods. However, most of these techniques do not make the concept of memory in gradient updates an explicit or learnable component. Methods like momentum and Adam utilize fixed memory decay parameters, which may not be ideal across all tasks, data distributions, or training stages.

An approach where memory dynamics are learnable, particularly at the level of gradient transformations, offers new opportunities in generalization, continual adaptation, and model interpretability. Gradient accumulation can be treated not as a static heuristic, but as a trainable mechanism capable of evolving with the task and training context.

Learnable Gradient Accumulation (LGA) is introduced as a versatile and interpretable optimizer primitive.

LGA formalizes accumulation as a learnable process—interpolating between memoryless updates and momentum-based schemes—thereby facilitating learning of task-sensitive behavior and enhanced control over training dynamics.

It generalizes gradient accumulation through a learned gradient transformation function, optionally incorporating spatial and/or temporal recurrence via memory retention. This generalization bridges stateless gradient methods and stateful accumulators, enabling flexible dynamics that adjust during training. LGA is particularly well-suited for domains where static heuristics struggle—such as low-data regimes and task-shifting scenarios.

Although the following sets forth a framework that generally teaches use of an accumulated gradient across batches, the framework (including each of the foregoing archetypes, variants, and exemplary instantiations) is equally applicable to only the current gradient where, *e.g.*, gradients are zeroed and not maintained or accumulated across batches.

II LGA FRAMEWORK

To ground this concept, we begin by formalizing the structure and mechanics of LGA. The framework centers on a learnable update rule that governs how gradient information is accumulated over time. This rule distinguishes between memory derived from prior steps and incoming signal from the current gradient, allowing each to be independently transformed and composed. The resulting formulation generalizes classical optimizers and creates a foundation for a family of adaptive and interpretable accumulation strategies.

A Framework Definition –

The General Archetype of LGA refers to the class of update rules that operate exclusively on the accumulated gradient state and the current raw gradient signal, without any additional conditioning on external inputs (*e.g.*, data, task, timestep). This variant captures the core intuition of LGA: that the process of accumulating gradients across steps can itself be parameterized and learned.

1 Gradient Accumulation, Generally –

In traditional optimizers such as SGD, Adam, or RMSProp, each parameter update depends directly on the current gradient $\mathbf{g_t}$, either in raw form or as part of a momentum-like moving average. In contrast, the LGA framework centers optimization around a persistent accumulated gradient memory, denoted $\mathbf{g_{acc}}$, which integrates filtered versions of past gradients over space and/or time.

In this context, the raw gradient $\mathbf{g_t}$ serves only as a transient signal. Once processed—*e.g.*, via spatial filtering (across dimensions) or temporal filtering (across steps)—it is integrated into the memory buffer. Only the accumulated state $\mathbf{g}_{\mathbf{acc}}^{(t_u)}$ is used to determine the parameter update.

Parameter updates occur at discrete batch steps $t_u \in \mathcal{U} \subseteq \mathbb{N}$, where $\mathcal{U} = \{t_0, t_1, t_2, \dots\}$ is the update schedule. This schedule may follow a fixed interval (*e.g.*, every k_u steps) or be dynamically determined (*e.g.*, via validation loss, training phase, or learned triggers).

The parameter update rule for a given parameter φ at update index u is:

$$\varphi^{(u+1)} = \varphi^{(u)} - \eta \cdot \mathbf{g}_{\mathbf{acc}}^{(t_u)}$$

where:

- $\varphi^{(u)}$ is the model parameter after update step u;
- $\varphi^{(u+1)}$ is the model parameter after update step $u + 1$;
- η is the learning rate; and
- $\mathbf{g}_{\mathbf{acc}}^{(t_u)}$ is the state of the accumulated gradient at step t_u, incorporating all prior transformed gradients.

The $\mathbf{g}_{\mathbf{acc}}^{(t_u)}$ is the current memory state used at the time of the parameter update. In the case where updates occur every step (*i.e.*, $t_u = u$), this reduces to standard per-step optimization.

At each update step u, occurring at batch index t_u, the change in model parameters satisfies:

$$\mathbf{g}_{\mathbf{acc}}^{(t_u)} = -\frac{\Delta\varphi}{\eta}, \text{ where } \Delta\varphi = \varphi^{(u+1)} - \varphi^{(u)}$$

This expression highlights that LGA determines both the direction and magnitude of updates through the learned, memory-driven gradient state, rather than through instantaneous gradients. The optimizer's behavior is thus shaped by a history of filtered gradient information — enabling structured, long-range, and interpretable learning dynamics.

2 LGA Update Step –

The LGA update is defined as:

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow \mathcal{F}\left(\mathbf{g}_{\mathbf{acc}}^{(t)}, \mathbf{g_t}; \theta\right)$$

where:

$\mathbf{g}_{\mathbf{acc}}^{(t)} \in \mathbb{R}^d$ is the current accumulated gradient memory before any projection and in its raw form, with the superscripts indicating its associated timestep;

$\mathbf{g}_t \in \mathbb{R}^d$ is the current gradient at time t; and

$\mathcal{F}$: $\mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^{d'}$, with $d' \leq d$, is referred to as the "Learnable Gradient Accumulation Function", and is parameterized by θ.

The Learnable Gradient Accumulation Function $\mathcal{F}$ is applied to the accumulated gradient $\mathbf{g}_{\mathbf{acc}}$ and/or the present gradient $\mathbf{g}_{\mathbf{t}}$ in determining the gradient accumulation information $\mathbf{g}_{\mathbf{acc}}^{(t+1)}$ used in the next iteration.

The LGA framework is considered in two main components: a memory component, corresponding to aggregated gradient information at the present step; and a signal component, corresponding to present gradient information. Thus, the LGA

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow \mathcal{F}_{\text{mem}}(\mathbf{g}_{\mathbf{acc}}^{(t)}; \theta_{\text{mem}}) + \mathcal{F}_{\text{sig}}(\mathbf{g}_{\mathbf{t}}; \theta_{\text{sig}})$$

where:

$\mathcal{F}_{\text{mem}}$: $\mathbb{R}^d \to \mathbb{R}^{d'}$, with $d' \leq d$, and is referred to as the "Memory Dynamics Function" and is a learnable transformation applied to the accumulated gradient $\mathbf{g}_{\mathbf{acc}}$;

$\mathcal{F}_{\text{sig}}$: $\mathbb{R}^d \to \mathbb{R}^{d'}$, with $d' \leq d$, and is referred to as the "Signal Injection Function" and is a learnable transformation applied to the current gradient $\mathbf{g}_{\mathbf{t}}$;

d' denotes the output dimension, typically equal to or less than d depending on projection or compression;

θ_{mem} represents learnable parameter(s) for the Memory Dynamics Function; and

θ_{sig} represents learnable parameter(s) for the Signal Injection Function.

3 Projected Memory Representations

To enable interpretability or structured memory access, the accumulated gradient may be projected into an internal memory representation:

$$\boldsymbol{v}_t = \mathbf{P}^{-1}\mathbf{g}_{\mathbf{acc}}^{(t)}$$

where:

$\mathbf{P}^{-1} \in \mathbb{R}^{d' \times d'}$ is a learned or predefined projection matrix;

$\boldsymbol{v_t} \in \mathbb{R}^{d'}$ is a projected or reparameterized memory vector used for internal computation.

This transformation allows the memory to be accessed or controlled in a lower-dimensional or structured space, which may be more interpretable or computationally efficient. That is, $\boldsymbol{v_t}$ represents an accumulated memory vector or state at step t, constituting a transformed internal representation of the memory, such that $\boldsymbol{v_t} = \mathbf{P}^{-1}\mathbf{g}_{\mathbf{acc}}^{(t)}$, such that $\boldsymbol{v_t}$ is derived from $\mathbf{g}_{\mathbf{acc}}$ via the inverse of some projection matrix $\mathbf{P}$, where $\mathbf{P}$ is a learned or predefined projection matrix. $\mathbf{P}$ is useful for transforming $\mathbf{g}_{\mathbf{acc}}$ into an interpretable or decodable internal representation.

In the forward flow, a new $\mathbf{g}_{\mathbf{acc}}$ is computed based on updates involving $\boldsymbol{v_t}$ (and possibly new gradients or inputs). The accumulated gradient $\mathbf{g}_{\mathbf{acc}}^{(t)}$ serves as a memory trace, capturing past update signals. To compute the memory's influence at time t, it is transformed via the projection mechanism into $\boldsymbol{v_t}$. The resulting vector $\boldsymbol{v_t}$ is then used to determine updates to the accumulated gradient, typically through a function of the form: $\Delta\mathbf{g} = f(\boldsymbol{v_t}, \cdot)$, possibly including new gradient signals, input context, or gating mechanisms. Crucially, this update can include both additive and multiplicative components, allowing for controlled scaling or forgetting:

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow \gamma_t \cdot \mathbf{g}_{\mathbf{acc}}^{(t)} + \Delta\mathbf{g}$$

where $\gamma_t \in [0,1]$ is a learnable or input-conditioned retention factor. This architecture supports non-linear, context-aware memory dynamics while remaining compatible with projected or compressed gradient representations.

B LGA Archetypes –

The general LGA framework admits multiple design variants, referred to as archtypes, depending on how the gradient update functions are conditioned. These archetypes define families of LGA implementations that differ in whether and how they utilize external context—such as model state, training dynamics, or direct input embeddings—to modulate memory and signal transformations. Each archetype retains the core structure of LGA but specializes its behavior based on the source and nature of contextual signals, enabling a range of capabilities from interpretable static updates to dynamic, task-sensitive optimization schemes. The following subsections detail three key archetypes: Context-Agnostic LGA, Context-Aware LGA, and Attentive LGA.

1 Context-Agnostic LGA

The present framework is the same as described above under the Framework Definition in Section A, and without direct use of the input or any embedding of it. All gradient signals are based solely on the model's internal computations. In this sense, there is no direct input context being provided, as the gradient information

is based on only the accumulated gradients and the current gradient derived from the model output. The Context-Agnostic LGA, thus, uses both a context-agnostic memory transformation and a context-agnostic signal transformation, and may be represented as $\mathcal{F}$:

$$\mathcal{F}\left(\mathbf{g}_{\mathbf{acc}}^{(t)}, \mathbf{g}_{\mathbf{t}}; \theta\right) = \mathcal{F}_{\text{mem}}(\mathbf{g}_{\mathbf{acc}}^{(t)}; \theta_{\text{mem}}) + \mathcal{F}_{\text{sig}}(\mathbf{g}_{\mathbf{t}}; \theta_{\text{sig}})$$

where:

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow \mathcal{F}\left(\mathbf{g}_{\mathbf{acc}}^{(t)}, \mathbf{g}_{\mathbf{t}}; \theta\right)$$

2 Context-Aware LGA (ALGA)

The Context-Aware LGA archetype introduces a structured extension to the General LGA by enabling modulation of gradient accumulation behavior in response to training-state-dependent cues, without reference to the input data itself. The present archetype is a variant in which both memory and signal transformations are explicitly conditioned on external context, such as conditioned on external context such as task identifiers, model phase indicators, or training statistics—not derived from the input x_t or its embedding. This enables dynamic modulation of gradient accumulation behavior in response to input-dependent cues or higher-level model state, with the main difference being this attentive or otherwise context-aware version.

The Context-Aware LGA framework is, thus, generally expressible as:

$$\mathcal{F}\left(\mathbf{g}_{\mathbf{acc}}^{(t)}, \mathbf{g}_{\mathbf{t}}; \theta, \mathbf{c}_{\mathbf{t}}\right) = \mathcal{F}_{\text{mem}}(\mathbf{g}_{\mathbf{acc}}^{(t)}; \theta_{\text{mem}}, \mathbf{c}_{\mathbf{t}}) + \mathcal{F}_{\text{sig}}(\mathbf{g}_{\mathbf{t}}; \theta_{\text{sig}}, \mathbf{c}_{\mathbf{t}})$$

where:

$\mathbf{c}_{\mathbf{t}}$: $\in \mathbb{R}^k$, and
is a context vector not derived from input data $\mathbf{x}_{\mathbf{t}}$, but from training dynamics or structural metadata.

It is noted that the dimension k generally represents an architecture dependent / selected dimensionality, and may or may not be less than or equal to than d depending on design choice / use case. This formulation enables the LGA mechanism to adapt to global training conditions or architecture-specific states while maintaining separation from direct input influence. As such, Context-Aware LGA strikes a balance between adaptability and modularity, offering a rich design space for training-aware memory systems.

These signals, denoted $\mathbf{c}_{\mathbf{t}}$, are derived not from the current input $\mathbf{x}_{\mathbf{t}}$ or its embedding, but from broader meta-information available during training or model execution. This includes features such as:

- Layer-wise or parameter-wise gradient norms,
- Task identifiers in multi-task or continual learning settings,
- Training phase encodings (*e.g.*, early-stage vs. late-stage training),
- Optimization statistics (*e.g.*, momentum magnitudes, variance), and
- Learned schedules or hypernetwork outputs.

These contextual cues guide the adaptation of both memory and signal transformations without referencing the input or any embedding of it. This allows for dynamic yet input-agnostic control of gradient accumulation behavior.

3 Attentive LGA Archetype –

The Attentive Learnable Gradient Accumulation (LGA) archetype is a specialization of the general LGA family where update dynamics are explicitly modulated by the current input or its embedding. This approach incorporates attention-style conditioning, enabling the system to dynamically tailor memory and signal updates based on real-time, input-specific information.

The Attentive LGA update rule is generally stated as:

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow \mathcal{F}_{\text{mem}}\left(\mathbf{g}_{\mathbf{acc}}^{(t)}; \theta_{\text{mem}}, \psi(\mathbf{x_t})\right) + \mathcal{F}_{\text{sig}}\left(\mathbf{g_t}; \theta_{\text{sig}}, \psi(\mathbf{x_t})\right)$$

where:

$\mathbf{x_t}$ is the current input; and

$\psi(\mathbf{x_t})$: $\in \mathbb{R}^k$, with k < d, and represents a learned embedding derived from a context encoder (*e.g.*, an MLP, attention module, or transformer encoder) based on the current input $\mathbf{x_t}$.

This design empowers the model to selectively retain, scale, or gate information in a data-aware manner, modulating updates according to characteristics of each input—such as its complexity, uncertainty, or semantic relevance. Because the input embedding can be generated by an attention mechanism or projection-based encoder, the model inherits both flexibility and responsiveness from architectures like transformers or adaptive optimizers.

By allowing the learning dynamics to be guided directly by the current input, Attentive LGA offers a powerful and expressive mechanism for tightly coupled data-driven control over gradient accumulation behavior.

III LGA INSTANTIATIONS

This section presents concrete implementations of the LGA framework, each specifying particular forms for the memory update function $\mathcal{F}_{\text{mem}}$ and the signal injection function $\mathcal{F}_{\text{sig}}$. These instantiations differ in how they accumulate, transform, or adapt to gradient signals over time. Some operate under the unconditioned General Archetype, while others incorporate input- or context-awareness, falling under the Attentive LGA (AGLA) variant.

A Affine Accumulation

A minimal yet expressive form is provided where both memory and signal transformations are affine. The affine accumulator is the simplest instantiation of the LGA framework and serves as a strong, interpretable baseline. It updates the accumulated gradient using a weighted sum of the previous accumulated state and the current gradient signal, along with an optional additive bias:

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow \lambda \times \mathbf{g}_{\mathbf{acc}}^{(t)} + a \times \mathbf{g}_{\mathbf{t}} + b$$

where:

- λ represents decay of the memory;
- a represents a current gradient scaling factor; and
- b represents an offset bias.

Each of λ, a, b is (or at least may be) a learnable scalar or tensor parameter per $\mathbf{g}_{\mathbf{t}}$ or may be a per-parameter learnable scalar or tensor. That is, each of the parameters λ, a, b may be defined at multiple levels of granularity:

- **Global scalar values** shared across all dimensions;
- **Per-parameter vectors** (*e.g.*, matching shape of a layer's weights); or
- **Grouped or structured forms** (*e.g.*, per-layer, per-block, or channel-wise)

This accumulator naturally generalizes classical exponential moving average (EMA) updates (when $a = 1, b = 0$) and can subsume momentum-like behaviors when $\lambda \in (0,1)$. Unlike traditional fixed decay rules, all coefficients in the affine accumulator are treated as learnable parameters, potentially constrained via clamping or regularization (*e.g.*, $\lambda \in [0,1]$) to ensure stability and bounded memory retention.

This formulation is purely temporal and operates under the General Archetype: it does not condition on the input, task, or external context. As such, it is particularly useful in scenarios where interpretability, simplicity, and optimization control are preferred over expressivity.

Despite its simplicity, the affine accumulator has significant modeling power, especially when incorporated across multiple layers or components of a deep network. It enables the model to learn how much of the past gradient history to retain and how to transform incoming gradient signals before committing them to parameter updates.

B Gated / Non-Linear Memory

This variant extends the basic LGA structure by incorporating learnable gating mechanisms and nonlinear transformations into the memory update. Inspired by recurrent architectures such as gated recurrent networks (GRUs), this design allows the model to dynamically regulate how much of the previous accumulated gradient is retained versus overwritten at each step, providing a learnable recurrency control parameter. Generally, this variation is formulated as:

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow z_t \odot \mathbf{g}_{\mathbf{acc}}^{(t)} + (1 - z_t) \odot \phi(\mathbf{g_t})$$

where:

- $z_t \in [0,1]^d$ is a gating vector computed from a learnable gating function z using both the current gradient and the memory;
- $\phi(\cdot)$ is a nonlinear transformation applied to the current gradient, and may be a multilayer perceptron (MLP), normalization function, or a projection function; and
- $\odot$ denotes the elementwise (Hadamard) product.

The gate z is typically implemented as:

$$z = \sigma\left(W_z \mathbf{g_t} + U_z \mathbf{g}_{\mathbf{acc}}^{(t)} + b_z\right)$$

where:

- $W_z \in \mathbb{R}^{d \times d}$ is a learnable weight matrix for introducing a current gradient scaling factor;
- $U_z \in \mathbb{R}^{d \times d}$ is a learnable weight matrix for introducing a memory decay factor;
- $b_z \in \mathbb{R}^d$ represents a learnable bias for the gating function; and
- $\sigma(\cdot)$ represents sigmoid activation for gating.

The signal transformation $\phi(\cdot)$ can be:

- A learned MLP;
- A normalization layer (*e.g.*, LayerNorm); or
- A projection into latent space $\mathbb{R}^{d'}$ if compression is used

This gated nonlinear memory variant offers enhanced expressivity compared to affine LGA, as it can model nonlinear, context-sensitive memory behaviors through both gating and signal transformations. The learnable gate z functions as a dynamic controller of memory retention, enabling the system to selectively preserve informative gradient directions while attenuating or discarding noisy or unstable components. The architecture is also compatible with compressed or projected gradient representations: the nonlinear transformation $\phi(\cdot)$, or the gate computations themselves, can operate in a latent space, allowing for scalable memory even in high-dimensional settings.

Attentive Gated LGA (AGLGA)

Attentive Gated LGA extends the gated nonlinear memory model by conditioning memory updates on external inputs or contextual embeddings. Both the nonlinear transformation ϕ and the gate z are modulated by a learned encoder $\psi(\mathbf{x_t})$, which maps the current input (or auxiliary context) into a latent control signal that influences memory dynamics. The update rule takes the form:

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow z_t \odot \mathbf{g}_{\mathbf{acc}}^{(t)} + (1 - z_t) \odot \phi(\mathbf{g_t})$$

This is useful for when either $z_t = z(\psi(\mathbf{x_t}))$ and $\phi = \phi(\,\cdot\,;\, \psi(\mathbf{x_t}))$, enabling the system to selectively retain, suppress, or transform information based on task-relevant signals. This input-conditioned gating mechanism allows AGLA to flexibly adapt memory behavior over time, making it particularly suited for settings where gradients alone are insufficient to guide learning—such as in multi-task, curriculum, or non-stationary environments. By incorporating both historical gradients and real-time contextual modulation, AGLA bridges gradient-based optimization and attention-driven memory control.

Accordingly, in implementations, both z_t and ϕ may be conditioned on external inputs or task- or architecture- specific signals via a learned encoder—denoted–this design extends naturally into the Attentive LGA archetype, where adaptation is guided not only by gradient history but also by input-dependent modulation.

C Recurrent Memory

This variant extends the general notion of gating by incorporating full recurrent neural architectures—specifically, *e.g.*, Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) cells—as the memory update engine. In this formulation, the memory state is no longer a direct interpolation between past memory and current signal, but rather evolves through a dedicated hidden state governed by recurrent dynamics. This design allows the optimizer to capture long-range temporal dependencies in the gradient trajectory, making it particularly suitable for settings where meaningful gradient structure emerges slowly over time, such as grokking, curriculum learning, or continual learning.

1 Definition

Let the optimizer maintain a hidden state vector $h_{t+1} \in \mathbb{R}^n$, where n is the size of the recurrent memory.

$$(h_{t+1}, c_{t+1}) = \text{RNN}(\mathbf{g_t}, h_t, c_t; \theta)$$

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} = \mathcal{W}_o h_{t+1} + b_o$$

where:

- $\text{RNN}(\cdot)$ is a recurrent unit such as an LSTM or GRU;
- h_t is the hidden state at timestep t;
- $c_t \in \mathbb{R}^n$, is the cell state (LSTM-specific);
- $\mathcal{W}_o \in \mathbb{R}^{d \times n}$ is a learnable projection parameter that maps the recurrent hidden state back into the accumulator space; and
- $b_o \in \mathbb{R}^d$ is a learnable projection parameter that maps the recurrent hidden state back into the accumulator space.

This formulation allows the gradient memory to be governed by a rich recurrent state, capturing long-range temporal dependencies and enabling nonlinear integration of the gradient trajectory.

The parameter update at learning step u_t (corresponding to the time index t_u) then follows the standard LGA rule defined above.

2 Comparison to Other Technologies

While the previous introduced gating-based mechanisms inspired by recurrent architectures such as GRUs, the memory dynamics in that variant were implemented through explicit, elementwise gates applied to the accumulated gradient. These mechanisms provide useful control over memory retention but remain relatively shallow in temporal scope, as they do not include an internal state that evolves across steps

Formally, the recurrent memory variant treats the memory component of LGA as a learned recurrent function over gradient inputs, optionally projecting the resulting hidden state into the accumulator space. Like the gating-based model of the previous section, this approach supports rich, nonlinear memory transformations—but with deeper recurrence, internal memory evolution, and increased expressivity.

The use of LSTM or GRU architectures within the LGA framework provides a principled mechanism for learning complex temporal dynamics in gradient behavior. Unlike shallow gating, which typically modulates memory based on only the current and previous gradient, recurrent units can maintain and update a hidden state across many steps, allowing the optimizer to recognize and exploit long-term patterns in gradient evolution. This is particularly advantageous in domains where

important learning signals emerge gradually—such as in delayed supervision, structural induction, or phase-shifted training regimes. Moreover, recurrent mechanisms enable conditional memory retention: gradients that are repeatedly reinforced over time can be retained, while transient or noisy signals can be attenuated. As a result, this variant of LGA offers greater temporal expressivity, improved generalization alignment, and increased robustness in non-stationary or low-signal environments.

One especially compelling use case for recurrent memory in LGA is in tasks where slow generalization signals emerge gradually over training—most notably in the phenomenon of grokking, where models initially memorize before abruptly shifting to generalizable solutions after many iterations. This behavior, frequently observed in transformer-based architectures trained on algorithmic or modular tasks, is driven by low-frequency components in the gradient trajectory that correlate with generalization. These components evolve slowly and are often masked by higher-frequency updates tied to memorization or overfitting. A recurrent LGA variant, particularly one using an LSTM or GRU, can learn to preserve these long-range patterns via its hidden state, amplifying gradients that align with consistent, slowly changing update directions. Recent work by Lee et al. (2024) in the Grokfast method demonstrates that emphasizing such low-frequency signals significantly accelerates generalization in transformer models. By incorporating recurrent mechanisms into the memory dynamics of LGA, the optimizer becomes temporally aware and structurally aligned with these long-term signals—thereby supporting faster convergence and better inductive alignment in slow-learning regimes.

D Residual-Style Injection

Residual-Style Injection applies shaped updates to the gradient accumulator in a residual fashion, promoting stability while allowing gradual integration of new information. The update rule is defined as:

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow \mathbf{g}_{\mathbf{acc}}^{(t)} + \alpha \cdot \mathcal{N}(\mathbf{g}_{\mathbf{t}})$$

where:

- α is a mixing coefficient that can be a fixed schedule or a learned scalar (or vector); and
- $\mathcal{N}$ denotes a shaping operator applied to the raw gradient $\mathbf{g}_{\mathbf{t}}$.

This operator may involve normalization (*e.g.*, LayerNorm, RMSNorm), dropout, or thresholding to suppress noise or enforce scale constraints. This residual-style mechanism encourages smooth updates and can be useful in deep or noisy learning settings where abrupt accumulator changes are undesirable.

E Spatial Gradient Filtering

1 Definition

The signal term is shaped through a learned nonlinear transformation:

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow \lambda \cdot \mathbf{g}_{\mathbf{acc}}^{(t)} + \mathcal{T}(\mathbf{g}_{\mathbf{t}})$$

where:

- λ is a retention factor that may be learned or scheduled, controlling how much of the past memory is preserved analogous to that of the Affine Accumulation variant; and
- $\mathcal{T}$ is a transformation function.

The transformation function $\mathcal{T}$ transforms the incoming gradient $\mathbf{g}_{\mathbf{t}}$ and may take various forms, such as a coordinate-wise gating mechanism, a masked or sparse network, or a soft-thresholded projection that suppresses small or noisy components. This design allows for selective integration of gradient information, enabling more structured or robust memory accumulation than simple averaging or residual addition.

2 Example Transformations

2.1 Coordinate-wise Gates

Each element of $\mathbf{g}_{\mathbf{t}}$ is scaled by a learned or input-dependent gate:

$$\mathcal{T}(\mathbf{g}_{\mathbf{t}}) = \sigma(w) \odot \mathbf{g}_{\mathbf{t}}$$

where $\sigma(w)$ is a sigmoid gate vector. This allows selective emphasis or suppression of gradient dimensions.

2.2 Soft Thresholding / Shrinkage

Enforces sparsity or denoising:

$$\mathcal{T}(\mathbf{g}_{\mathbf{t}}) = \text{sign}(\mathbf{g}_{\mathbf{t}}) \cdot \max(|\mathbf{g}_{\mathbf{t}}| - \boldsymbol{\tau}, 0)$$

where $\boldsymbol{\tau}$ is a learned or scheduled threshold and $\text{sign}(\mathbf{g}_{\mathbf{t}}) \in \{-1,0,1\}^d$ is the sign function over the current gradient $\mathbf{g}_{\mathbf{t}}$.

2.3 Masked Networks

A small MLP or linear layer conditioned on input or task context:

$$\mathcal{T}(\mathbf{g}_{\mathbf{t}}) = \text{MLP}(\mathbf{g}_{\mathbf{t}}; \psi(\mathbf{x}_{\mathbf{t}}))$$

where MLP is a multilayer perceptron and $\psi(\mathbf{x_t})$ injects context into the transformation (blending into AGLA territory if used).

2.4 Low-Rank or Projective Filtering

Project onto a learned or fixed subspace:

$$\mathcal{T}(\mathbf{g_t}) = \boldsymbol{P}^\top \boldsymbol{P}\mathbf{g_t}$$

where $\boldsymbol{P}$ is a low-rank projection matrix. Useful for dimensionality control or filtering irrelevant modes.

2.5 Normalization + Nonlinearity

Apply RMSNorm, LayerNorm, or batchnorm-like operations followed by a nonlinearity:

$$\mathcal{T}(\mathbf{g_t}) = \text{LayerNorm}(\mathbf{g_t}) \cdot \tanh(W\mathbf{g_t})$$

This can stabilize updates and introduce curvature into the update flow.

2.6 Composite or Hybrid Filtering

In practice, multiple transformation mechanisms can be composed within $\mathcal{T}(\mathbf{g_t})$ to leverage the strengths of each. For example:

$$\mathcal{T}(\mathbf{g_t}) = \sigma(w) \odot \text{LayerNorm}(\boldsymbol{P}^\top \boldsymbol{P}\mathbf{g_t})$$

This example combines:

- Projection: $\boldsymbol{P}^\top \boldsymbol{P}\mathbf{g_t}$ for filtering the gradient through a low-rank subspace;
- Normalization: LayerNorm stabilizes the scaled representation;
- Gating: $\sigma(w)$ applies element-wise modulation, possibly learned or context-dependent.

Such hybrid forms enable both structure-aware filtering and dynamic control, making them especially effective in architectures where gradient relevance varies across tasks, layers, or timesteps.

F Temporal Gradient Routing

In addition to filtering gradients across dimensions at a single timestep (spatial filtering), it is equally important to consider the temporal dynamics of gradients across training steps. In tasks that require slow structural discovery — such as algorithmic reasoning, symbolic manipulation, or modular arithmetic — the signal that drives generalization often develops slowly over many iterations. Standard

optimizers, however, treat each gradient independently and lack any memory of which directions may carry long-term value.

To address this, we introduce temporal gradient filtering: a mechanism that emphasizes slowly evolving gradient directions by smoothing gradients over time. This approach builds upon recent findings in the Grokfast method (Lee *et al.*, 2024), which showed that isolating and amplifying the low-frequency components of the gradient trajectory (*i.e.*, the slowly changing parts) can drastically accelerate the grokking process in transformer-based models. These slow components tend to align more strongly with generalization-relevant features of the task, while faster components often correspond to noise or memorization.

Unlike spatial filtering, which modifies the internal structure of the gradient vector at a given step, temporal filtering modifies the input to the signal path by aggregating history — effectively making the optimizer "remember" what gradient directions have been consistently useful. This temporal memory can be implemented through exponential smoothing, windowed averaging, or learnable recurrence mechanisms. Crucially, the output of temporal filtering can be combined with spatial filtering strategies, enabling rich and flexible spatiotemporal control of gradient flow.

1 Definition

In temporal gradient filtering, the incoming gradient $\mathbf{g_t} \in \mathbb{R}^d$ at time t is transformed into a filtered version $\hat{\mathbf{g}}_\mathbf{t} \in \mathbb{R}^d$ that incorporates information from previous steps. A temporally-smoothed component of the gradient is given by:

$$\boldsymbol{\mu}_t = \boldsymbol{\kappa}_t \cdot \boldsymbol{\mu}_{t-1} + (1 - \boldsymbol{\kappa}_t) \cdot \mathbf{g_t}$$

$$\tilde{\mathbf{g}}_\mathbf{t} = \mathbf{g_t} + \boldsymbol{\beta}_\mathrm{t} \cdot \boldsymbol{\mu}_t$$

where:

- $\boldsymbol{\mu}_t$ is the temporally smoothed component of the gradient (also referred to as the slow gradient);
- $\boldsymbol{\kappa}_t$: $\in [0, 1]$, and is the retention factor, which may be fixed, learned, or conditioned on training context;
- $\boldsymbol{\beta}_\mathrm{t}$: $\in \mathbb{R}^d$ is the amplification, which scales the influence of the slow component; and
- $\tilde{\mathbf{g}}_\mathbf{t}$: is the temporally filtered gradient, passed to the signal transformation path of LGA (*i.e.*,

This formulation generalizes the Grokfast smoothing mechanism, with two key extensions:

1. The retention and amplification parameters $\boldsymbol{\kappa}_t$ and $\boldsymbol{\beta}_\mathrm{t}$ may be learned or dynamic, allowing them to adapt during training.

2. The slow component $\boldsymbol{\mu}_t$ can be further transformed or conditioned, allowing integration with the architectural or task-level context.

The resulting filtered signal $\tilde{\mathbf{g}}_\mathbf{t}$ may then be processed by any of the spatial gradient filtering mechanisms introduced in Section III.D (*e.g.*, gating, projection, normalization), enabling composable spatiotemporal filtering.

G Attentive Gradient Routing

1 Definition

Attentive Gradient Routing defines a generalized mechanism for modulating gradient accumulation using contextual signals derived from the input or task environment. Rather than applying uniform transformations to memory and signal terms, AGR uses a learned contextual embedding to condition both memory dynamics and signal injection. This enables input-dependent, data-aware updates that adaptively route gradient flow.

The update rule is defined as follows:

$$\mathbf{g}_{\mathbf{acc}}^{(t+1)} \leftarrow \mathcal{F}_{\text{mem}}\left(\mathbf{g}_{\mathbf{acc}}^{(t)}; \theta_{\text{mem}}, \psi(\mathbf{x_t})\right) + \mathcal{F}_{\text{sig}}\left(\mathbf{g_t}; \theta_{\text{sig}}, \psi(\mathbf{x_t})\right)$$

This structure enables real-time adaptation of gradient flow based on the current input $\mathbf{x_t}$ and its associated context $\psi(\mathbf{x_t})$, promoting input-aware control over memory dynamics. AGR can be viewed as a generalization of Attentive LGA, capable of supporting per-parameter, per-layer, or global modulation depending on how $\psi(\mathbf{x_t})$ is defined and applied.

2 Comparison to Attentive LGA

Attentive Gradient Routing (AGR) can be understood as a specialization within the broader Attentive LGA archetype, but with a distinct emphasis on flexible, fine-grained input-conditioned control over both memory and signal paths. While both mechanisms rely on a contextual encoder $\psi(\mathbf{x_t})$ derived from the current input, their functional orientation and granularity differ in the following ways:

2.1 Structural Modulation Scope:

Attentive LGA typically applies the input context embedding $\psi(\mathbf{x_t})$ to modulate the memory and signal transformations at a global or layer-wise level. It is most often used to influence general retention dynamics or signal shaping uniformly across a parameter group.

AGR, in contrast, is designed for more targeted routing—potentially operating at per-parameter, per-module, or per-pathway levels. This means $\psi(\mathbf{x_t})$ may condition gradient updates differently for different components of the model, enabling heterogeneous routing of gradient flow.

2.3 Functional Role of Input Context Embedding

In ALGA, $\psi(\mathbf{x_t})$ provides data-driven adaptivity primarily by gating or scaling memory/signal transformations.

In AGR, $\psi(\mathbf{x_t})$ may determine how gradients are distributed, transformed, or prioritized across subspaces—functioning more as a routing signal than a simple modulator.

2.4 Interpretation and Application

AGR reframes this as an input-conditioned gradient flow control mechanism, allowing dynamic selection or weighting of update paths. AGR is particularly applicable in architectures with modular or conditional execution (*e.g.*, MoEs, sparsely activated layers).

2.5 Expressivity

AGR generalizes ALGA in terms of control resolution. Whereas ALGA adapts the form of update functions, AGR can adapt both the structure and routing pattern of those updates.

3 Summary

Attentive LGA introduces input-awareness into gradient updates, while Attentive Gradient Routing treats the entire update architecture as dynamically reconfigurable based on the input. AGR subsumes ALGA when the latter's $\psi(\mathbf{x_t})$ modulated transformations are applied uniformly, but offers additional flexibility for architectures demanding conditional execution or gradient specialization.

IV Comparison to Other Gradient Transformation Methods

[Table being finalized – contact for us if you would like the draft in the interim]

V Functional Advantages / Disadvantages of LGA

The Learnable Gradient Accumulation (LGA) framework provides both practical and conceptual advantages across a wide range of optimization scenarios:

- **Task-sensitive memory dynamics**: LGA enables dynamic retention and transformation of gradients, allowing the optimizer to adapt to properties such as gradient scale, curvature, or noise—especially important in non-stationary or multi-task settings.

- **Control and interpretability**: The use of explicit affine parameters (*e.g.*, memory decay λ, gradient scale a, and bias b) yields transparent update rules. These parameters offer direct insight into the learning dynamics and can be analyzed, tuned, or constrained as needed.

- **Unified view of gradient methods**: LGA generalizes classical approaches. It smoothly interpolates between:
    - Stochastic gradient descent (SGD) or ascent (SGA) (when $\lambda = 0$) for stateless updates;
    - Momentum (when $\lambda \in (0,1)$) for stateful accumulation; and
    - Custom schemes, such as via learned, nonlinear, or context-aware update dynamics.
- **Compatibility**: The modular nature allows LGA to be embedded within existing optimization frameworks or used as a meta-optimization layer.

Ablation studies show that the presence of memory ($\lambda \neq 0$) plays a critical role in scenarios requiring long-term gradient integration, such as continual learning or curriculum-based training. Stateless variants ($\lambda = 0$) perform adequately in transfer or one-shot learning environments.

VI Extensions and Future Directions

LGA provides a foundation for numerous architectural and theoretical expansions:

- **Task-conditioned adaptation**: Incorporating task embeddings or auxiliary inputs allows the LGA parameters to shift dynamically in response to new tasks or domains.
- **Hierarchical and structured accumulation**: Introducing parameter tying, group sparsity, or hierarchical sharing can reduce memory and improve generalization.
- **Probabilistic extensions**: Treating λ, a, b as random variables under a Bayesian framework may capture uncertainty and induce regularization effects.
- **Selective gradient propagation**: Augmenting LGA with gating or thresholding mechanisms supports conditional updates and sparse optimization.
- **Multi-scale memory systems**: Multiple LGA channels with different decay constants may emulate long-term and short-term memory mechanisms in gradient processing.

VII Conclusion

Learnable Gradient Accumulation (LGA) introduces a principled and extensible framework that treats gradient memory as a first-class, learnable component of the optimization process. By parameterizing how past and present gradients interact, LGA unifies a spectrum of update styles—from memoryless to momentum-like to fully context-aware schemes—within a consistent mathematical structure. Its modularity allows for broad integration across architectures and tasks, while its

interpretability enables deeper insight into optimization dynamics. As modern learning systems demand more flexible, context-sensitive, and transparent training strategies, LGA offers a foundational approach to rethinking how gradient information is retained, transformed, and leveraged. Future extensions in task conditioning, probabilistic modeling, and structured memory design further position LGA as a compelling direction for the next generation of optimizers in deep learning.

References

Being finalized – contact for us if you would like the draft in the interim.