Member-only story

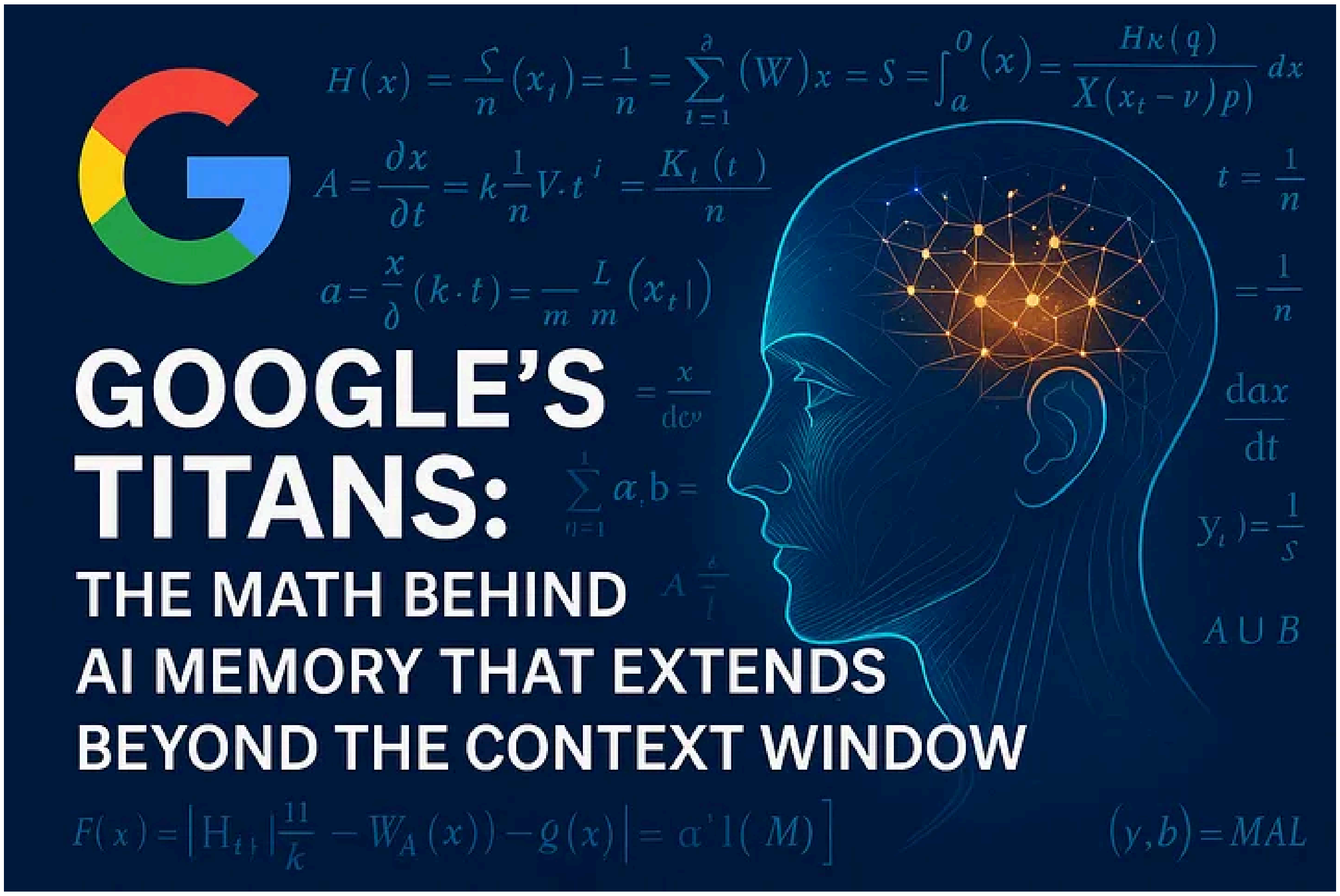
Google’s Titans: The Math Behind AI Memory That Extends Beyond the Context Window

Dive deep into how Titans’ novel long-term memory module learns to remember, overcoming Transformer limitations for truly long sequences

Cristian Leo Following

23 min read · Just now

1



Google's Titans: The Math Behind AI Memory That Extends Beyond the Context Window

The sheer scale and capability of models we see today, often built upon the foundational Transformer architecture, are nothing short of astounding. They can generate coherent text, translate languages, answer complex questions, and even write code. At the heart of much of this magic, of course, lies the self-attention mechanism — a brilliant innovation that allows models to weigh the importance of different words in an input sequence, dynamically capturing dependencies no matter how far apart the words are *within the context window*. And for a while, that felt like the answer to everything sequence-related.

But here's the rub, and it's a significant one: that wonderful attention mechanism scales quadratically with the length of the input sequence. Imagine trying to use a magnifying glass to examine an entire football field at once; you can only focus on a small area, and to see the whole field, you'd need an impossibly large lens or an astronomical amount of effort to scan every tiny bit individually. That's a bit like attention with very long contexts — it quickly becomes prohibitively expensive in terms of both computation and memory. This limitation is keenly felt in real-world tasks like understanding massive documents, processing long videos, or forecasting far into the future based on extensive historical data.

Of course, the idea of giving neural networks memory isn't new. Before Transformers dominated the scene, we had recurrent neural networks (RNNs) and their more sophisticated cousins like LSTMs (Long Short-Term Memory). These models attempted to maintain a form of "memory" in a hidden state that was updated sequentially as new data arrived. However, this hidden state often acted more like a compressed summary, struggling to retain specific details or access information from the distant past reliably. It was a bit like trying to remember an entire book by only keeping a few key points in your head — you lose the nuance and the ability to recall specific passages when you need them.

This brings us to a fundamental challenge in building truly intelligent, context-aware systems: **how can we equip them with memory that is not only vast but also flexible, efficient, and capable of adapting to new information *as it arrives***? It seems to me that we need something that transcends the limitations of both fixed-context attention and simple state compression.

And that's precisely where the recent work on **Titans** comes in. In their compelling paper, "Titans: Learning to Memorize at Test Time" ([Behrouz et al., 2024]), the authors propose a novel approach that centers around a *neural long-term memory module* designed to learn how to memorize historical context *at test time*. This isn't just a static lookup table or a fixed state; it's a dynamic, neural component that actively updates its understanding and storage of information as it processes data during inference. Think of it as giving the model the *capacity* to learn *how* to remember and forget based on the ongoing data stream, much like our own biological memory system, which isn't just a passive recording device.

Titans: Learning to Memorize at Test Time
Over more than a decade there has been an extensive research effort on how to effectively utilize recurrent models and...
arxiv.org

In this article, I want to take you on a journey beneath the surface of Titans. We won't just admire its reported performance on long-context tasks, which, spoiler alert, looks very promising, especially when scaling to previously unattainable sequence lengths (up to 2M tokens and beyond!) — but we will truly dig into the **math** that underpins this fascinating neural memory module. How does it learn to memorize? What are the equations that govern its updates and its ability to forget? How is this "surprise" mechanism mathematically formulated? And how do the different architectural choices in integrating this memory impact its effectiveness?

A Historical and Conceptual Look at Memory in Models

When we talk about intelligence, whether human or artificial, memory is just… inseparable, isn't it? It's the thread that connects past experiences to present understanding, allowing us to learn, adapt, and make sense of the world over time. It's no wonder that researchers building neural networks have always looked to memory systems, both biological and computational, for inspiration. From the early associative memories like Hopfield Networks

([Hopfield, 1982]) to the more complex state-tracking abilities of LSTMs ([Schmidhuber and Hochreiter, 1997]), the quest to give models a robust memory has been central to our field.

In the realm of sequence processing, recurrent neural networks (RNNs) tackled memory by maintaining a hidden state, *M_t*, which was updated sequentially with each new input, *x_t*. You could think of this hidden state as the model's evolving summary or compression of everything it had seen up to time *t*–1. The general process often involves a "write" operation to update the memory based on the new input and the previous state, and a "read" operation to use the current memory state, *M_t,* and input, *x_t,* to produce an output *y_t*. The paper describes this elegantly in a general form (Section 2, Modern Linear Models):

$$M_t = f(M_{t-1}, x_t) \quad \text{(Write Operation)}$$

$$y_t = g(M_t, x_t) \quad \text{(Read Operation)}$$

The functions *f* and *g* could be anything from simple linear transformations to complex gating mechanisms like those in LSTMs. The core idea, however, was this sequential update compressing the history into a fixed-size vector state. This approach, while powerful for capturing sequential dependencies, often struggles with very long sequences because, well, there's only so much information you can realistically cram into a fixed-size vector without losing crucial details from the far past. It's a bit like trying to remember every sentence of a lengthy novel just by keeping a summary in your head — eventually, the details blur together.

Then came the Transformer ([Vaswani et al., 2017]), and it shifted our perspective dramatically. Instead of a single, sequentially updated hidden state, the Transformer's "memory" lies primarily within its self-attention mechanism, operating over a *context window*. Here, the model learns to store information as associative key-value pairs (K and V matrices derived from the input) and retrieve relevant information by querying these keys (Q matrix). Unlike the RNN's compression, the Transformer, within its context window, essentially keeps the *data itself* (in the form of key-value pairs) and learns *associations* between them. The paper views the pair of key and value matrices as the model's memory (Section 2, Memory Perspective). This direct access to information based on learned similarity is incredibly powerful for modeling dependencies, allowing any token to directly interact with any other token *within that window*.

However, this power comes at a cost, as we touched on earlier. The computational and memory complexity of calculating attention over all pairs grows quadratically with the window size. This means Transformers, as revolutionary as they are, function more like models with a highly accurate *short-term* memory — perfect recall within their limited scope, but effectively "forgetting" everything outside that window unless external mechanisms like sliding windows or complex retrieval systems are added. It's a different kind of limitation than the RNN's compression problem, but a significant bottleneck nonetheless when the "context" truly spans millions of tokens.

So, we have these two prominent paradigms: RNNs, which compress history into a sequential, fixed state but struggle with detail over extreme lengths, and Transformers, which keep data in a window for rich interactions but are limited by the window's size and its quadratic cost. Neither, perhaps, fully embodies the kind of vast, yet accessible, long-term memory we intuitively associate with human cognition, or that's needed for tasks demanding deep context over very long periods.

This leads us to some fundamental questions that researchers, including the authors of the Titans paper, are grappling with (Section 2, Memory Perspective): What is the optimal *structure* for a memory module in these large models? How should this memory *update* effectively over time, handling new information while retaining relevant past data? And crucially, how can information be *retrieved* efficiently from such a memory? Beyond structure and updates, there's the question of architecture: how do these memory components fit together? Should short-term and long-term memory modules be distinct but interconnected, much like in our own brains? And finally, is a deep, non-linear memory module necessary to truly capture and store the rich, abstract representations of long-past events, going beyond simple linear compression?

These questions point to the need for memory systems that are not just larger or faster versions of what we have, but fundamentally different, perhaps dynamic, learning, and capable of managing information actively. This is the fertile ground where the ideas behind Titans begin to take root.

Learning to Remember When It Matters Most

Alright, so we've established the limitations of our current memory paradigms in deep learning, particularly when facing truly massive contexts. Now, let's get to the heart of the Titans architecture: the neural long-term memory module, or LMM. This is where the really interesting math comes into play, because, as the paper describes, this module is designed as a "meta in-context model" that literally *learns how to memorize* at test time ([Behrouz et al., 2024], Section 3).

The core idea, and I find this particularly fascinating, isn't just to *have* a memory, but to make the memory *learnable*. It's like teaching the model *how* to take notes during a lecture, rather than just giving it a notepad and hoping it figures things out. The parameters of this neural network *are* the memory, and the model learns to update these parameters dynamically based on the incoming data stream during inference. This stands in contrast to the more traditional view, where memorization during training is often seen as a bad thing, leading to overfitting. Here, memorization *at test time* is the goal, aiming for better adaptation to new, potentially out-of-distribution data ([Behrouz et al., 2024], Section 3.1).

How does this dynamic memorization work? Well, the memory module itself is envisioned as an associative memory. Inspired by Transformers, the model first projects the current input *x_t* into a key *k_t* and a value *v_t* using linear layers. These are like the incoming piece of information you might want to store: the 'key' is what you'd use to look it up later, and the 'value' is the information itself. The math for this projection is straightforward enough:

$$k_t = x_t W_K$$

$$v_t = x_t W_V$$

Here, *W_K* and *W_V* are the learnable weight matrices that define how the input is transformed into these key-value pairs. This is similar to the initial steps in Transformer attention, as noted in the paper ([Behrouz et al., 2024], Section 3.1, Eq 11).

The memory module, which is itself a neural network (let's call its parameters M*M*), learns to associate these keys and values. Specifically, it learns to output the value vt*vt* when queried with the key *k_t*. The objective function that drives this learning process within the memory module is framed as minimizing the difference between what the current memory *M_t*−1 "predicts" for *k_t* and the actual value *v_t*:

$$\ell(M_{t-1}; x_t) = ||M_{t-1}(k_t) - v_t||^2$$

This is the associative memory loss ([Behrouz et al., 2024], Section 3.1, Eq 12). It's an inner loop optimization problem within the overall model's learning process; the outer loop optimizes the parameters like *W_K*,*W_V*, while the inner loop (implicitly handled during inference/test time learning) optimizes *M*.

Now, how does the memory know *what* to memorize? This is where the concept of "surprise" comes in. Drawing inspiration from human memory, where surprising events are often more memorable, the authors define surprise for the model based on the gradient of the associative memory loss with respect to the memory parameters. The larger this gradient, the more the incoming data *x_t* deviates from what the memory *M_t*−1 expected when presented with *k_t*, and thus, the more "surprising" it is. The simplest idea for updating the memory based on this surprise is a form of gradient descent:

$$M_t = M_{t-1} - \theta_t \nabla \ell(M_{t-1}; x_t)$$

Here, ∇ℓ(*M_t*−1 ; *x_t*) represents the gradient of the loss with respect to *M_t*−1, and *θ_t* is a learning rate controlling the update strength. This is the basic surprise-driven update (similar to Eq 8, [Behrouz et al., 2024], though the paper immediately refines it).

However, relying solely on the instantaneous "momentary surprise" (the gradient at the current step) can be problematic. A sudden big surprise might lead the gradient to become very small afterwards, causing the model to miss important information that follows. Think about reading a complex technical document — one surprising sentence might grab your attention initially, but you still need to process the subsequent sentences to understand the full concept. To capture this, the paper introduces a mechanism akin to **momentum** in optimization, allowing the memory update to also incorporate a "past surprise" component ([Behrouz et al., 2024], Section 3.1). The update rule becomes:

$$M_t = M_{t-1} + S_t$$

$$S_t = \eta_t S_{t-1} - \theta_t \nabla \ell(M_{t-1}; x_t)$$

In this formulation, *S_t* acts as a momentum term, accumulating surprise over time (Eq 9 and 10, [Behrouz et al., 2024]). *S_t* itself becomes a "memory of surprise." The terms *η_t* and *θ_t* are crucial here — they are *data-dependent*. This is a neat touch! *η_t* (a function of *S_t* or *x_t*) controls how quickly the "past surprise" (*S_t*−1) decays, while *θ_t* (a function of *x_t*) controls how much the "momentary surprise" (the current gradient) contributes. This data-dependency allows the model some flexibility: it can decide if recent surprises are still relevant based on the new input, perhaps dampening the past surprise if the context changes drastically (*η_t*→0) or fully incorporating it if the new token is highly related (*η_t*→1). It's an adaptive way to manage the flow of surprise information.

Beyond just incorporating past surprise, effectively managing a memory that can grow over very long sequences requires knowing *when to forget*. Even with a deep, potentially large memory space, you can't keep *everything* forever. The paper incorporates an adaptive forgetting mechanism using a gating parameter *α_t*∈[0,1] ([Behrouz et al., 2024], Section 3.1). The memory update rule is refined again:

$$M_t = (1 - \alpha_t) M_{t-1} + S_t$$

$$S_t = \eta_t S_{t-1} - \theta_t \nabla \ell(M_{t-1}; x_t)$$

Here, *α_t* is a gating mechanism (dependent on *x_t* and/or *M_t*−1) that determines how much of the previous memory state *M_t*−1 is retained. If *α_t* is close to 0, the memory changes little due to forgetting, preserving the past abstraction. If *α_t* is close to 1, it effectively clears the previous memory state, letting the new surprise dictate the update. This adaptive forgetting is related to weight decay in optimization and similar gating mechanisms in modern recurrent models, providing a flexible way to manage the memory's capacity based on the data itself.

What about the structure of the memory module *M_t*? While the paper explores simple MLPs (Multi-Layer Perceptrons) for clarity, they explicitly state that this framework allows for more complex neural architectures. They use MLPs with *LM*≥1 layers and find, interestingly, that deeper memory modules (*LM*≥2) are more effective in practice, especially for longer sequences ([Behrouz et al., 2024], Section 3.1, Section 5.5). This aligns with theoretical results showing that deeper networks can be more expressive than linear models, suggesting that encoding long-term history might require capturing complex, non-linear abstractions of the past. It's a trade-off, of course; deeper networks can be slower, as the experiments show (Section 5.5, Figure 8).

Once the memory module has learned and updated its parameters, how is the information retrieved and used by the rest of the model? It's quite elegant: you simply query the memory module with the current input (projected to a query *q_t*) using a forward pass, *without* performing any weight updates at this stage.

$$Y_t = M^*(q_t)$$

Here, *q_t=x_tW_Q* (using a different projection *W_Q* for the query, similar to Transformers) and $M*$ signifies the memory module used in inference mode, leveraging the knowledge stored in its parameters *M_t* to produce a useful output *Y_t* ([Behrouz et al., 2024], Section 3.1, Eq 15).

One technical challenge with a gradient-based memory update like this is parallelization, since recurrence is inherently sequential. However, the authors build upon recent work (like [Yu Sun et al., 2024]) to show how this update, particularly with momentum and weight decay, can be reformulated to leverage matrix multiplications and parallel associative scans over chunks of the sequence (Section 3.2). This is key to making the approach practical on modern hardware accelerators.

In essence, the LMM is a recurrent module whose parameters *are* the memory, and it learns an online update rule driven by "surprise," modulated by momentum, and equipped with an adaptive forgetting gate. This seems like a powerful way to move beyond static state compression and fixed-window attention, aiming for a memory system that is both vast and dynamically adaptive.

Next, let's explore the different ways this clever LMM can be integrated into a full deep learning architecture, leading to the different "Titans" variants presented in the paper.

Different Ways to Combine Short, Long, and Persistent Recall

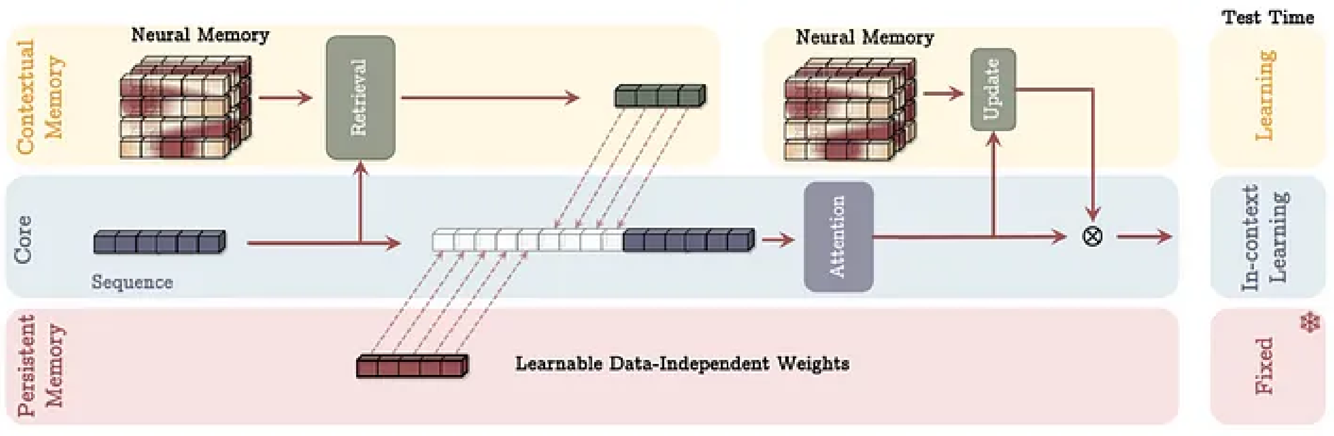
Okay, so we've explored the inner workings of the neural long-term memory (LMM) module, which learns to adaptively memorize information at test time, driven by surprise and managed with momentum and forgetting. It's a powerful concept on its own, but the real trick is figuring out how to best integrate it into a larger, functional architecture that can process sequences effectively. After all, the paper posits that a truly effective learning system, much like the human brain, might rely on distinct but interconnected memory modules (Section 2).

In the world of deep learning, particularly after the rise of Transformers, we've seen various approaches to combining different computational blocks. Should they be stacked sequentially? Should they operate in parallel and combine their outputs? Should one module influence how another processes the data? These design choices aren't just aesthetic; they profoundly impact how the model handles information flow, efficiency, and ultimately, performance on complex tasks requiring long context.

The authors of the Titans paper propose a family of architectures, suggesting three main ways to weave their neural long-term memory into a model that also utilizes attention (interpreted as a form of accurate, but limited, short-term memory). Alongside the LMM, Titans includes what they call "Persistent Memory" — a set of learnable, but *data-independent* parameters (Section 3.3). Think of Persistent Memory as storing general knowledge about the *task* or the overall structure of the problem, information that doesn't change based on the specific input sequence tokens themselves. It's appended to the input sequence right at the start (Section 3.3, Eq 19), like giving the model some fixed instructions or context notes before it even looks at the real data.

Now, let's look at the three main architectural variants they explored, each offering a different philosophy for integrating these memory types:

Memory as a Context (MAC)



Memory as a Context (MAC)Architecture — Image from original paper

The first approach, Memory as a Context (MAC), treats the output of the LMM and the Persistent Memory as additional context that's prepended to the current input sequence segment (Section 4.1). Imagine processing a long document by reading it in chunks (segments). For each new chunk, you don't just read the chunk in isolation; you also retrieve relevant information from your long-term memory (the LMM's current state, Mt−1*Mt*−1) and recall some fundamental background knowledge (the Persistent Memory). All of this, the current chunk, the retrieved long-term history, and the persistent knowledge, is then fed into the attention mechanism as a combined, enriched sequence.

Specifically, given an incoming segment *S*(*t*), the model uses a query derived from this segment, *q_t*, to retrieve information *h_t* from the previous state of the long-term memory *M_t*−1 ∗ (the asterisk denotes inference mode, without weight updates, as we discussed):

$$h_t = M^*_{t-1}(q_t)$$

where *q_t*=*S*(*t*)*W_Q* ([Behrouz et al., 2024], Section 4.1, Eq 21).

Then, this retrieved long-term memory *h_t* is concatenated with the Persistent Memory (a set of tokens *p_*1,…,*p_N*) and the current input segment *S*(*t*) to form a new, extended sequence:

$$\tilde{S}^{(t)} = [p_1, p_2, \ldots, p_{N_p}] \; || \; h_t \; || \; S^{(t)}$$

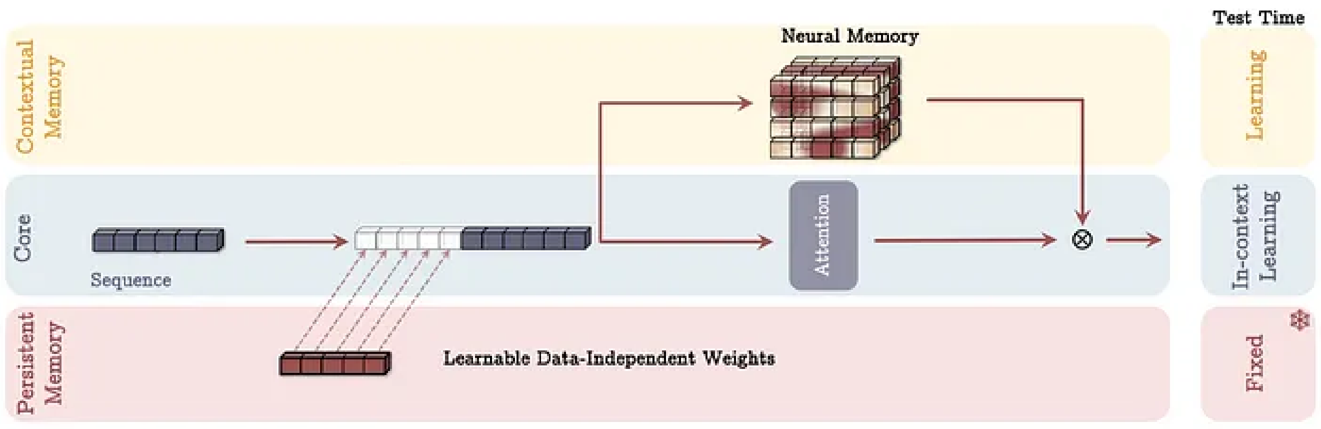
([Behrouz et al., 2024], Section 4.1, Eq 22). This entire sequence then goes through a standard attention module:

$$y_t = \text{Attn}(\tilde{S}^{(t)})$$

([Behrouz et al., 2024], Section 4.1, Eq 23).

The output of the attention module, *y_t*, is then used in two ways: to update the parameters of the long-term memory module for the *next* segment, and to compute the final output. The paper shows the LMM update here using *y_t* (Eq 24) and the final output combining yt*yt* with the memory output (Eq 25). This design feels intuitive — it's like the attention module gets a broader view, seeing not just the current segment but also relevant historical context provided by the LMM and the persistent knowledge. This allows the attention mechanism to decide dynamically whether the long-term information is needed for the current task and, importantly, helps the LMM decide *what* information from the current segment is worth storing for the future by observing what attention focuses on. It's a cooperative model of memory and attention.

Memory as a Gate (MAG)



Memory as Gating — Image from original paper

The second variant, Memory as a Gate (MAG), takes a different approach (Section 4.2). Instead of feeding the LMM output *into* the attention mechanism's input sequence, the LMM updates directly from the input data and its output is used to *gate* or modulate the output of a separate attention mechanism. Here, the model doesn't segment the input data in the same way as MAC; it processes the continuous input stream.

In this architecture, the input (prepended with Persistent Memory) is fed into a sliding window attention (SWA) module. SWA is often used for efficiency with long sequences, limiting attention to a fixed window around each token, plus potentially some global tokens (which here include Persistent Memory, see Figure 3b in the paper).

$$\tilde{x} = [p_1, p_2, \ldots, p_{N_p}] \mid\mid x$$

$$y = \text{SW-Attn}^*(\tilde{x})$$

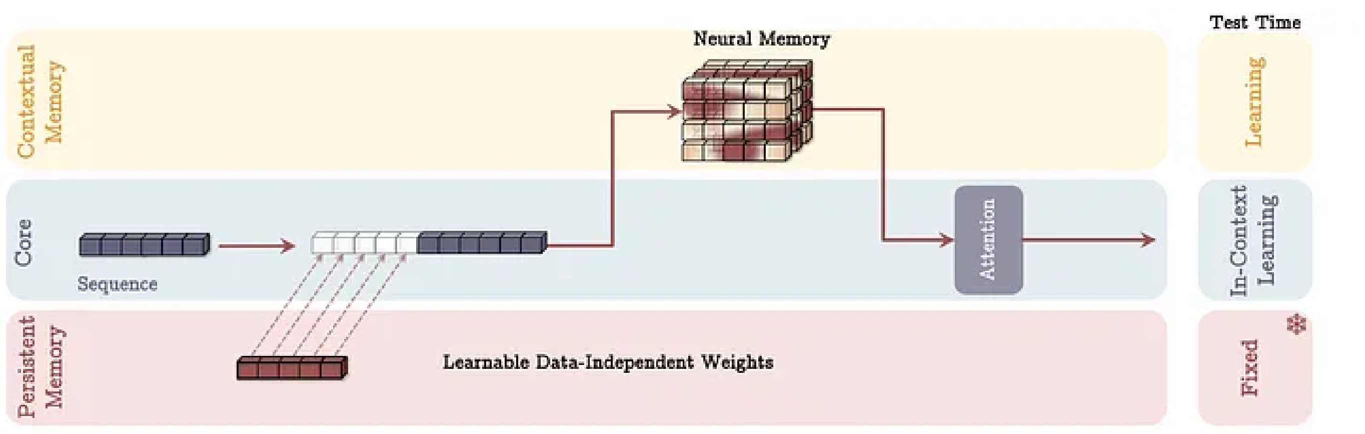
The LMM module, on the other hand, processes the raw input stream (or perhaps a representation derived from it) to maintain its long-term state. At each step, the output from the SWA (*y*) is combined with the output from the LMM *M*(*x*~)) using a non-linear gating function ⊗:

$$o = y \otimes M(\tilde{x})$$

([Behrouz et al., 2024], Section 4.2, Eq 26, 27, 28).

The memory module M*M* here represents the result of the cumulative updates over the sequence tokens. The gating mechanism ⊗ effectively allows the LMM's long-term memory signal to selectively influence or gate the output from the short-term, local attention. It's as if the long-term memory acts as a filter, deciding which parts of the short-term context are most relevant or how they should be interpreted based on the accumulated history. This design emphasizes the LMM's role as a fading memory, working alongside the precise short-term attention of the SWA.

Memory as a Layer (MAL)



Memory as a Layer Architecture — Image from original paper

Finally, the third variant, Memory as a Layer (MAL), follows a more traditional hybrid model structure seen in some prior work (Section 4.3). In this design, the memory module and the attention module are stacked sequentially as layers within the network.

Similar to the other variants, the input sequence is first prepended with the Persistent Memory:

$$\tilde{x} = [p_1, p_2, \ldots, p_{N_p}] \;||\; x$$

([Behrouz et al., 2024], Section 4.3, Eq 29).

This extended input is then processed by the LMM module:

$$y = M(\tilde{x})$$

([Behrouz et al., 2024], Section 4.3, Eq 30). The output of the LMM, *yy*, then serves as the input to a sliding window attention layer:

$$o = \text{SW-Attn}(y)$$

([Behrouz et al., 2024], Section 4.3, Eq 31).

The MAL architecture is perhaps the most straightforward in terms of conceptual design — process the input with memory first, then apply attention. However, the paper points out a potential limitation here (Section 4.3). By processing the data sequentially through separate layers, this design might not fully leverage the *complementary* strengths of the LMM and attention in the same way that MAC or MAG potentially can, where the memory output and attention output interact more directly or attention operates over an enriched input. The power of the overall model becomes limited by the capabilities of each layer individually.

Each of these architectural designs offers a different hypothesis about how best to combine the strengths of attention (accurate short-term dependency modeling) with the novel LMM (learnable, vast long-term memory) and Persistent Memory (task-specific knowledge). As we'll see in the experimental results, each variant has its own advantages and disadvantages, often showing interesting trade-offs between efficiency and effectiveness for very long contexts. It's a clear signal that how we structure these complex memory systems matters a great deal.

Titans in the Arena: Advantages and Considerations

So, we've dissected the clever engineering and mathematical foundations of the Neural Long-Term Memory (LMM) and explored a few ways it can be integrated into the broader Titans architecture. But how does this all stack up in practice? Does this learnable, dynamic memory system actually deliver on the promise of handling those challenging, long sequences where other models falter? Based on the experimental results presented in the paper, it seems the answer is a resounding yes, though with some interesting nuances and trade-offs worth considering.

One of the most compelling advantages of Titans, as highlighted by the experiments (Section 5), is its performance on tasks requiring genuinely *long context*. Think about the Needle-in-a-Haystack (NIAH) task, where the model has to find a specific piece of information buried deep within a very long document. Standard Transformers hit a wall due to their fixed context window, while many recurrent models struggle to retain information accurately over such lengths. The paper shows Titans, particularly the LMM module on its own and the MAC variant, achieving significantly higher accuracy on NIAH across varying sequence lengths (2K, 4K, 8K, 16K) compared to baselines like TTT, Mamba2, and DeltaNet (Section 5.3, Table 2). This ability to perform well even as the sequence grows isn't always a given, even for models designed for long context, as the paper points out (Section 5.3, referencing [Hsieh et al., 2024]).

Beyond just accuracy on retrieval tasks, Titans also shows strong performance on broader language modeling and commonsense reasoning benchmarks, often outperforming both Transformer-based models (with the same context window) and recent linear recurrent models across various parameter sizes (Section 5.2, Table 1). The paper attributes this superiority to the LMM's ability to manage memory effectively, contrasting it with models like TTT (which uses a gradient-based update but lacks the sophisticated forgetting/momentum) and Mamba2 (which has gating but, perhaps, benefits less from the deep, non-linear nature of the LMM's memory module).

The ability to scale is another major win. While comparing directly to Transformers with an *infinite* context window is difficult (as it's computationally impractical), Titans demonstrates the capacity to handle contexts much larger than typical Transformer windows, showing competitive or superior performance to Transformers with limited context, and crucially, scaling effectively to **over 2 million tokens** in needle-in-haystack tasks ([Behrouz et al., 2024], Abstract, Section 5). This is a realm where traditional Transformers simply aren't feasible without complex modifications.

What gives Titans this edge? I believe it ties back to the core innovations in the LMM we discussed. The combination of a surprise-driven, learnable update, the momentum term that smooths the surprise signal and retains past context details, and the adaptive forgetting gate that intelligently manages memory capacity seems to create a more robust and flexible long-term memory system compared to simpler state updates or fixed-size memory structures. The ablation studies (Section 5.9, Table 5) provide empirical evidence for this, showing that removing any of these components — momentum, weight decay (forgetting), or even the convolutional layers used in the LMM's implementation — negatively impacts performance. Persistent Memory also plays a positive role, likely providing a stable foundation of task-relevant knowledge that complements the dynamic LMM.

Furthermore, the paper's exploration of different architectural variants (MAC, MAG, MAL) reveals valuable insights into how memory should ideally interact with other model components like attention. The MAC and MAG variants, which integrate the LMM and attention in parallel or via gating, fluidly, perhaps even influencing each other's processing or which information is prioritized, is more beneficial than simply stacking them layer by layer. It feels like a nod to the complexity of human memory systems, where different components cooperate rather than just hand off information linearly.

However, it's important to maintain a balanced perspective. While Titans shows significant advantages, there are also considerations, particularly regarding efficiency and architectural choices. The paper notes that while the LMM approach is parallelizable (Section 3.2), the LMM module itself can be slightly slower than some highly optimized linear recurrent models like Mamba2 ([Behrouz et al., 2024], Section 5.8). This is perhaps understandable given the LMM's deeper, more expressive non-linear architecture and potentially more complex update process compared to the simpler, linear recurrences of models like Mamba2. There's often a trade-off between model expressivity and raw throughput.

Another interesting trade-off revealed in the ablation studies is the impact of deep memory within the LMM itself. While increasing the depth of the LMM improves performance on longer sequences (Section 5.5, Figure 7), it also leads to slower training throughput (Figure 8). This highlights a practical reality in model design: you often have to balance the desire for more powerful, complex components with the need for computational efficiency, especially when training large models on massive datasets.

So, Titans presents a compelling step forward, particularly for long-context sequence modeling, by introducing a novel learnable memory module. But it also opens up new questions about the optimal ways to combine different memory types and the inherent trade-offs between memory complexity and computational speed. It seems we're still early in understanding the true art of building neural systems with robust, scalable memory.

Next, let's look at the concluding thoughts from the paper and reflect on what this work means for the future of building large language models and other sequence processing systems.

Conclusion

The mathematical framework behind the LMM, with its associative loss function, its use of gradients as a "surprise" signal, and the incorporation of momentum and an adaptive forgetting gate, really paints a picture of a memory system that's dynamic and attentive in a very different way. It's not just passively storing or compressing; it's actively deciding what's salient based on the incoming data's novelty and relevance, all while managing its capacity. This blending of online learning principles with a recurrent neural network structure feels like a genuinely promising direction.

And the empirical evidence seems to back it up. The experiments show that Titans, in its various architectural configurations (MAC, MAG, and even the standalone LMM), demonstrably pushes the boundaries of effective context length. Scaling gracefully to over 2 million tokens on tasks like Needle-in-a-Haystack, where quadratic-cost models simply cannot operate, is a significant achievement. While there are trade-offs, as we saw with the impact of memory depth on throughput, the overall picture suggests that this approach offers a powerful new tool for tackling problems that were previously out of reach due to sequence length limitations.

The exploration of the MAC, MAG, and MAL architectures also provides valuable insights into how memory and attention can best cooperate. It seems the integrated approaches (MAC and MAG), allowing for more direct interplay and influence between the short-term attention and the long-term learnable memory, often outperform a simple stacked structure (MAL). This highlights that designing these complex systems isn't just about building better individual components, but also about the thoughtful architecture that enables their complementary strengths to shine.

Titans, with its learnable, dynamic, deep neural memory, represents a compelling step forward in equipping models with the kind of vast, adaptable memory needed for the next generation of language understanding and generation tasks over truly long contexts. It's exciting to see the field exploring these bio-inspired, online learning approaches to memory.

What are your thoughts on Titans and its unique approach to learnable memory at test time? Do you think this is the key to unlocking truly long-context capabilities? Share your ideas in the comments below!

If you found this deep dive into the math and ideas behind Titans interesting, consider following me here on Medium for more articles exploring the fascinating world of data science and machine learning.

Artificial Intelligence Large Language Models Deep Learning Mathematics Machine Learning

Written by Cristian Leo

34K followers · 13 following

Following

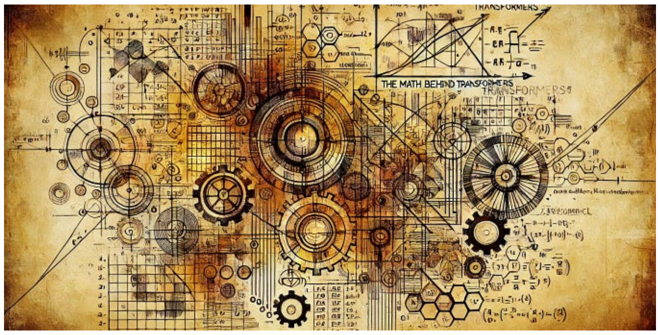
Data Scientist @ Amazon with a passion about recreating all the popular machine learning algorithm from scratch.

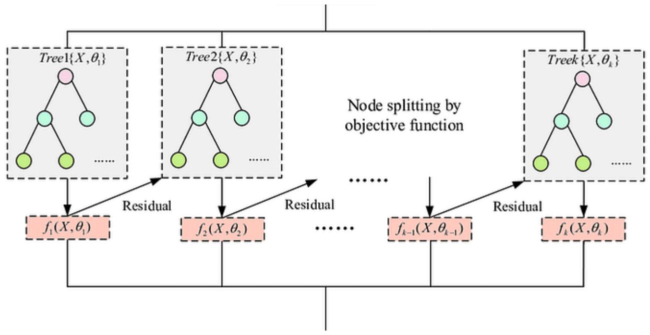
No responses yet

Alex Mylnikov

What are your thoughts?

More from Cristian Leo





Cristian Leo

The Math Behind Transformers

Deep Dive into the Transformer Architecture, the key element of LLMs. Let's explore its...

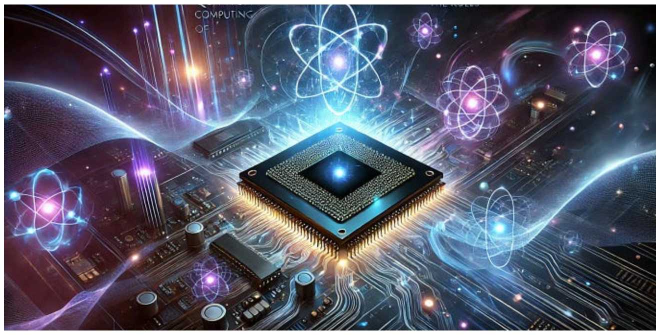
Jul 25, 2024 1.1K 12

Cristian Leo

The Math Behind XGBoost

Building XGBoost from Scratch Using Python

Jan 10, 2024 500 5

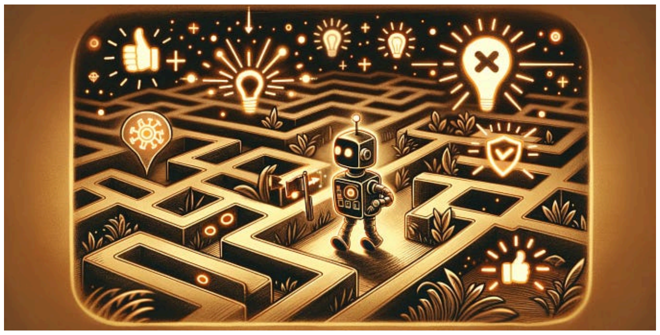


In Data Science Collective by Cristian Leo

How Qubits Are Rewriting the Rules of Computation

From Classical Certainty to Quantum Possibility: Exploring the Science, Math, and...

Mar 24 91



In TDS Archive by Cristian Leo

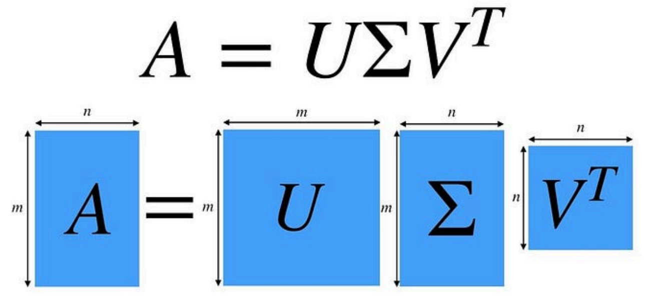
Reinforcement Learning 101: Building a RL Agent

Decoding the Math behind Reinforcement Learning, introducing the RL Framework, an...

Feb 19, 2024 907 10

See all from Cristian Leo

Recommended from Medium

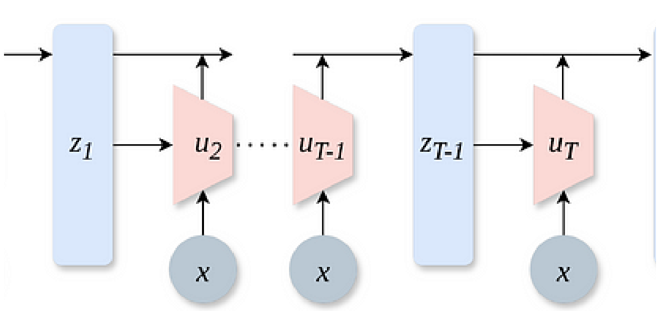


Ameh Emmanuel Sunday

Cracking Complexity: How Singular Value Decomposition Makes Data...

Dimensionality reduction made easy!

Mar 1 31 4

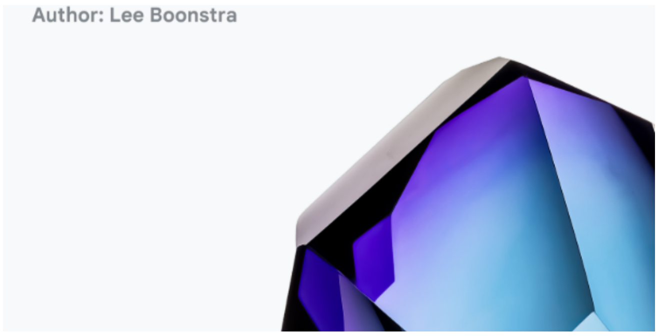


Pietro Bolcato

NoProp: Training Neural Networks Without Back-Propagation or...

A Gradient-Free Alternative that Beats Prior "No-Backprop" Methods

Apr 25 25



In Coding Nexus by Algo Insights

Google's 69-Page Prompt Engineering Masterclass: What's...

I've been writing about tech for three years, and let me tell you, nothing's grabbed me...

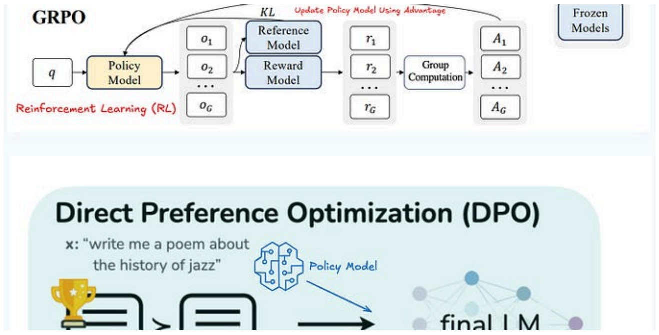
Apr 13 351 11

Benjamin Marie

How Well Does Qwen3 Handle 4-bit and 2-bit Quantization?

Let's review Qwen3 and check which one you should use

4d ago 79


GRPO
KL
Update Policy Model Using Advantage
Frozen Models
q
Policy Model
Reference Model
Reward Model
Group Computation
Reinforcement Learning (RL)
Direct Preference Optimization (DPO)
x: "write me a poem about the history of jazz"
Policy Model
final LM

rohola zandie

How to read machine learning papers? A practical perspective...

Part 1: Scalars, Vectors, Matrices, and Tensors

Apr 26 109

In AI Exploration Journey by Florian June

The Best Way to Understand PPO, GRPO, and DPO: 3 Simple...

Recently, with the rise of various reasoning LLMs, terms like GRPO, DPO, and PPO have...

Apr 6 85

See more recommendations