Permanent Domain Memory and Faster KV-cache: Tauformer, the Topological Transformer

Lorenzo Moriondo
Independent Researcher - tuned.org.uk
ORCID: 0000-0002-8804-2963

1 January 2026

Abstract

Starting from my research about vector search for topological spaces (spectral search with *taumode*), I introduce in this paper the definition, implementation and testing of a new class of Transformer architecture [1] focused on improving the attention mechanism. The Topological Transformer, or *Tauformer* applies all the features of current attention-based transformers (taking as a baseline *nanoGPT* and its architecture [3]) and redesigns the attention mechanism at its core to: *(i)* allow delivering domain-specific context at the level of the attention mechanism for more domain-relevant token generation, *(ii)* improve KV-cache memory by saving 50% compared to traditional KV-caching and *(iii)* draw a path to improve the Transformer performance on large context windows with high-dimensional heads, by substituting inner-product with *taumode*'s synthetic index-based distance, aiming to provide relevant linear gains in training and generation compared to current GPTs.

1 Introduction

Concepts and tools developed in my previous publication *ArrowSpace: introducing Spectral Indexing for vector search* [2] have been here reused to redesign the attention mechanism starting from the *nanoGPT* implementation of Transformer architecture; in particular *taumode*, a synthetic index based on the Rayleigh quotient [13] to compute the distribution of energy in the network defined by the embeddings space. Examples about how to compute the *taumode* distribution for any space of embeddings is available in the *pyarrowspace* repository.

Tauformer makes possible downstream technical improvements in computing and memory usage for the Transformer (GPT) and they are detailed in the following sections. These improvements are a consequence of pursuing the concept of providing a memory layer to LLMs brought forward the idea of leveraging **distilled knowledge graphs (dkb) to deliver domain information to the attention mechanism at token generation level**. For dkb I intend for example: *(i)* Text embeddings generated by feeding to an embedding model (in one of the examples TSDAE [10]) a representative corpus of the domain to map, this provides a vector space with manageable number of dimensions (384 in the example) that is a prerequisite to build the context windows at the attention level; *(ii)* Graph embeddings of triples generated using graph embeddings techniques like the SEPAL workflow demonstrated in [11], again the result is a vector space with a manageable dimensionality. Once a stable dkb is generated and its Graph Laplacian computed using *arrowspace* [2], it can be considered from the point of view of the LLM system as persistent memory that is founded on real-world-defined relations. Assuming a well-designed embedding pipeline, the ground truths' numerical representations are the text embeddings of a curated text corpus or a curated

KB turned into graph embeddings. Theoretically the Tauformer can also work, as any other transformer, on images and audio vectors or a mix of them but one of the critical points focuses on the process of producing the embeddings, how the vector space is designed and how its dimensionality fits in the attention mechanism. The choices for: embeddings, attention mechanism and seeding the latent space are taken as design constraints for the data pipeline from the corpus to the latent space of the model. What is meant with seeding the latent space? Once the attention mechanism is working with *taumode* synthetic scores the content of the dkb percolates in the context windows through the usual Q, K, V search iteration. Each distance metrics computed for Q, K is considered in the optics of the corpus/dataset from which the original embeddings and subsequently the token embeddings used also in the KV-cache are defined, producing domain-specific V.

2 Statement of Need

This research is motivated by encoding at generation time an explicit (controllable and interpretable) domain structure beyond what the model can implicitly absorb into weights during training. Tauformer keeps the familiar Q, K, V and causal softmax pipeline but replaces the dot-product kernel with a distance metrics built on domain-specific manifold: each token/head is mapped to a topological signal and built into a scalar (*taumode* score) derived from the Graph Laplacian; this way attention is driven by distances in that manifold-aware scalar space rather than raw vector similarity. The hypothesis is that this process makes the model more scoped to the context in the training phase (forward) and in the generating phase (decoding), because weights and attention scores can be biased toward tokens that are similar under the learned domain manifold (e.g. a knowledge graph, citation graph, text embedding manifold). This has the potential of improving contextual faithfulness relative to purely dot-product-based retrieval inside the context window. Not having permanent reference about the context is especially consequential in scientific and knowledge-intensive settings, where the desired notion of "relevance" is frequently defined less by raw semantic proximity and more by the topology of domain relations and how information propagates over them.

Tauformer is also motivated by the practical difficulty of scaling traditional GPTs to long contexts: full-sequence self-attention has $O(T^2)$ compute in both the dot-product logits and the value aggregation, and decode-time still grows $O(T)$ per token while KV-cache memory grows linearly with T. Tauformer's cache design reduces KV memory by storing values plus a compact key-side scalar (k_i, λ_i) instead of full K and V tensors, yielding roughly a halving of KV-cache memory per layer (up to a small overhead for the scalar stream). With a sparse Laplacian from a domain manifold, the extra cost of computing the tau scalars can shift from $O(D^2)$ to a sparsity-dependent cost $O(\mathrm{nnz}(L))$, making the incremental overhead largely independent of sequence length and therefore more attractive as context windows grow.

Tauformer addresses this by redesigning attention around a synthetic, topology-aware index computed over the domain embedding space. Beyond accuracy and controllability, this substitution is also motivated by systems constraints: the approach reduces dependence on storing key/value histories for long contexts and enables more memory-efficient inference, opening space for higher-dimensional heads and richer internal representations under fixed hardware budgets. The Graph Laplacian is usually a DxD matrix where D is the number of dimensions that for the test dataset are defined by current SOTA embeddings models to 384. So basically a constant 384x384 matrix allows limiting the computing cost for the search of Q over K in the attention mechanism. The Graph Laplacian can be seen as a distillation of the domain knowledge and its generation takes a cheap pre-training step that for the test dataset is 300 seconds on the common laptop hardware for a dataset of 300000 vectors on 384 dimensions.

The code for the GPT implementation is available at [4]. The code used to test and collect benchmark for the KV-cache is available at [5].

3 Memory Model and Algorithm

3.1 Generics

By leveraging the Graph Laplacian matrix (gl) built on the vector space defined by the embeddings for a given domain, it is possible to define a synthetic score ($\lambda\tau$, lambda-tau) that can be used to spot approximate nearest neighbours of a given vector (x) as: $\lambda\tau = taumode(x, gl)$. This distance metrics has been applied to the process of computing query, keys, value vectors (Q, K, V) in the attention mechanism; allowing the substitution $softmax(Q^T\dot{(}K))$ with $-|\lambda_q - \lambda_k|$. This also enable saving a relevant percentage of the memory space used by the KV-cache and opens to having vectors and attention heads with larger number of dimensions because part of the computing used by the inner-product operation can be reused to compute on higher dimensional vectors. These improvements sound already a nice step forward but, from knowledge engineering perspective, they are a side-effect of the main concept brought forward by the implementation overall: to develop a delivery system to bring distilled domain knowledge from the vector space (representing the knowledge graph, graph of citations, or any relations-representative structure) directly into the attention mechanism. This to test the hypothesis that token generation can be made more context-relevant if domain-specific metadata is provided at the attention level. If this is made possible a new class of Transformers that are more or less narrowly scoped for context-specific generation in a given domain can be developed.

3.2 Tau Attention

In the process of causal attention using the *softmax* function, attention is driven by a transcendental normalisation over dot-product logits, where each new query must form inner-products against **all cached keys** (length T) in head dimension D, i.e., the decode-time kernel cost scales as $O(B\,H\,T\,D)$ for the $QK^\top$ logits plus another $O(B\,H\,T\,D)$ to aggregate values, with an additional $O(B\,H\,T)$ for masking and the softmax itself. Where the letters are:

- **B**: Batch size (number of sequences processed in parallel).
- **H**: Number of attention heads.
- **T**: Sequence length / number of time steps (tokens) in the context window.
- **D**: Head dimension (the per-head vector size, typically $D = C/H$ where C is the model embedding width).

In practice this also forces the KV-cache to store both K and V tensors, i.e., $2 \cdot B\,H_{kv}\,T\,D$ floats per layer, because the dot products cannot be reconstructed without the full key vectors. As context length grows, this "match against all previous keys and renormalise" pattern dominates both compute and memory, since every step repeats the same D-dimensional comparisons against an ever-growing set of cached vectors. Obviously this can be become a problem for high D.

Tauformer's *taumode* mechanism replaces the dot-product kernel with a scalar spectral signature per head vector: for each query/key $x \in \mathbb{R}^D$, it computes a bounded Rayleigh quotient energy $x^\top Lx/(x^\top x)$ (optionally blended with an item/edge dispersion statistic), producing a single λ-like score per token per head. Attention logits are then built by comparing the query's λ_q to **previously stored** λ_k values for cached keys:

$$a_{ij} = -|\lambda_{q,i} - \lambda_{k,j}|/\text{temp}$$

after which the causal mask and V-weighted sum are reused unchanged. This changes what must be cached: tauformer stores V and the **scalar** λ_k history rather than full K, reducing cache size

from $2B\,H_{kv}\,T\,D$ floats to $B\,H_{kv}\,T\,D + B\,H_{kv}\,T$ floats

(about 50% savings for typical D). Compute shifts accordingly: instead of spending $O(B\,H\,T\,D)$ on dot-products at each decode step, tauformer pays $O(B\,H\,D)$ to compute the query's Rayleigh term with a dense $D\times D$ Laplacian (or $O(B\,H\,\mathrm{nnz}(L))$ with a sparse Laplacian), and only $O(B\,H\,T)$ to compare scalars against cached λ_k, making the scoring step dimension-light while keeping value aggregation $O(B\,H\,T\,D)$ the same.

The key improvement is that the Graph Laplacian is designed to be a sparse matrix, so optimising the code for a sparse representation the $O(D)$ terms become $O(nnz)$ with nnz being the number of non-zero elements in the matrix.

Further computing advantages are discussed in 4.

3.3 Built-in Memory

Beside the additional potential computing advantages that are analysed below, the Tauformer/-tauGPT concept allows the kind of persistent memory described in 3.1; **merging a persistent (long-term, unchanging in this initial design) memory with the attention mechanism**. This follows current hypothesis of adding memory sub-systems to LLMs: for example as seen in [6] where a neural long-term memory module for 2M+ token contexts works in parallel and provides prefix sub-vectors to the context window; or in [7] that categorises memory into sensory, short-term and long-term within agentic workflows; or in [8] that uses a "Memory Bank" to store and retrieve long-form text for in-context learning. In addition, the papers that tries to supplement LLMs with agentic workflows: for example [9] that evaluates "Agentic Memory" across multi-hop and temporal reasoning tasks. For the purpose of this paper I call the running attention mechanism the "momentary memory" and the Graph Laplacian against which the distance scores are computed "persistent memory". This aligns with the scope of my current research of building an "AI Memory Layer" based on topological vector search or defining "Memory Models" (MM) [12].

Tauformer/tauGPT is the only, at my current knowledge, Transformer architecture that delivers domain knowledge directly in the attention mechanism; leveraging the compression made possible by the *arrowspace* library and the *taumode* synthetic index. While technical and philosophical advantages are brought forward by this paper, further research is necessary to ascertain the improved accuracy in the training and token generation on very large context windows and the improved capability in avoiding hallucinations or deadlocks at token generation time. As a following version, it is also possible to imagine defining an updating mechanism for the permanent memory that would make it into a medium-term memory that can be adapted to the newly generated tokens.

3.4 Code

The first version of Tauformer/tauGPT [4] has been implemented using the Rust programming language [14] and the Burn Deep-Learning framework [15] as they provide fast, structured and type-safe development cycles with mature production ecosystem.

No inner-product TauGPT's most peculiar change is that it swaps dot-product attention for lambda-distance attention, where each token/head vector is compressed into a single scalar λ derived from a Laplacian energy, so that attention logits become $-|\Delta\lambda|/T$. The core of compression is in `lambdas_from_heads(...)`, which flattens $[B, H, T, D]$ into $[N, D]$, computes xL via a matmul, then forms $E_{\text{raw}} = (x^\top Lx)/(x^\top x + \epsilon)$ and bounds it as $e_raw/(e_raw + tau)$. The scalar λ is then broadcast into a full attention matrix using `taumode_distance_logits(...)`, which literally builds logits as $-((lq - lk).abs()/temp)$ after reshaping into $[B, H, Tq, 1]$ and $[B, H, 1, Tk]$.

```rust
// taumode.rs
pub fn lambdas_from_heads<B: Backend>(
x: Tensor<B, 4>,
lap: Param<Tensor<B, 2>>,
cfg: &TauModeConfig,
) -> Tensor<B, 3> {
  let [b, h, t, d] = x.dims();

  // Flatten [B,H,T,D] -> [N,D]
  let n = b * h * t;
  let x_nd = x.reshape([n, d]);

  // y = x L  -> [N,D]
  let y_nd = x_nd.clone().matmul(lap.val());

  // numerator = sum_i x_i * (xL)_i
  let numerator = (x_nd.clone() * y_nd).sum_dim(1); // [N]

  // denominator = sum_i x_i^2 + eps
  let denom = x_nd.powf_scalar(2.0).sum_dim(1) + cfg.eps; // [N]

  let e_raw = numerator / denom; // [N]
  let e_bounded = e_raw.clone() / (e_raw + cfg.tau); // [N]

  e_bounded.reshape([b, h, t])
}

pub fn taumode_distance_logits<B: Backend>(
lambda_q: Tensor<B, 3>,
lambda_k: Tensor<B, 3>,
cfg: &TauModeConfig,
) -> Tensor<B, 4> {
  let lq = lambda_q.unsqueeze_dim::<4>(3); // [B,H,Tq,1]
  let lk = lambda_k.unsqueeze_dim::<4>(2); // [B,H,1,Tk]
  let temp = cfg.temperature.max(cfg.eps);
  -((lq - lk).abs() / temp)
}
```

Sparse Laplacian The other distinctive design choice is that `TauModeAttention` is built to operate with either a dense "toy" Laplacian or, as intended by design, a sparse Laplacian loaded from a manifold (a `.parquet` file with the computed Laplacian). The code explicitly switches between those representations at runtime. In `TauModeAttention`, you can see the dual storage for the sparse matrix and the dense tensor. All tests are run using the sparse matrix from a pre-trained `arrowspace` (manifold file and embeddings for the test queries available in [5]).

```rust
// tauattention.rs (struct fields + sparse/dense selection)
pub struct TauModeAttention<B: Backend> {
  // ...
  laplacian_tensor: Option<Param<Tensor<B, 2>>>, // using dense
  laplacian_matrix: Ignored<Option<CsMat<f64>>>, // using sparse
  pub(crate) tau_mode: Ignored<Option<TauMode>>,
  // ...
}

fn lambdas_from_heads_any(&self, heads: Tensor<B, 4>) -> Tensor<B, 3> {
  let tau_cfg = self.get_tau_config();
  if let Some(lap) = self.laplacian_matrix.0.as_ref() {
    let mode =
        self.tau_mode.0.unwrap_or(crate::pretraining::parquet::TauMode::Median);
```

```
      crate::taumode::lambdas_from_heads_sparse::<B>(heads, lap, mode,
          tau_cfg.eps)
    } else {
      let lap = self.get_laplacian_tensor().clone();
      crate::taumode::lambdas_from_heads::<B>(heads, lap, &tau_cfg)
    }
  }
```

KV-caching layout KV-cache in tauGPT is also different compared to standard GPT: instead of caching K and V, each layer caches $(V, lambda_k)$ only, which matches the fact that scoring uses lambda scalars rather than key vectors. The cache type is defined as *pub type TauCacheLayer<B> = Option<(Tensor<B, 4>, Tensor<B, 3>)>;* and tauGPT wraps a vector of these per layer in *TauKVCache store: Vec<TauCacheLayer<B», position: usize* , updating `position` each decode step.

```
// tauattention.rs (KV-cache payload + append logic)
pub type TauCacheLayer<B> = Option<(Tensor<B, 4>, Tensor<B, 3>)>;

let lambda_k_new = self.lambdas_from_heads_any(k_new); // [B, Hkv, 1]

// Cache management
let (v_full, lambda_k_full) = match cache_layer.take() {
  Some((v_all, lk_all)) => (
  Tensor::cat(vec![v_all, v_new.clone()], 2),              // time axis
  Tensor::cat(vec![lk_all, lambda_k_new.clone()], 2),      // time axis
  ),
  None => (v_new.clone(), lambda_k_new.clone()),
};

*cache_layer = Some((v_full.clone(), lambda_k_full.clone()));
let tk = v_full.dims()[2];
let y = self.scaled_tau_attention_decode(q, lambda_k_full, v_full, 1, tk);
```

Inside `TauModeAttention::forward_decode(...)`, the cache logic appends along the time axis and then runs attention against the full cached history via `scaled_tau_attention_decode(q, lambda_k_full, v_full, 1, tk)`.

```
// taugpt.rs (model-level decode uses cache.position for RoPE step
    slicing)
pub fn forward_decode(
&self,
last_ids: Tensor<B, 2, Int>, // [B,1]
cache: &mut TauKVCache<B>,
use_softcap: bool,
) -> Tensor<B, 3> {
  let tpos = cache.position;
  let d2 = self.cos.dims()[3];

  // Slice RoPE for the current absolute position: [1,1,1,D/2]
  let cos_step = self.cos.clone().slice([0..1, tpos..tpos + 1, 0..1,
      0..d2]);
  let sin_step = self.sin.clone().slice([0..1, tpos..tpos + 1, 0..1,
      0..d2]);

  for (i, block) in self.blocks.iter().enumerate() {
    let layer_cache = &mut cache.store[i];
    x = block.forward_decode(x, (&cos_step, &sin_step), layer_cache);
  }

  cache.position += 1;
```

```
    logits
}
```

4 Results

Multiple tests have been run and results have been collected using [5].

These are initial results running a limited benchmark on a local laptop but still promising and worth replicating at larger scale. The base assumption is confirmed that tauGPT is slightly faster than nanoGPT in a "no cache" mode and this is a good baseline to work on, in particular for the training phase of a large model. NanoGPT's "kv-cache" mode is dominated by extremely optimised GEMM and fused softmax kernels, while tauGPT's "kv-cache" likely pays additional lambda-computation overhead and less-optimized element-wise/broadcast kernels. There is a lot of room for optimisations in particular if we consider, as mentioned above, that the Graph Laplacian is computed to be a sparse matrix; so optimising the code for a sparse representation turns the number of dimensions D into nnz or the number of non-zero elements in the matrix (or a projection of) with potentially outperforming the D-bound (or projected) softmax.

The potential gains of tauformer/tauGPT in this run are clear: tauGPT in "no cache" mode is slightly faster than nanoGPT "no cache" (median tokens/sec 3.921 vs 3.610, and median p50 token latency 257 ms vs 279 ms), suggesting some benefit even when recomputing attention each step. More importantly for deployment, tauGPT's "kv-cache" mode retains the "flat-latency" behaviour while operating with a different scoring kernel (lambda-distance rather than dot-product), indicating that the method can be implemented without introducing an obvious per-token latency blow-up over the tested decode length.

Larger improvements are tested in larger windows generation tests (256, 512, 1024 token). The driving hypothesis is that when caching becomes too expensive memory-wise (like for very long windows with high-dimensional embeddings), tauGPT can provide a better solution both in terms of contextual accuracy and performance. Current research is exploring at the order of magnitude of 10000 dimensions with values for new token generation (compute logits $\rightarrow$ pick next token $\rightarrow$ append cycle) of 1024.

5 Conclusion

Here some basic data analysis of the results.

6 Acknowledgments

The author is an independent researcher who self-funded this work. All works available at his research page. Thanks to all the backers and my supporting network of peers.

References

[1] Vaswani, Ashish and Shazeer, Noam and Parmar, Niki and Uszkoreit, Jakob and Jones, Llion and Gomez, Aidan N and Kaiser, Lukasz and Polosukhin, Illia *Attention is All You Need*, Advances in Neural Information Processing Systems (NeurIPS), volume 30, 2017

[2] Lorenzo Moriondo, *ArrowSpace: introducing Spectral Indexing for vector search*, 2025. https://github.com/tuned-org-uk/arrowspace-rs DOI: 10.21105/joss.09002

[3] Andrej Karpathy *nanoGPT: The simplest, fastest repository for training open-source GPT models*. https://github.com/karpathy/nanoGPT

[4] Lorenzo Moriondo (@Mec-iS) Tauformer . https://github.com/tuned-org-uk/taugpt-kvcache-bench

[5] Lorenzo Moriondo (@Mec-iS) KV-cache bench for Tauformer. https://github.com/tuned-org-uk/taugpt-kvcache-bench

[6] Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to Memorize at Test Time. *arXiv preprint arXiv:2501.00663*, 2025.

[7] Lianlei Shan, Shixian Luo, Zezhou Zhu, Yu Yuan, Yong Wu Cognitive Memory in Large Language Models. *arXiv preprint arXiv:2504.02441*, 2025.

[8] Weizhi Wang, Li Dong, Hao Cheng, Xiaodong Liu, Xifeng Yan, Jianfeng Gao, Furu Wei Augmenting Language Models with Long-Term Memory. *arXiv preprint arXiv:2306.07174*, 2024 (Updated).

[9] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, Yongfeng Zhang A-MEM: Agentic Memory for LLM Agents. *arXiv preprint arXiv:2502.12110*, 2025.

[10] Kexin Wang, Nils Reimers, and Iryna Gurevych. 2021. TSDAE: Using Transformer-based Sequential Denoising Auto-Encoder for Unsupervised Sentence Embedding Learning. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 671–688, Punta Cana, Dominican Republic. Association for Computational Linguistics.

[11] Félix Lefebvre and Gaël Varoquaux. 2025. SEPAL: Scalable Embedding Propagation ALgorithm for large knowledge graphs. In *Proceedings of the 13th International Conference on Learning Representations (ICLR 2025)*, Singapore.

[12] Blog posts at tuned.org.uk. Tuned Blog. www.tuned.org.uk/blog, 2025. Accessed: December 29, 2025.

[13] J. W. Strutt (Lord Rayleigh), *The Theory of Sound*, Vol. 1, London: Macmillan and Co., 1877.

[14] The Rust Project Developers, *The Rust Programming Language*, 2024. [Online]. Available: www.rust-lang.org

[15] Tracel-AI, *Burn: A Deep Learning Framework Designed from Engineers' Perspectives*, 2025. [Online]. Available: https://github.com/tracel-ai/burn