Analysis of the Geometric Transformer

Sridhar Mahadevan, Adobe Research and U.Mass, Amherst

Recap

- The ordinal category Δ has as objects [n], and arrows weakly order-preserving functions on [n]
- Simplicial objects are contravariant functors $X[n] : \Delta \to C$
- GAIA: Generative AI Architecture based on simplicial objects
- Geometric Transformer is a GAIA architecture
- Diagrammatic Backpropagation is used to train a GT model

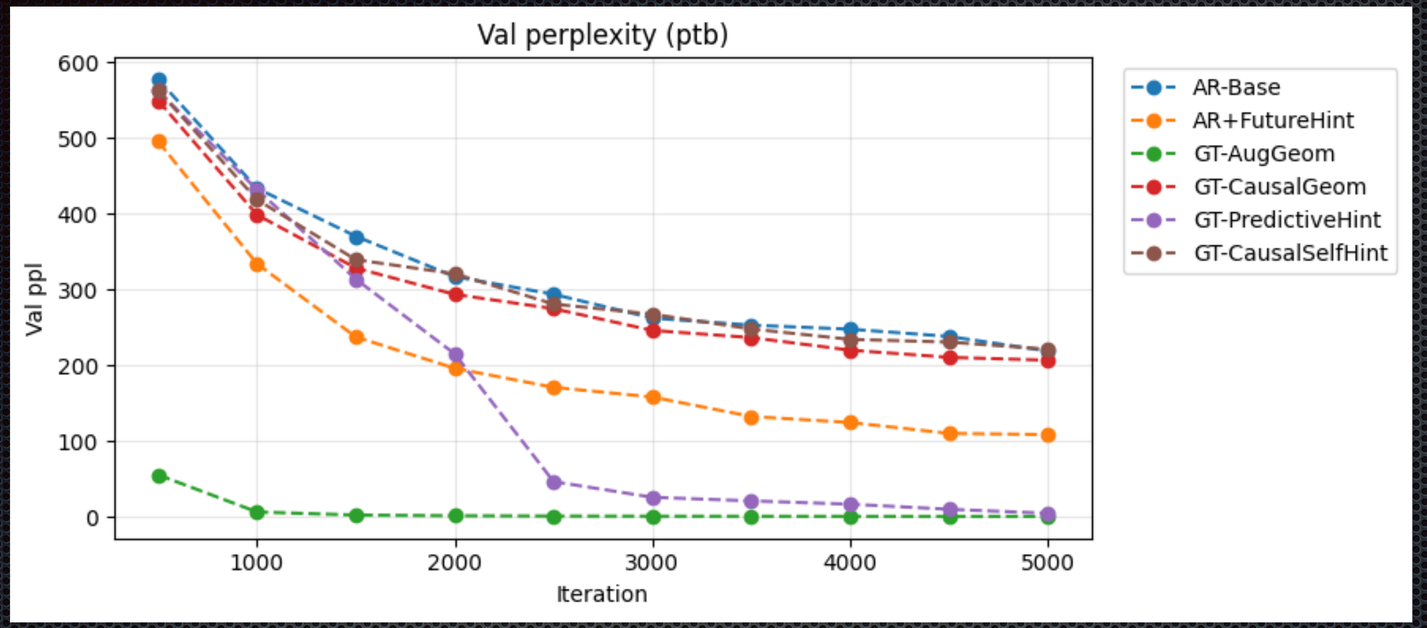
Comparing GT models

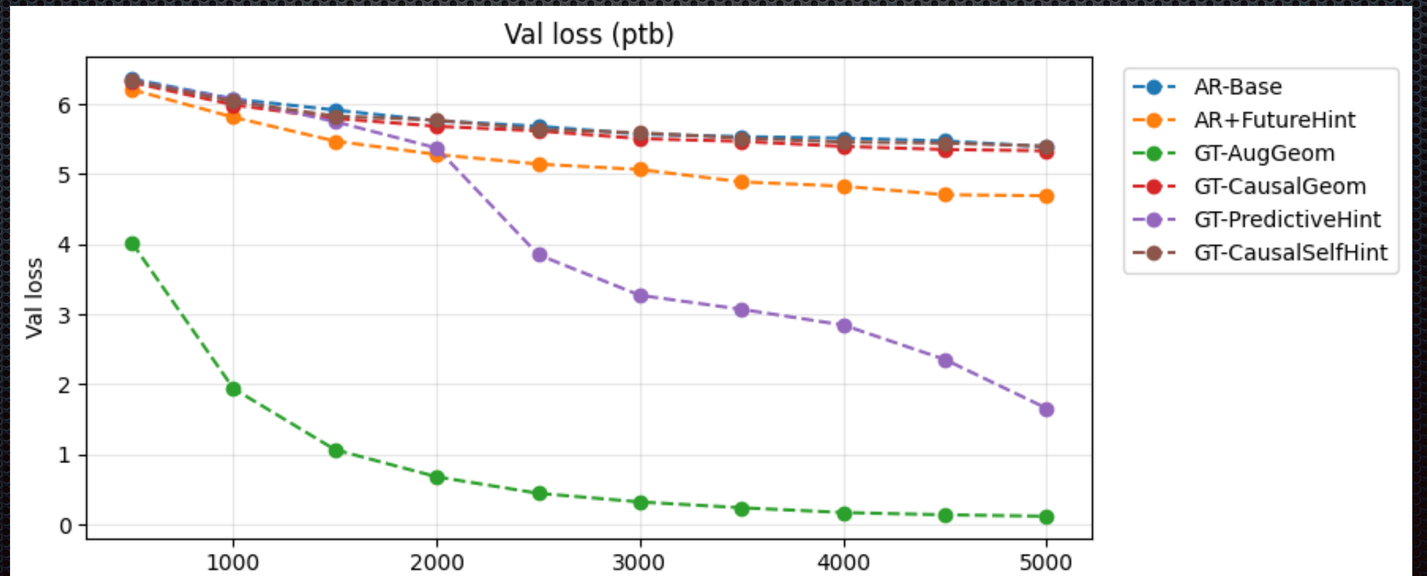
- In the last lecture, we compared a non-causal GT with a causal Transformer
  - We need to separate architecture vs. information regimes
- We introduce several novel variants of GT and baseline Transformers
  - Strict causal models: logits at position t depend only on tokens prior to and including t
  - Augmented context: future-informative side channels are allowed
  - Non-causal models: logits can depend on future tokens

Model zoo

AR-Base = autoregressive transformer, prefix-only
AR+FutureHint = transformer with explicit non-causal side hint
GT-CausalGeom = GT-Lite with strict causal geometric mixer
GT-AugGeom = GT-Lite with augmented (non-causal) geometric mixer
GT-PredictiveHint = GT-Lite with predicted next-token expectation hint
GT-CausalSelfHint = GT-Lite causal control with previous-step prediction hint

New notebook comparing GT Models



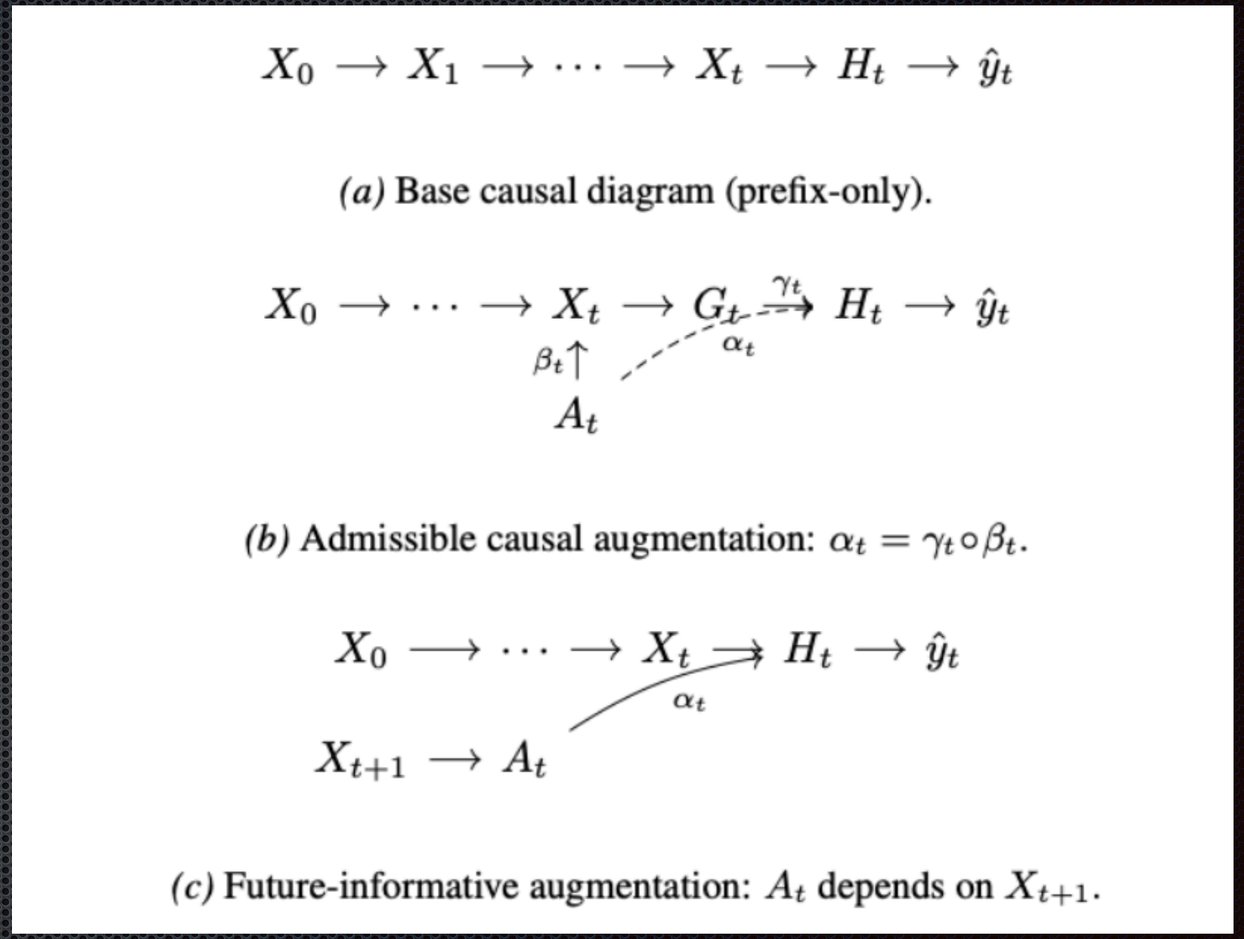


Diagrams for GT

Strict Causal

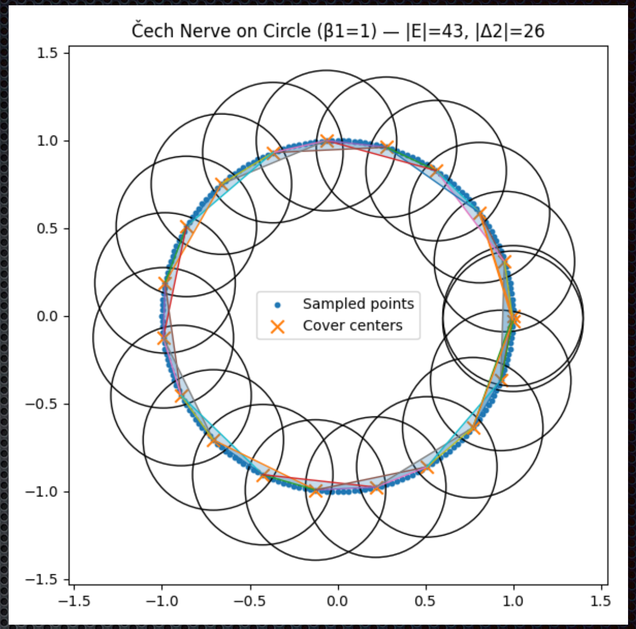
Admissible Causal

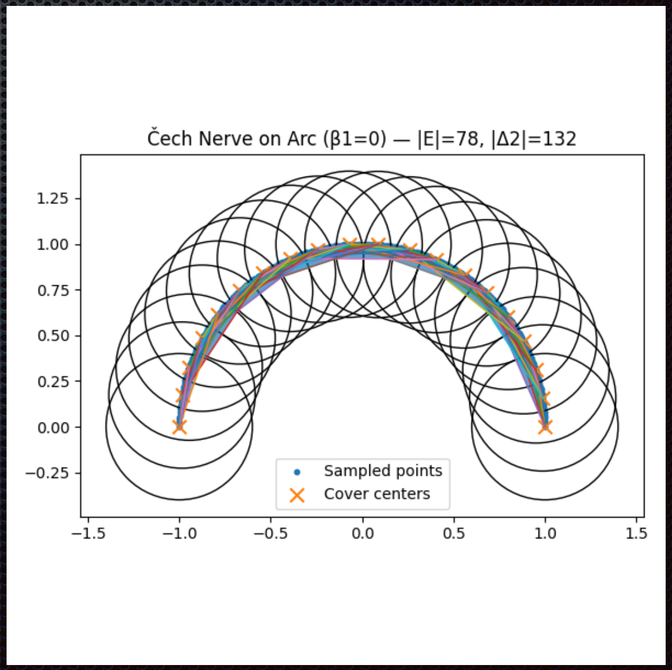
Future-informative augmented



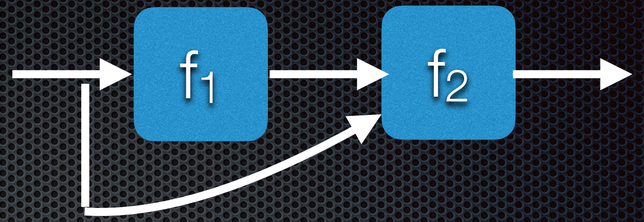
Cech Cohomology

How can we quantify whether local computations “glue” together coherently?





Dynamic Compositionality



$$
\begin{aligned}
T_2(T_1(x)) &= x + f_1(x) + f_2(x + f_1(x)), \\
T_1(T_2(x)) &= x + f_2(x) + f_1(x + f_2(x)).
\end{aligned}
$$

Static: forward computation

Dynamic: commutator energy

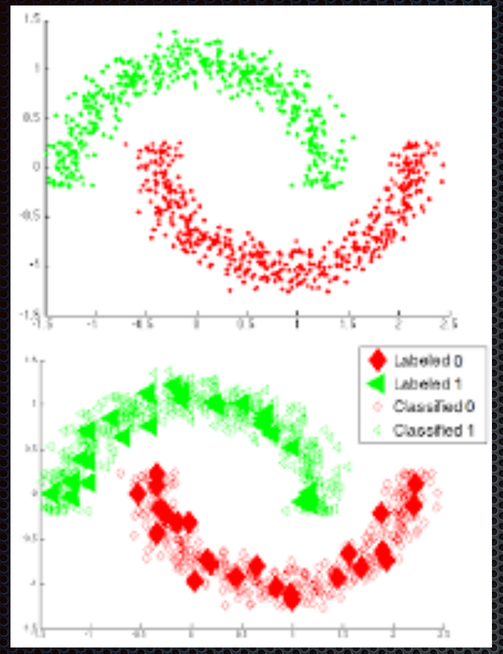
To make this notion of interaction operational, we introduce a forward-only probe that measures *order sensitivity* on representations encountered during training. Given two sub-operators T_1 and T_2, define the *commutator residual* at x as

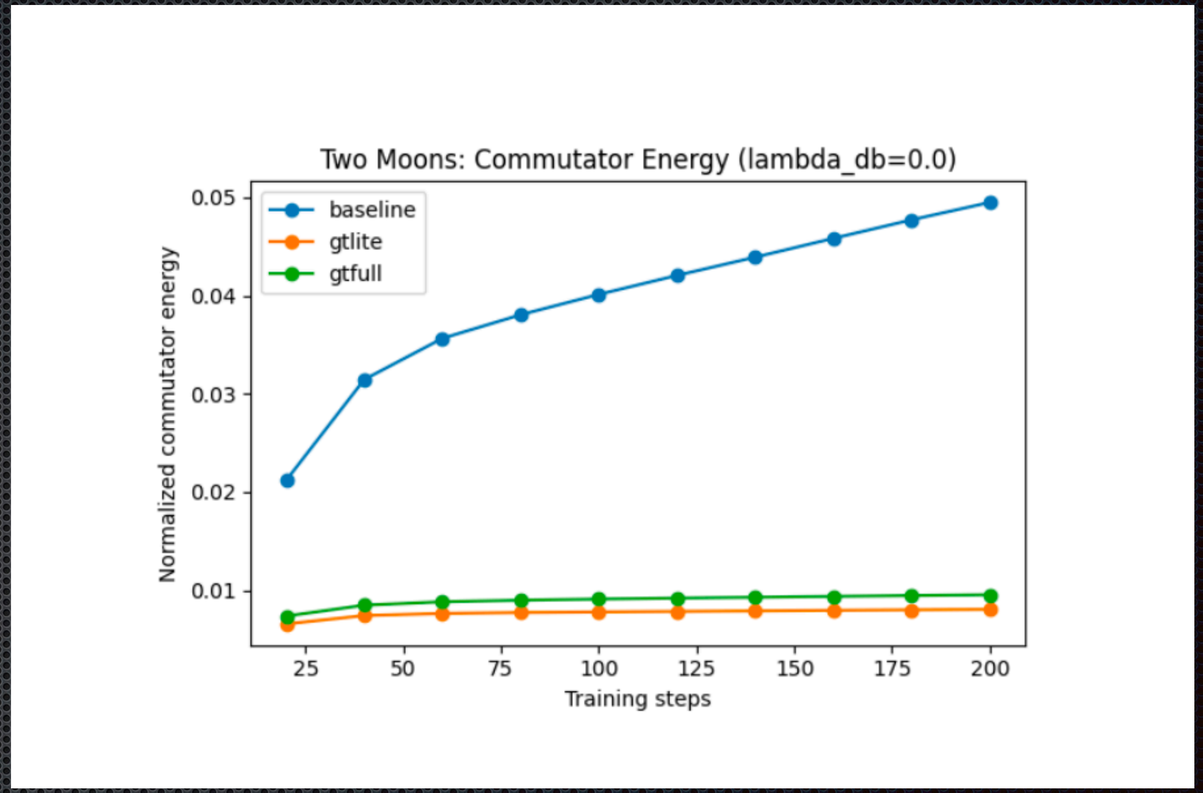
$$r(T_1, T_2; x) = T_1(T_2(x)) - T_2(T_1(x)).$$

We then define the corresponding *commutator energy* as the mean-squared magnitude of this residual, optionally normalized by the input energy:

$$E_{\text{comm}}(T_1, T_2; x) = \frac{\|r(T_1, T_2; x)\|_2^2}{\|x\|_2^2 + \varepsilon}.$$

Two Moons Dataset





What is commutator energy?

- It tracks a key failure mode of deep learning systems: gradient interference between interacting modules.
- When the representation space is such that sub-operators induce highly order-sensitive transformations, optimization signals propagated through different subpaths can conflict, producing unstable training and poor generalization.
- Conversely, when sub-operators are more compatible on the data manifold, training becomes more stable and the model can support coherent global structure.
- Dynamic compositionality is not about enforcing commutativity, but about controlling destructive order-sensitivity in learned module interactions.

Why GT + DB?

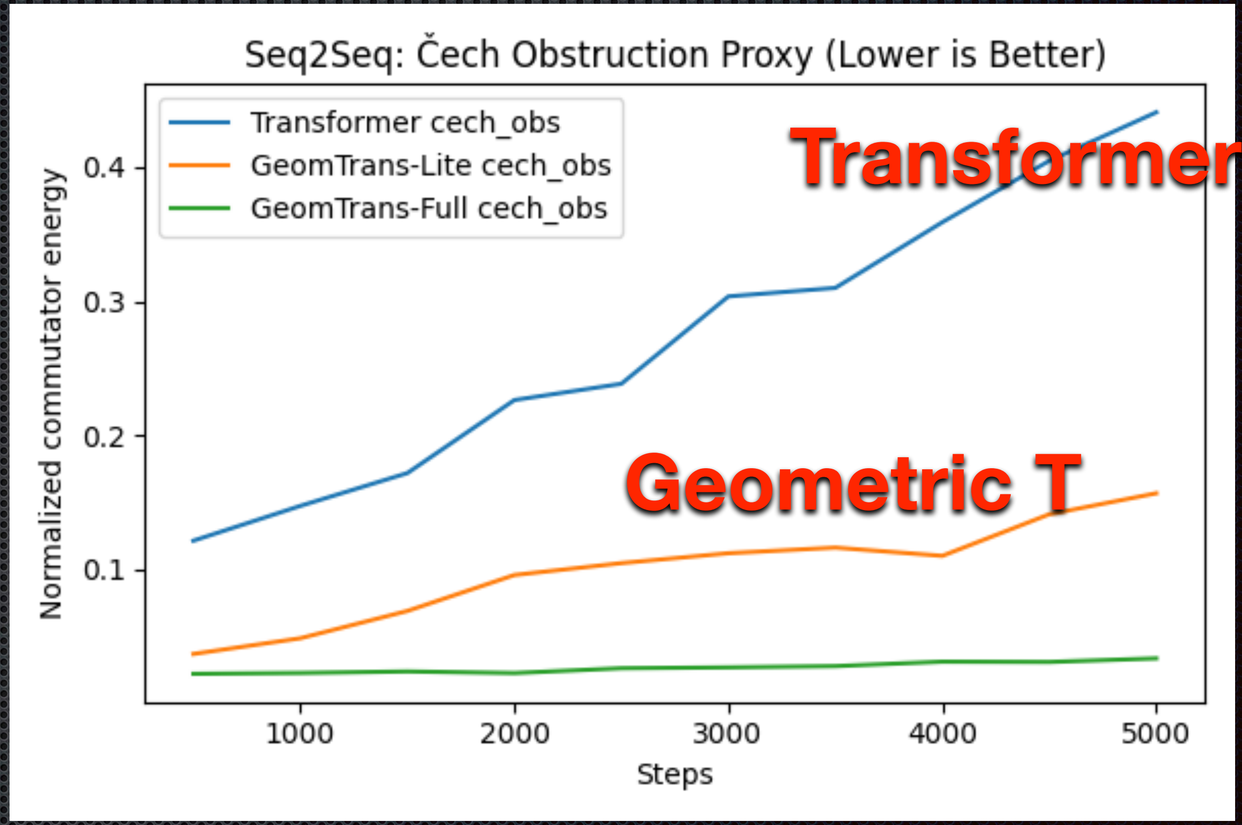
TLDR: GT with DB exploits structure in novel ways that regular Transformer with vanilla backprop does not

A new microscope

```
# GeomFullEncoderBlockSeq (Full)
elif hasattr(layer, "gt") and hasattr(layer, "edge_index") and hasattr(layer,
    "ln_attn") and hasattr(layer, "ln_ff"):
    def A(z):
        attn_out, _ = layer.self_attn(z, z, z, need_weights=False)
        return layer.ln_attn(z + layer.dropout_attn(attn_out))

    def Ff(z):
        ff = layer.lin2(layer.dropout_ff(F.relu(layer.lin1(z))))
        return layer.ln_ff(z + ff)

    def G(z):
        B, L, D = z.shape
        h = z.reshape(B * L, D)
        device = z.device
        mask = layer.edge_index[0] < L
        edge_index = layer.edge_index[:, mask].to(device)
        rel_ids = layer.rel_ids[mask].to(device)
        dom_ids = layer.dom_ids[mask].to(device)
        h_ref = layer.gt(h, edge_index, rel_ids, dom_ids)
        return h_ref.reshape(B, L, D)

    obs = (
        _commutator_energy(A, Ff, x_in) +
        _commutator_energy(Ff, G, x_in) +
        _commutator_energy(A, G, x_in)
    ) / (3.0 * denom)

    x = layer(x)
```

Transformers accumulate compositionality errors!

Wiki-103 dataset



Sudoku

	3	1	2
2			4
3			1
		4	

4x4 randomly generated problems

Mask all but k cells

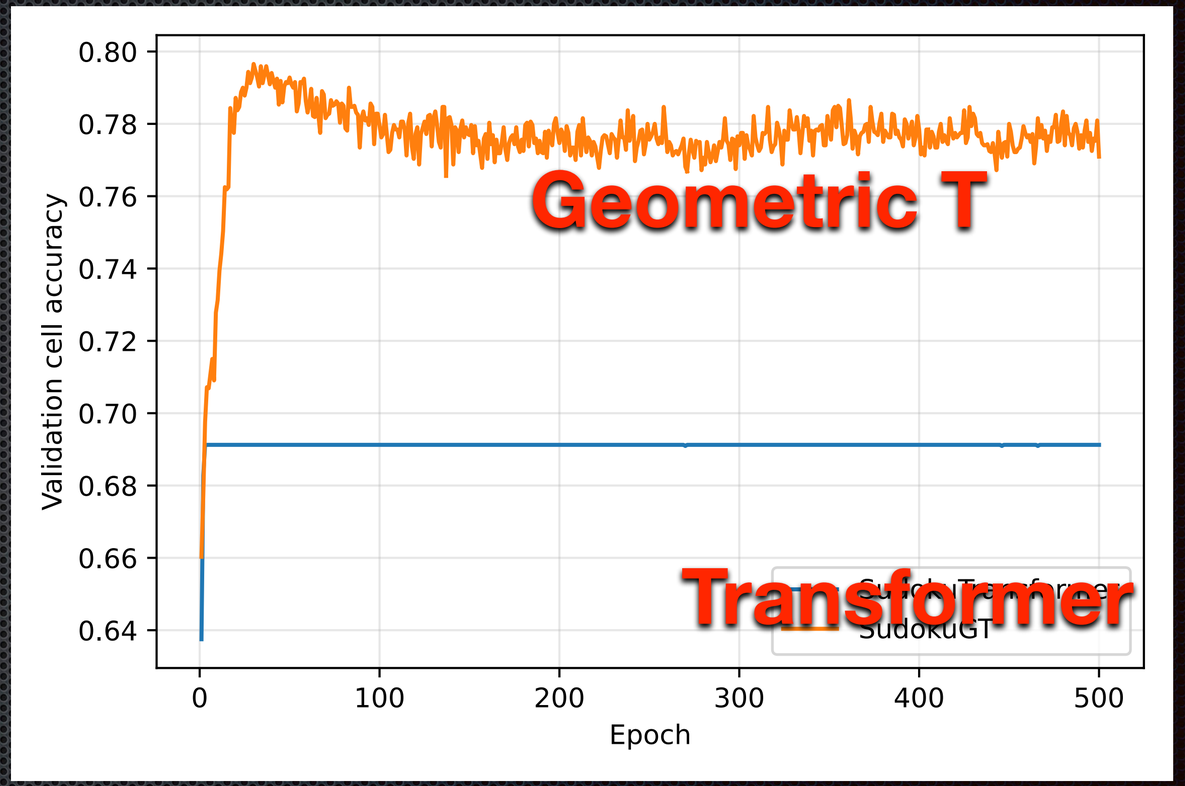
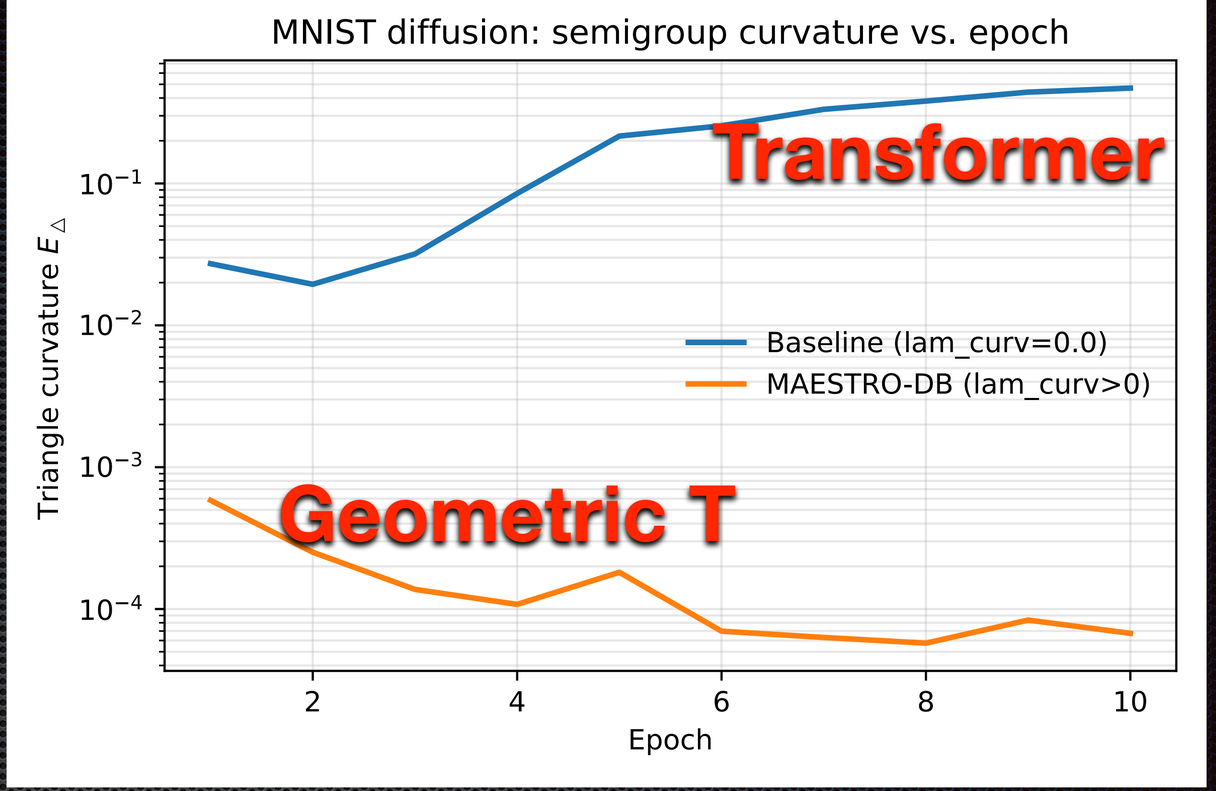


Image diffusion

DB can learn semigroup structure



M.2. Learning Diffusion Semigroup Consistency

Diffusion models provide a continuous domain in which the functorial structure of diagrammatic backpropagation can be evaluated. A diffusion process introduces a finite chain of noise levels $t = 0, 1, \ldots, T$, together with learned denoising operators $D_{t\to t-1}$. Ideally, the reverse process forms a *time-indexed semigroup*: for any $t_1 < t_2 < t_3$, one expects the compositional constraint

$$D_{t_3\to 0} \;\approx\; D_{t_2\to 0} \circ D_{t_3\to t_2}, \tag{5}$$

which is precisely a triangle diagram of the form studied in Section 2.1. A denoising diffusion probabilistic model (DDPM; Ho et al. 2020) is defined by a fixed forward noising process $q(x_t \mid x_{t-1})$ and a learned reverse denoiser $p_\theta(x_{t-1} \mid x_t)$; sampling is performed by iteratively applying the denoiser from the highest noise level to $t = 0$. However, standard DDPM training imposes no such constraint; each denoising step is learned independently, and the resulting dynamics are typically not compositionally coherent.

PreLN and PostLN Transformers

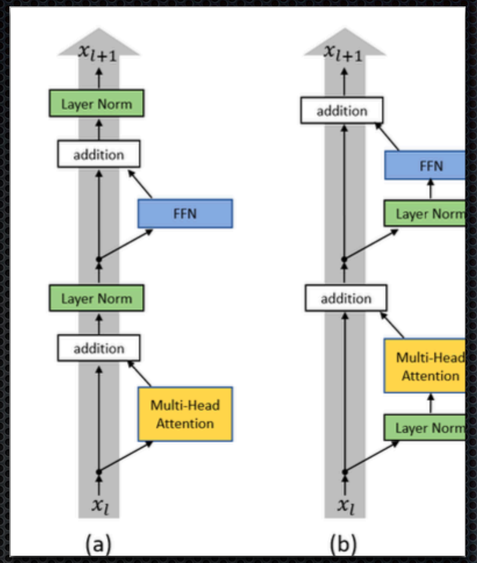
- The original Transformer (Vasvani et al., 2017) placed the layer normalization (LN) module after the self-attention layer
- PreLN Transformer does the opposite! [Xiong et al., ICML 2020]
- Why does this produce better results?
- How do GT models compare with PreLN/PostLN Transformers?

PreLN vs PostLN

The original Transformer is a PostLN architecture

It can be difficult to train

Placing the Layer Normalization before self-attention makes it easier to train



Theorem 1 (Gradients of the last layer in the Transformer). *Assume that* $\|x^{post,5}_{L,i}\|_2^2$ *and* $\|x^{pre}_{L+1,i}\|_2^2$ *are* (ϵ, δ)*-bounded for all* i*, where* ϵ *and* $\delta = \delta(\epsilon)$ *are small numbers. Then with probability at least* $0.99 - \delta - \frac{\epsilon}{0.9+\epsilon}$*, for the Post-LN Transformer with* L *layers, the gradient of the parameters of the last layer satisfies*

$$\|\frac{\partial \tilde{\mathcal{L}}}{\partial W^{2,L}}\|_F \leq \mathcal{O}(d\sqrt{\ln d})$$

and for the Pre-LN Transformer with L *layers,*

$$\|\frac{\partial \tilde{\mathcal{L}}}{\partial W^{2,L}}\|_F \leq \mathcal{O}\left(d\sqrt{\frac{\ln d}{L}}\right).$$

[Xiong et al., ICML 2020]

What is Layer Normalization doing?

Layer Normalization as a Jacobian Rescaling Operator

Layer normalization maps an input vector $x \in \mathbb{R}^d$ to

$$\mathrm{LN}(x) = \frac{x - \mu(x)\mathbf{1}}{\sigma(x)}, \tag{20}$$

where $\mu(x)$ and $\sigma(x)$ denote the mean and standard deviation of the coordinates of x. In the mean-field limit $d \to \infty$, $\mu(x)$ concentrates around zero and $\sigma(x)$ concentrates around $\sqrt{\mathbb{E}[x_i^2]}$.

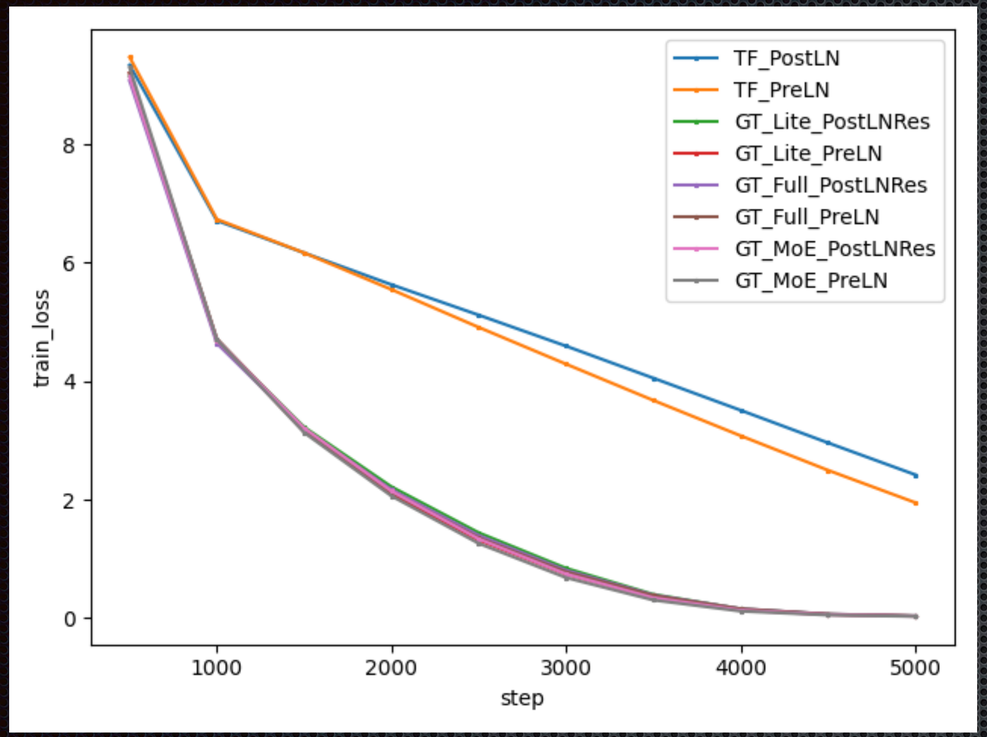
The Jacobian of LN has the form

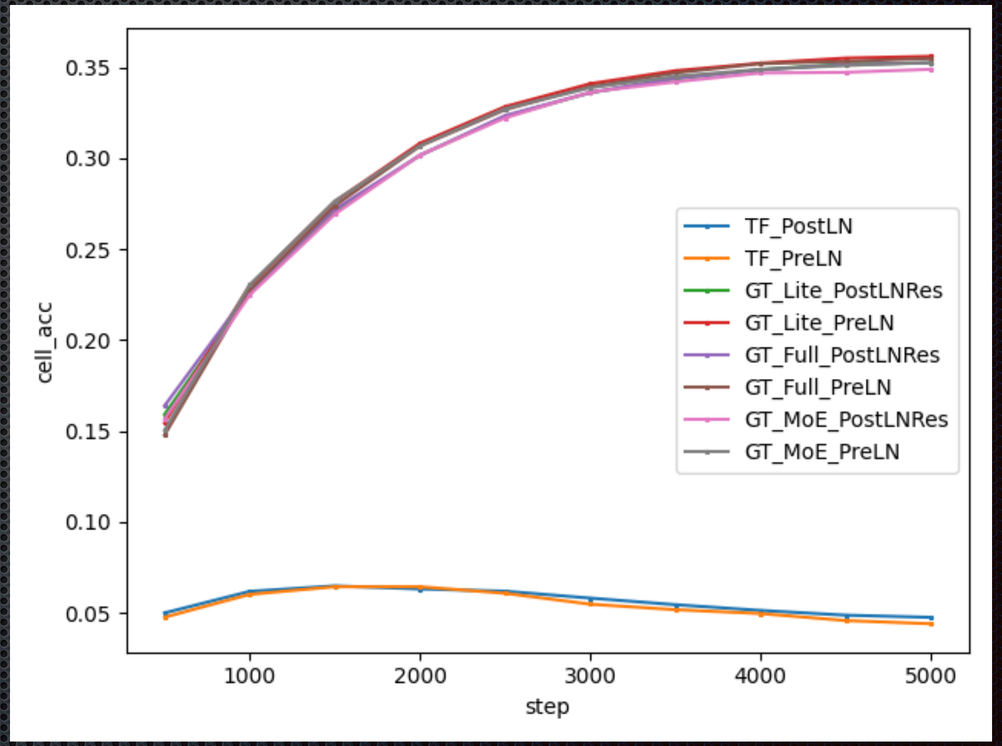
$$J_{\mathrm{LN}}(x) = \frac{1}{\sigma(x)} \left(I - \frac{1}{d}\mathbf{1}\mathbf{1}^\top - \frac{(x - \mu(x)\mathbf{1})(x - \mu(x)\mathbf{1})^\top}{d\,\sigma(x)^2} \right), \tag{21}$$

which acts approximately as an isotropic rescaling operator on directions orthogonal to $\mathbf{1}$ in high dimensions. Crucially, LN suppresses variations in the magnitude of inputs entering subsequent nonlinear transformations.

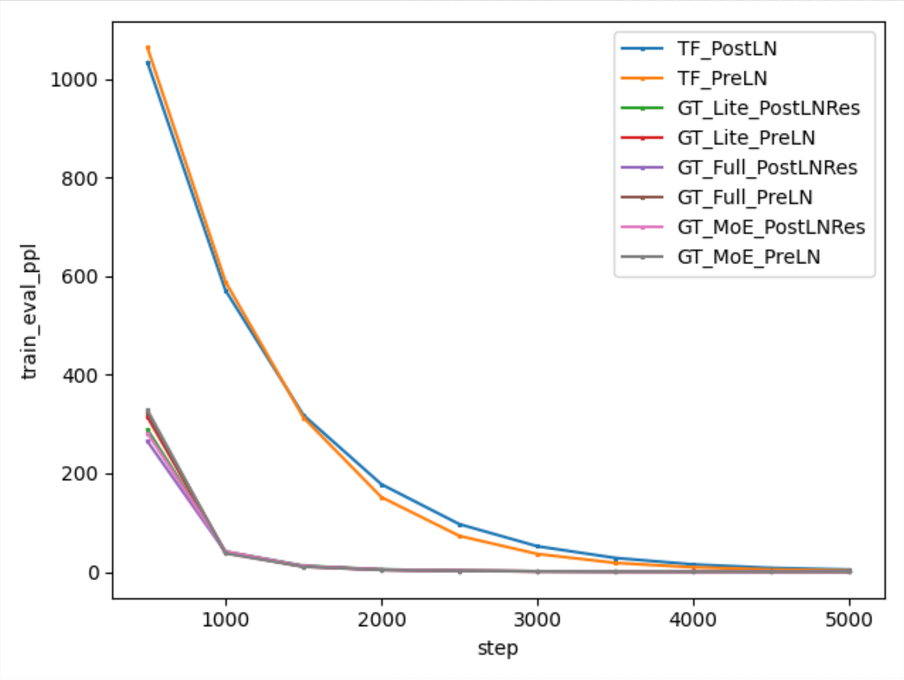
[Mahadevan, Categories for AGI, 2026]

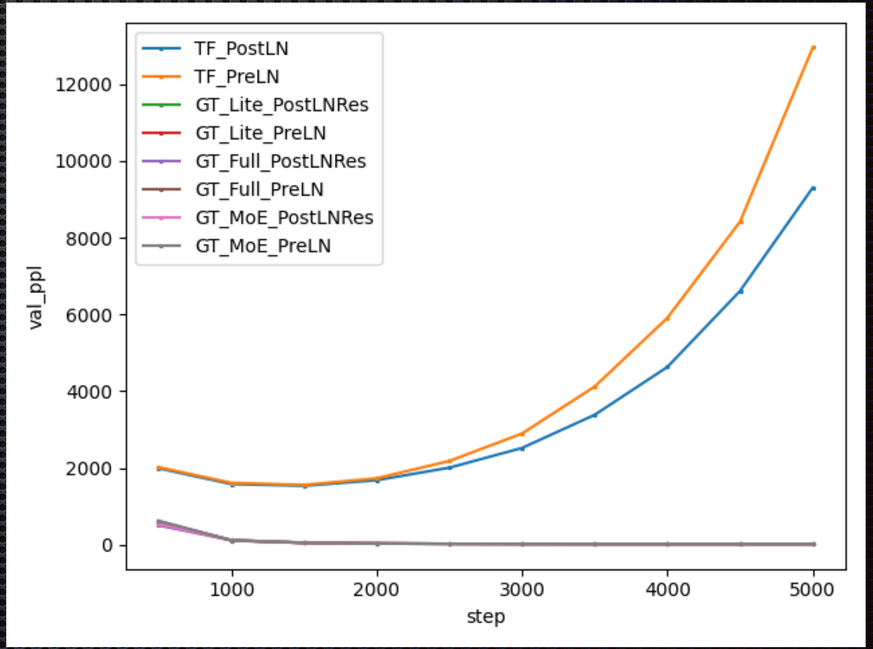
Comparing PreLN, PostLN, and GT



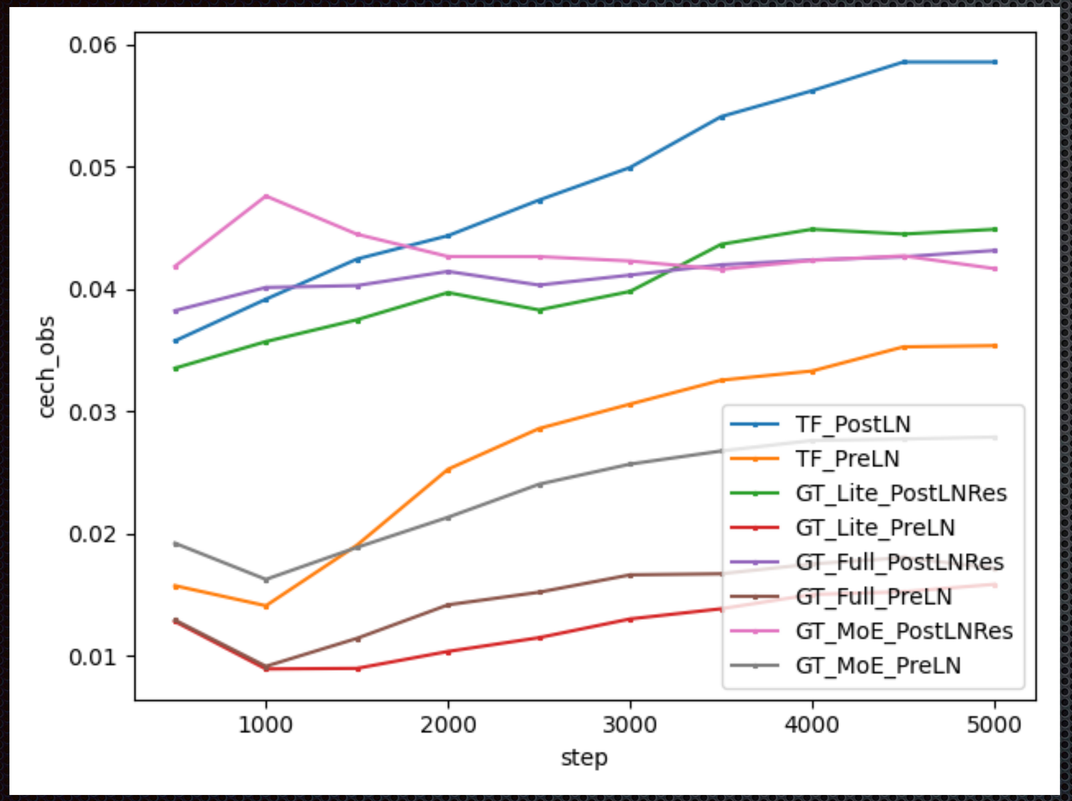


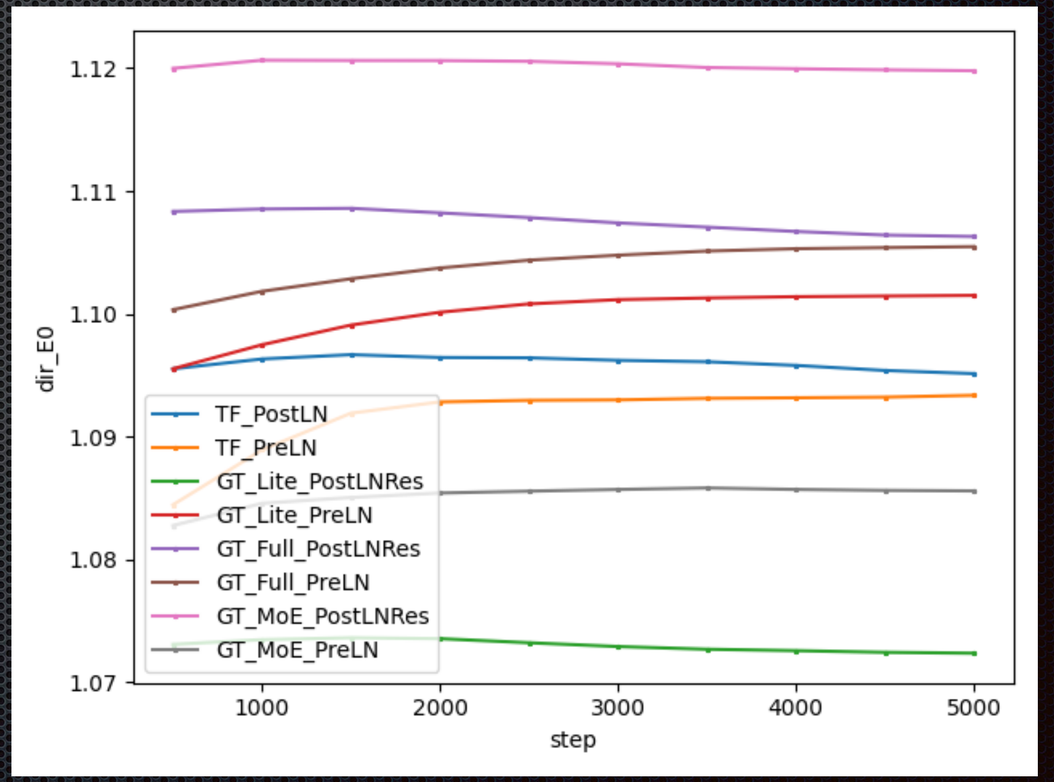
Comparing PreLN, PostLN, and GT



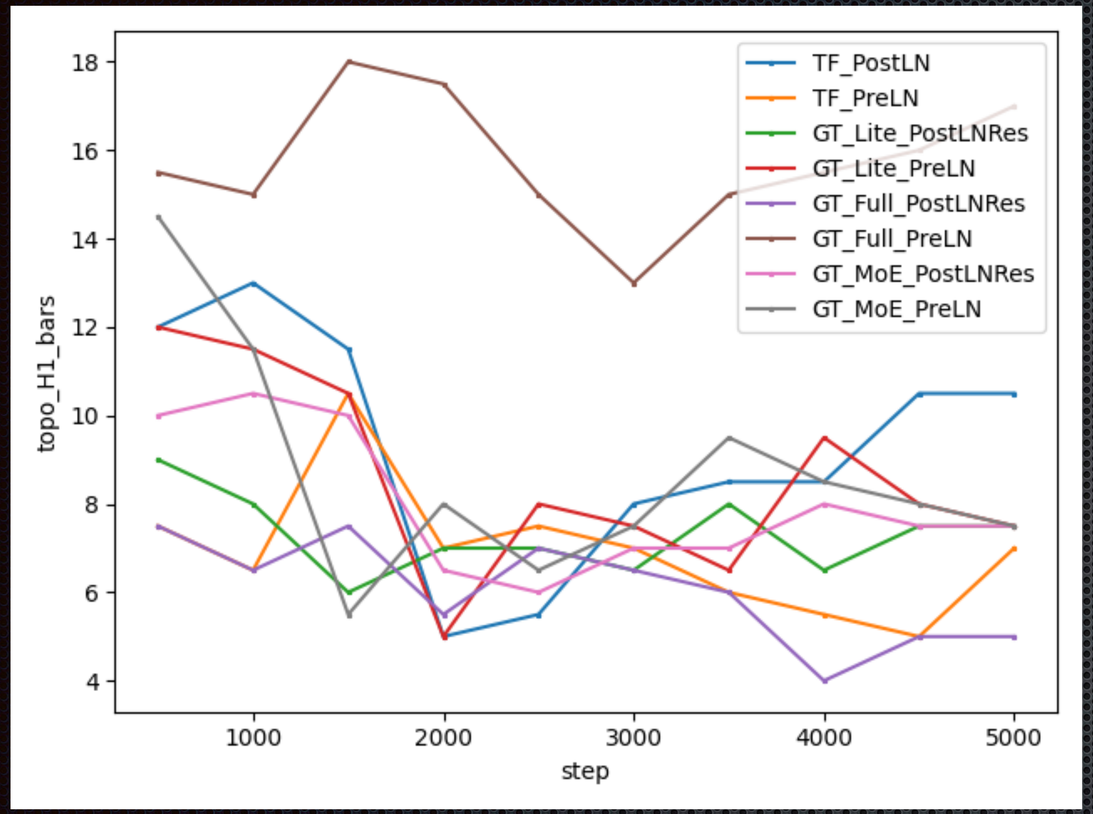


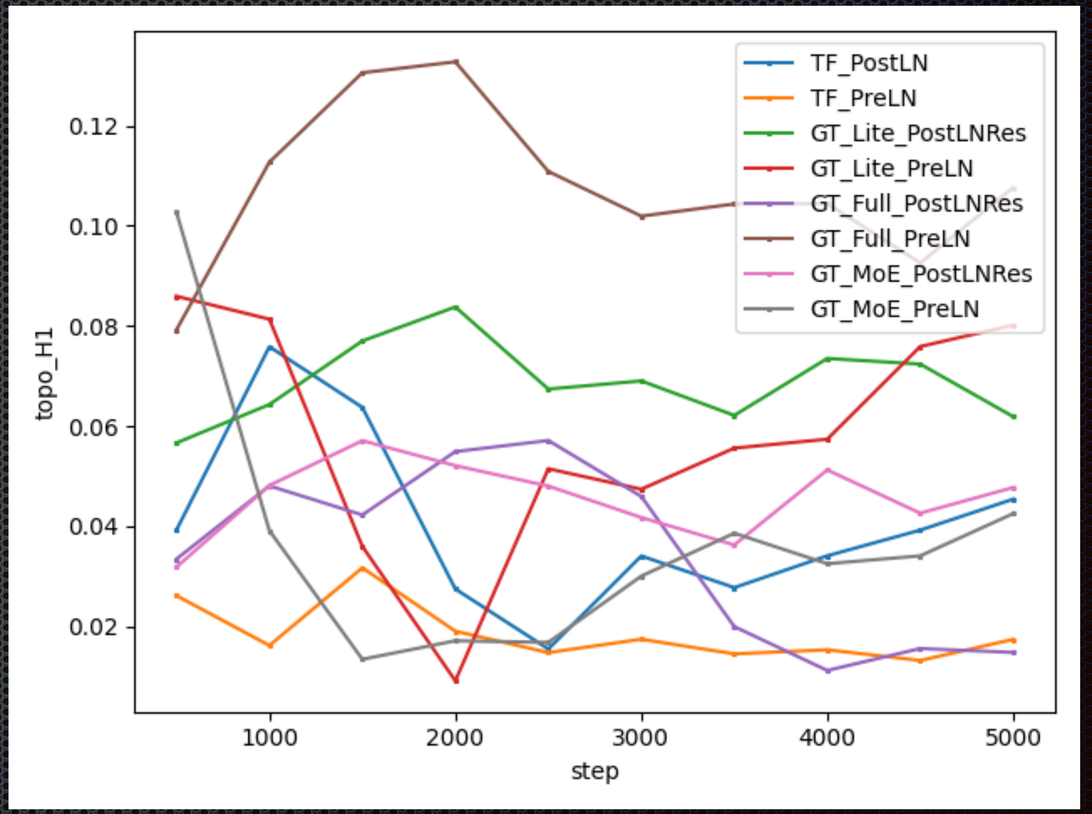
Comparing PreLN, PostLN, and GT





Comparing PreLN, PostLN, and GT





Comparing PreLN, PostLN, and GT: Wiki-103 dataset

Model	Val CE ↓	Val PPL ↓	Tok Acc ↑	Cech ↓	Dir E_{end} ↓	H_1 ↓	Bars
GT_Full_PreLN_L4	**6.536**	**689.8**	0.181	2.84e-02	**0.873**	1.88e-02	6.500
GT_MoE_PreLN_L4	6.538	690.7	**0.182**	5.54e-02	0.924	3.33e-02	11.000
GT_Lite_PreLN_L4	6.557	704.4	0.181	**2.68e-02**	0.875	7.44e-03	3.000
GT_Full_PostLNRes_L4	6.611	743.2	0.179	4.73e-02	0.993	1.68e-02	5.000
GT_Lite_PostLNRes_L4	6.630	757.6	0.174	4.54e-02	0.937	4.07e-02	10.000
GT_MoE_PostLNRes_L4	6.697	810.3	0.176	4.85e-02	1.085	7.95e-04	1.500
TF_PreLN_L4	8.976	7908	0.035	4.49e-02	1.009	7.54e-03	4.000
TF_PostLN_L4	9.105	9000	0.036	5.74e-02	1.053	1.77e-03	1.500

[Mahadevan, Categories for AGI, 2026]

Listing 1: GT-Lite encoder block

```python
class GeomEncoderBlock(nn.Module):
    def __init__(self, d_model: int, n_heads: int, dim_ff: int, dropout: float = 0.1):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(
            d_model, n_heads, dropout=dropout, batch_first=True
        )
        self.conv = nn.Conv1d(d_model, d_model, kernel_size=3, padding=1)

        self.lin1 = nn.Linear(d_model, dim_ff)
        self.lin2 = nn.Linear(dim_ff, d_model)

        self.ln_attn = nn.LayerNorm(d_model)
        self.ln_conv = nn.LayerNorm(d_model)
        self.ln_ff   = nn.LayerNorm(d_model)

        self.dropout = nn.Dropout(dropout)
        self.dropout_ff = nn.Dropout(dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        attn_out, _ = self.self_attn(x, x, x, need_weights=False)
        x = self.ln_attn(x + self.dropout(attn_out))

        x_conv = self.conv(x.transpose(1, 2)).transpose(1, 2)
        x = self.ln_conv(x + 0.2 * x_conv)

        ff = self.lin2(self.dropout_ff(F.relu(self.lin1(x))))
        x = self.ln_ff(x + ff)
        return x
```

[Mahadevan, Categories for AGI, 2026]

[Mahadevan, Categories for AGI, 2026]

Sub-operators and dynamic compositionality

From the perspective of dynamic compositionality, the GT-Lite encoder block consists of three interacting sub-operators, each applied with a residual connection and layer normalization:

- **Attention operator** A: global token mixing via multi-head self-attention,

$$A(x) = \mathrm{LN}_{attn}(x + \mathrm{Attn}(x)).$$

- **Local smoothing operator** C: convolution over neighboring token positions,

$$C(x) = \mathrm{LN}_{conv}(x + \alpha\,\mathrm{Conv}(x)), \quad \alpha = 0.2.$$

- **Feed-forward operator** F: positionwise nonlinear deformation,

$$F(x) = \mathrm{LN}_{ff}(x + \mathrm{FF}(x)).$$

The overall block computes $F(C(A(x)))$. Architecturally, this is a fixed composition. Dynamically, however, learning depends on how strongly these sub-operators interfere when applied to the same representation.

Why local smoothing reduces commutator energy

The commutator-energy probe evaluates order sensitivity between sub-operators, for example

$$E_{\mathrm{comm}}(A, F; x) = \|A(F(x)) - F(A(x))\|_2^2.$$

In a baseline Transformer, A and F interact directly. Attention introduces sharp, token-dependent changes to the representation, and the feed-forward network is highly sensitive to such changes. As a result, $A(F(x))$ and $F(A(x))$ often differ substantially, producing large commutator energy.

GT-Lite reduces this effect by inserting the smoothing operator C. The convolution averages local neighborhoods in token space, acting as a low-pass filter on representation geometry. This regularization reduces the sensitivity of downstream sub-operators to small perturbations induced by upstream ones. Consequently, the pairwise commutator energies

$$E_{\mathrm{comm}}(A, C; x), \quad E_{\mathrm{comm}}(C, F; x), \quad E_{\mathrm{comm}}(A, F; x)$$

are all reduced relative to the baseline Transformer.

Listing 2: GT-Lite decoder block

```
class GeomDecoderBlock(nn.Module):
    def __init__(self, d_model: int, n_heads: int, dim_ff: int, dropout: float
        super().__init__()
        self.self_attn = nn.MultiheadAttention(
            d_model, n_heads, dropout=dropout, batch_first=True
        )
        self.cross_attn = nn.MultiheadAttention(
            d_model, n_heads, dropout=dropout, batch_first=True
        )
        self.conv = nn.Conv1d(d_model, d_model, kernel_size=3, padding=1)

        self.lin1 = nn.Linear(d_model, dim_ff)
        self.lin2 = nn.Linear(dim_ff, d_model)

        self.ln_self  = nn.LayerNorm(d_model)
        self.ln_cross = nn.LayerNorm(d_model)
        self.ln_conv  = nn.LayerNorm(d_model)
        self.ln_ff    = nn.LayerNorm(d_model)

        self.dropout = nn.Dropout(dropout)
        self.dropout_ff = nn.Dropout(dropout)

    def forward(self, x, enc_out, causal_mask):
        self_out, _ = self.self_attn(x, x, x, attn_mask=causal_mask, need_weigh
        x = self.ln_self(x + self.dropout(self_out))

        cross_out, _ = self.cross_attn(x, enc_out, enc_out, need_weights=False)
        x = self.ln_cross(x + self.dropout(cross_out))

        x_conv = self.conv(x.transpose(1, 2)).transpose(1, 2)
        x = self.ln_conv(x + 0.2 * x_conv)

        ff = self.lin2(self.dropout_ff(F.relu(self.lin1(x))))
        x = self.ln_ff(x + ff)
        return x
```

[Mahadevan, Categories for AGI, 2026]

```python
class GeomFullEncoderBlockSeq(nn.Module):
    def __init__(
        self,
        d_model: int,
        n_heads: int,
        dim_ff: int,
        max_len_src: int,
        num_rel: int = 1,
        dropout: float = 0.1,
        gt_depth: int = 1,
    ):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(
            embed_dim=d_model,
            num_heads=n_heads,
            dropout=dropout,
            batch_first=True,
        )
        self.lin1 = nn.Linear(d_model, dim_ff)
        self.lin2 = nn.Linear(dim_ff, d_model)

        self.ln_attn = nn.LayerNorm(d_model)
        self.ln_ff   = nn.LayerNorm(d_model)

        self.dropout_attn = nn.Dropout(dropout)
        self.dropout_ff   = nn.Dropout(dropout)

        self.gt = GeometricTransformerV2(
            dim=d_model,
            depth=gt_depth,
            num_rel=num_rel,
        )

        # fixed positional graph over sequence indices
        src_indices, dst_indices = [], []
        for i in range(max_len_src - 1):
            src_indices.extend([i, i + 1])
            dst_indices.extend([i + 1, i])
        edge_index = torch.tensor([src_indices, dst_indices], dtype=torch.long)

        self.register_buffer("edge_index", edge_index)
        self.register_buffer("rel_ids", torch.zeros(edge_index.size(1), dtype=torch.long))
        self.register_buffer("dom_ids", torch.zeros(edge_index.size(1), dtype=torch.long))
```

[Mahadevan, Categories for AGI, 2026]

```python
class GeomEncoderMoEBlock(nn.Module):
    def __init__(
        self,
        d_model: int,
        n_heads: int,
        dim_ff: int,
        n_experts: int = 4,
        dropout: float = 0.1,
        top_k: int = 2,
    ):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(
            d_model, n_heads, dropout=dropout, batch_first=True
        )

        self.experts = nn.ModuleList([
            nn.Sequential(
                nn.Linear(d_model, dim_ff),
                nn.GELU(),
                nn.Linear(dim_ff, d_model),
            )
            for _ in range(n_experts)
        ])

        self.gate = nn.Linear(d_model, n_experts)

        self.ln_attn = nn.LayerNorm(d_model)
        self.ln_ff   = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        attn_out, _ = self.self_attn(x, x, x, need_weights=False)
        x = self.ln_attn(x + self.dropout(attn_out))

        gate_logits = self.gate(x)
        gate_probs = gate_logits.softmax(dim=-1)

        if self.top_k < gate_probs.size(-1):
            top_vals, top_idx = gate_probs.topk(self.top_k, dim=-1)
            sparse = torch.zeros_like(gate_probs)
            sparse.scatter_(2, top_idx, top_vals)
            gate_probs = sparse / (sparse.sum(dim=-1, keepdim=True) + 1e-8)

        ff_out = torch.zeros_like(x)
        for e_idx, expert in enumerate(self.experts):
            w_e = gate_probs[..., e_idx].unsqueeze(-1)
            ff_out = ff_out + w_e * expert(x)
```

[Mahadevan, Categories for AGI, 2026]

Mean-Field Theory of GT

Definition of Commutator Energy

Given two residual blocks

$$T_i(x) = x + \Delta_i(x), \qquad T_j(x) = x + \Delta_j(x),$$

we define the commutator residual at input x as

$$R_{ij}(x) \;=\; T_i(T_j(x)) - T_j(T_i(x)). \tag{12}$$

The commutator energy is then given by

$$\mathcal{E}_{ij}(x) \;=\; \mathbb{E}\big[\|R_{ij}(x)\|_2^2\big], \tag{13}$$

where the expectation is taken over both input randomness and random initialization of the network parameters.

Exact commutativity corresponds to $\mathcal{E}_{ij}(x) \;=\; 0$. Our objective is to understand when $\mathcal{E}_{ij}(x)$ is generically nonzero and how it scales in the mean-field limit.

First-Order Expansion

Expanding the compositions explicitly yields

$$T_i(T_j(x)) = x + \Delta_j(x) + \Delta_i\big(x + \Delta_j(x)\big), \tag{14}$$
$$T_j(T_i(x)) = x + \Delta_i(x) + \Delta_j\big(x + \Delta_i(x)\big). \tag{15}$$

Subtracting gives

$$R_{ij}(x) = \Delta_i\big(x + \Delta_j(x)\big) - \Delta_j\big(x + \Delta_i(x)\big) + \Delta_j(x) - \Delta_i(x). \tag{16}$$

[Mahadevan, Categories for AGI, 2026]

Under the mean-field assumption that $\Delta_i(x)$ and $\Delta_j(x)$ are $\mathcal{O}(1)$ but small relative to x in the large-width limit, we linearize Δ_i and Δ_j around x:

$$\Delta_i(x + \Delta_j(x)) \approx \Delta_i(x) + J_i(x)\,\Delta_j(x),$$

where $J_i(x) = \nabla_x \Delta_i(x)$ is the Jacobian.

Substituting and simplifying, first-order terms cancel and we obtain

$$R_{ij}(x) \;\approx\; J_i(x)\,\Delta_j(x) - J_j(x)\,\Delta_i(x). \tag{17}$$

This expression shows that commutator energy arises from the mismatch between how each block responds to the other's perturbation.

Jacobian Commutator Approximation

Using $\Delta_k(x) \approx J_k(x)\,x$ at initialization, Equation (17) can be further approximated as

$$R_{ij}(x) \;\approx\; \big(J_i(x)J_j(x) - J_j(x)J_i(x)\big)\,x. \tag{18}$$

Thus, up to first order, commutator energy measures the non-commutativity of the Jacobian matrices associated with distinct residual blocks.

[Mahadevan, Categories for AGI, 2026]

Geometric Transport

Geometric Transport in GT Architectures

GT architectures implement geometric alignment through explicit transport operators that propagate representations across a fixed relational or positional structure. In the simplest case (GT-Lite), local smoothing encourages nearby representations to evolve coherently, partially aligning the directions of subsequent transformations. In the full GT architecture, message passing over a graph or simplicial complex enforces consistency across overlapping neighborhoods of representations. Importantly, alignment does not require sub-operators to be equal or commute exactly; it only requires that their induced deformations act in compatible directions on the data manifold.

From a Jacobian perspective, geometric transport introduces coupling terms that bias $\nabla\Delta_i(x)$ toward acting along shared subspaces across blocks. This reduces directional mismatch between J_i and J_j, suppressing the commutator even when normalization alone is insufficient.

Mean-Field Limit of GT

Mean-Field Interpretation of Alignment

In the mean-field limit, geometric alignment can be interpreted as reducing the variance of off-diagonal terms in the joint distribution of Jacobians. Whereas Pre-LN reduces the variance of individual Jacobians, GT-style alignment reduces their *cross-covariance*:

$$\mathrm{Cov}(J_i, J_j) \;\to\; \text{aligned}.$$

As a result, the expected Frobenius norm of the commutator decreases further:

$$\mathbb{E}\left[\|J_i J_j - J_j J_i\|_F^2\right] \;\ll\; \mathbb{E}\left[\|J_i^{\mathrm{Pre}} J_j^{\mathrm{Pre}} - J_j^{\mathrm{Pre}} J_i^{\mathrm{Pre}}\|_F^2\right].$$

This qualitative inequality aligns with empirical observations across residual MLPs, sequence-to-sequence models, and language modeling benchmarks.

Normalization vs. Alignment

Normalization alone cannot enforce approximate commutativity unless the dominant eigenspaces of distinct Jacobians are already aligned by chance. The distinction between normalization and alignment clarifies the relationship between Pre-LN Transformers and GT architectures:

Summary

- We analyzed the reasons for why Geometric Transformers work well
  - They control commutator energy better than PreLN Transformers
  - We did a mean-field analysis showing theoretically their strengths

Further Reading

- The textbook Categories for AGI has several chapters devoted to this week's topics
  - Chapters 5-10
  - It is the longest section in the book!
- We will study these concepts further in the coming weeks