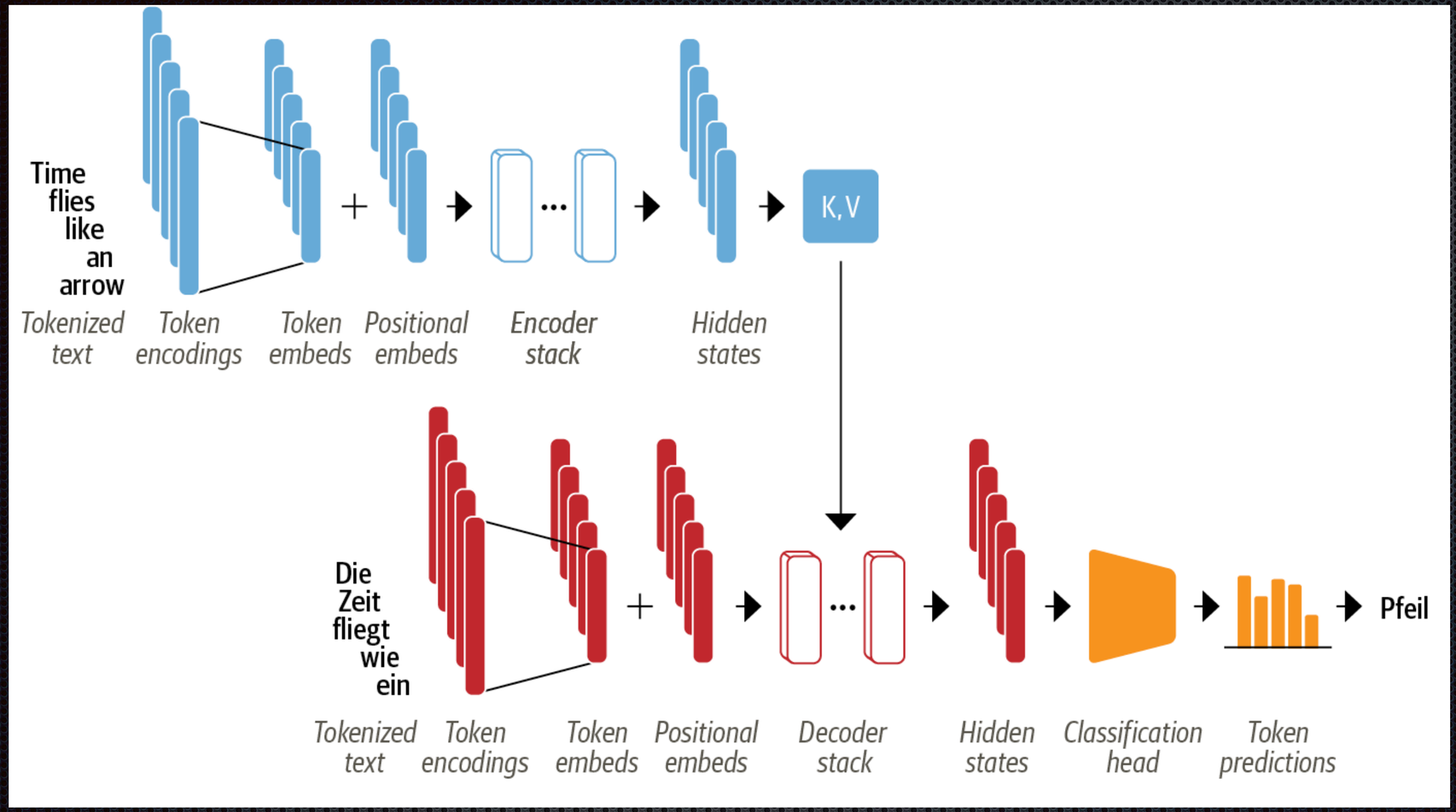
Kan Extension and TopoCoend Transformers: Categorical Attention Models

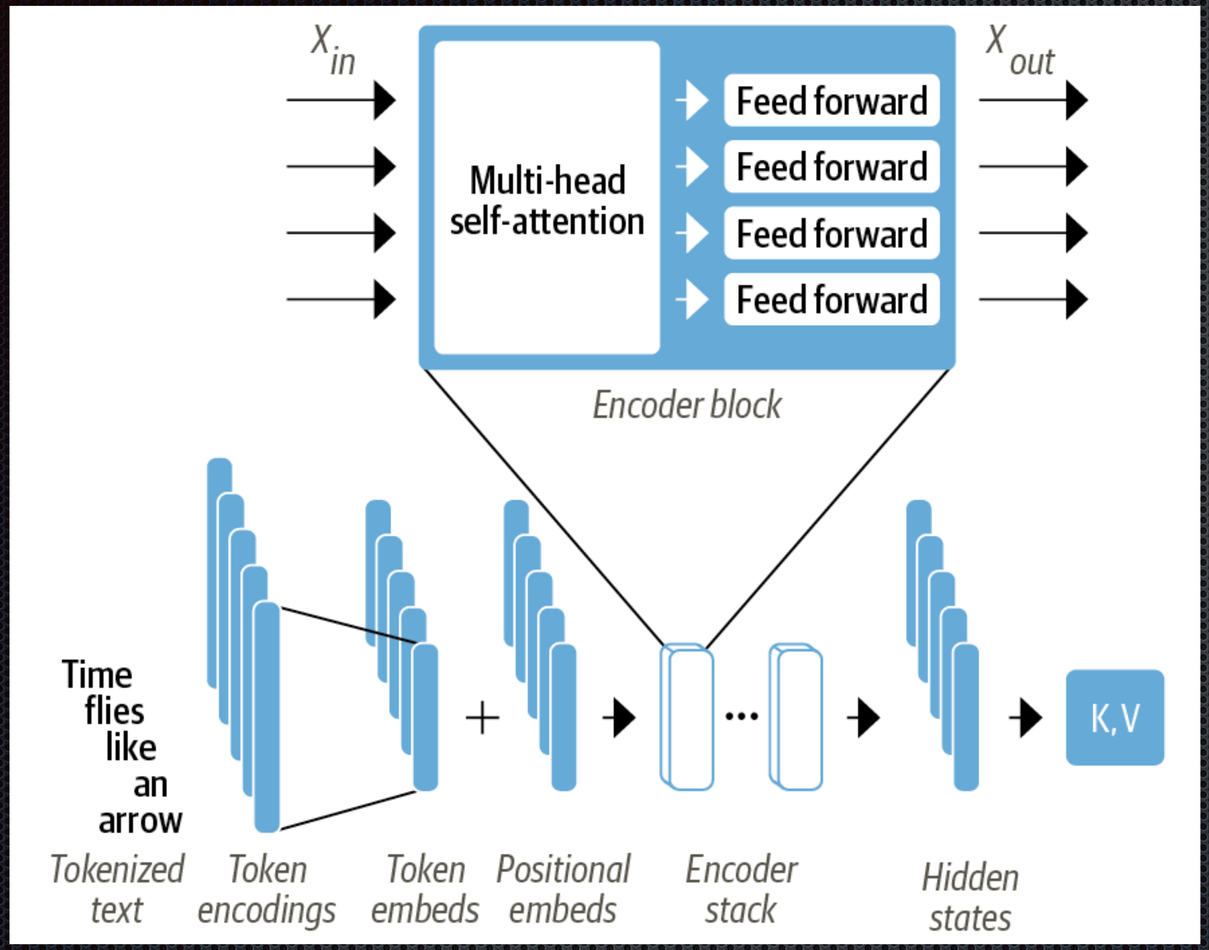
Sridhar Mahadevan, Adobe Research and U.Mass, Amherst

Unpacking the Transformer



https://github.com/nlp-with-transformers/notebooks/blob/main/03_transformer-anatomy.ipynb

Multi-head Attention

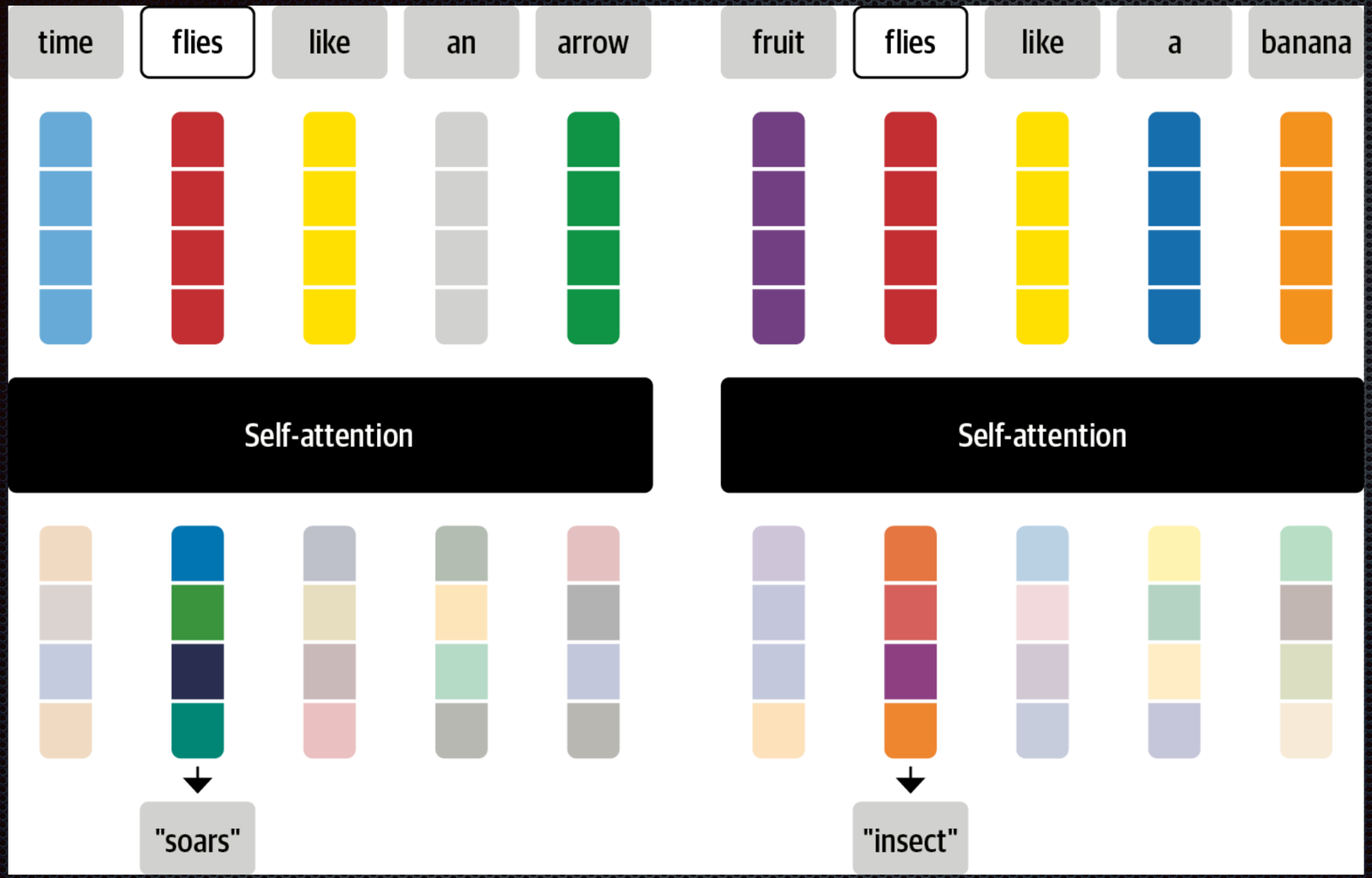


https://github.com/nlp-with-transformers/notebooks/blob/main/03_transformer-anatomy.ipynb

Contrasting Approaches

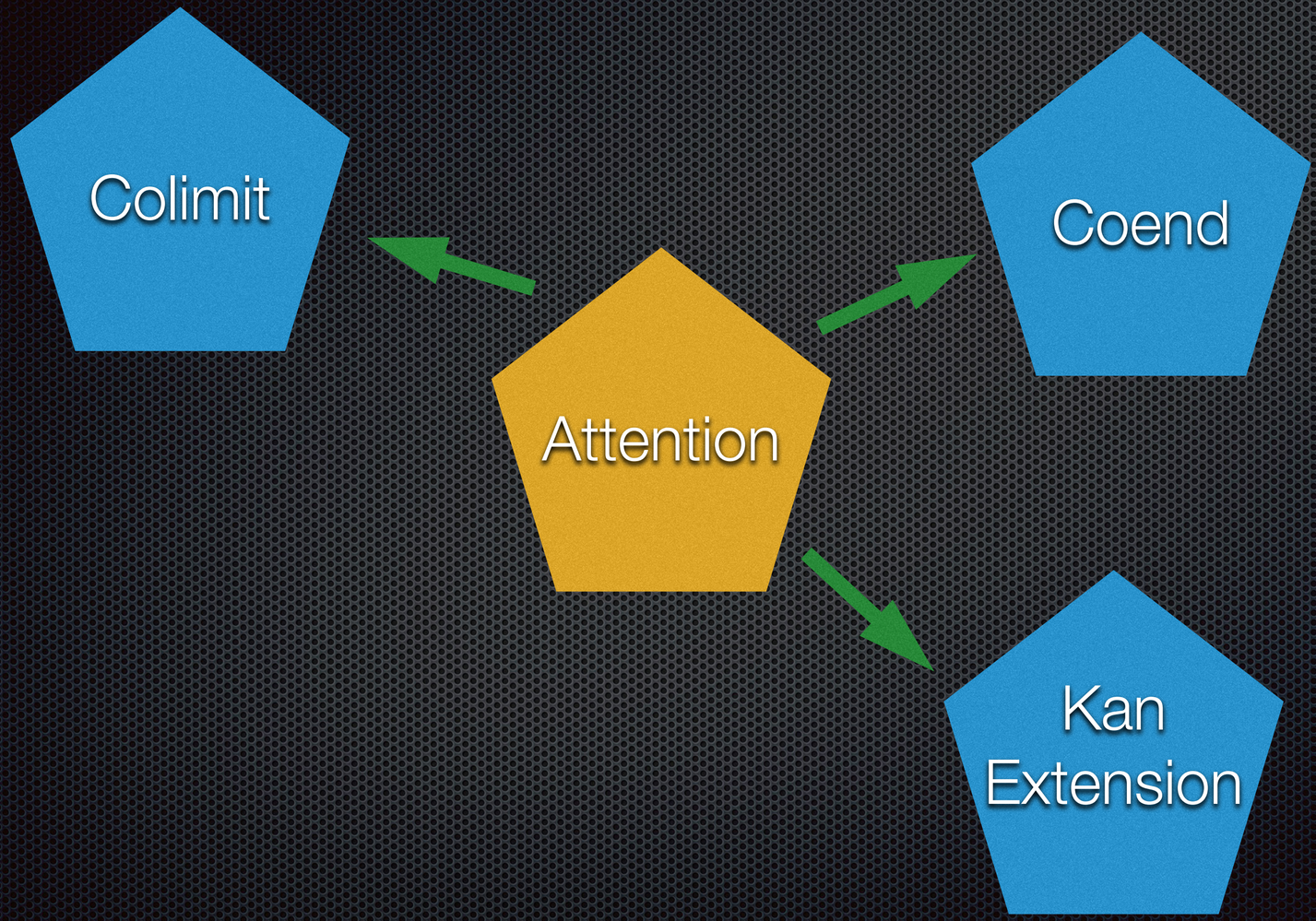
Model	Attention Weights
Transformer	Dot Product Similarity
Geometric Transformer	Learned Geometric Convolution Kernel
Kan Extension Transformer	Learned Morphism Kernel
TopoCoend Transformer	Geometric Neighborhood Kernel

Resolving ambiguity



https://github.com/nlp-with-transformers/notebooks/blob/main/03_transformer-anatomy.ipynb

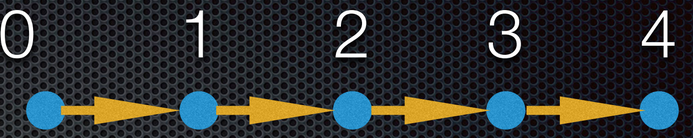
Categorical Attention



A categorical attentional mechanism

Traditional Transformer: weighted sum over tokens

KET/TopoCoend: colimit or coend over allowable morphisms



Prefix category:

Arrow between s, t if $s \leq t$

Kan Attention: compute colimit

over allowable morphisms

TopoCoend: compute coend

Simplicial Indices Functor

- Let the token window be positions $t \in \{0, 1, \ldots, S-1\}$
- Indexing category I
  - Objects are 0-simplices (tokens) and 1-simplices (adjacent tokens)
  - Morphisms encode adjacency
- Define a functor F: I -> T
  - T is a discrete category of token positions

Attention is a colimit

- Whenever you see an expression like this $\sum_c w(c,t)F(c)$
  - *w(c,t)* tells us how much token *c* contributes towards token *t*
  - You should interpret that as a colimit
- But we don’t have to take the colimit in Vect!
  - We can compute colimits in any category (e.g., topological spaces!)
  - We can design a TopoCoend or a Kan Extension Transformer

KET Quad vs. TopoCoend

- KET Quad Transformer aggregates over all simplices with a learned kernel
  - $m_t = \sum_{\sigma} w(t, \sigma) V(\sigma)$
  - Global structured attention
- TopoCoend Transformer aggregates over k-nearest neighbors
  - $m_t = \sum_{s \in N_k(t)} w(t, s) V(s)$

Intuition for what a coend is

- Given some functor F, where values F(c) are known
- We are given “weights” W(c,t), of how much c contributes to t
- Then, coend(t) ~ $\sum_c W(c,t)F(c)$
- In Transformer models:
  - F(c) are feature vectors
  - W(c,t) are attention-style compatibility weights
  - Coend reduces a pooling type operator

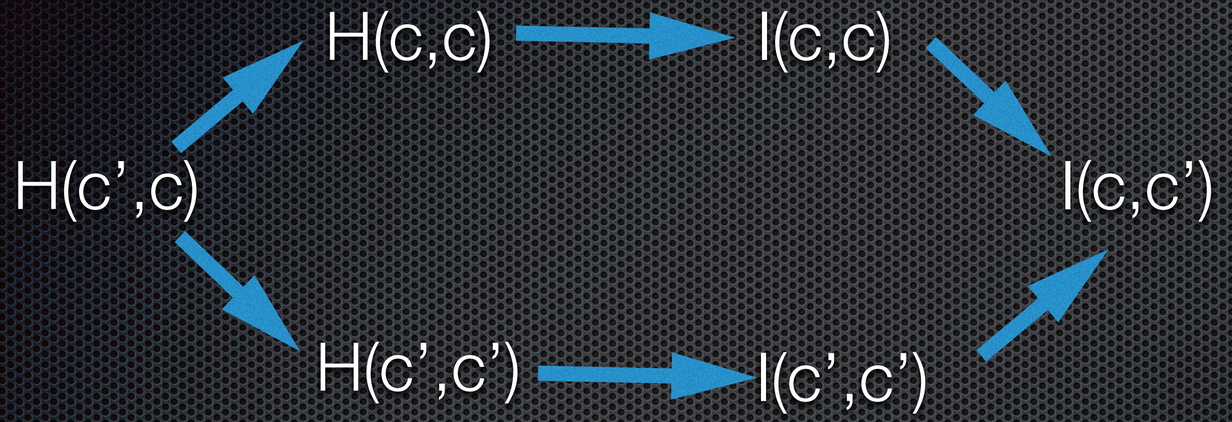
Coends

- Coends are a generalization of integration using bifunctors like Hom(c,d)
  - $\int^{c} H(c, c)$ denotes a coend of a bi-functor H
  - If H: C^{op} x C -> D, then the coend of H is an object of D
- It is defined based on the concept of dinatural transformations

Diagonal Naturality

H, I: C^{op} x C -> D

f:c -> c’



Coend of a Functor

- A coend of a functor H: C^{op} x C -> D is a pair (d, ξ)
  - d is an object of category D
  - ξ is a dinatural transformation that is universal among all such transformations

Topological Coend Transformer

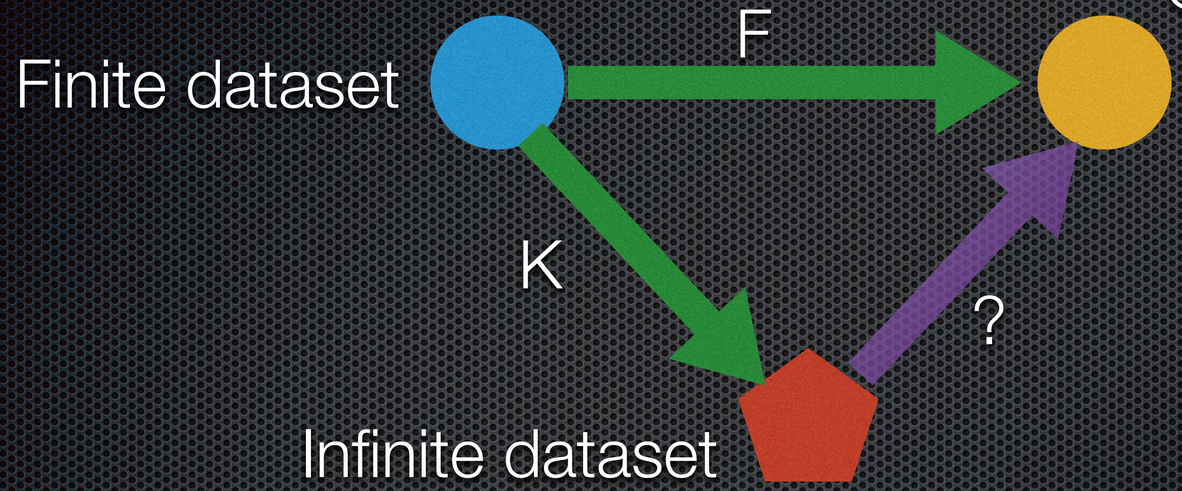
- Weights depend on geometric neighborhood in a learned manifold

```python
class TopoCoendHead(nn.Module):
    """
    Topological coend-style aggregation:
      - Learn a low-dim topological coordinate z_t = topo_proj(h_t)
      - Build a kNN neighborhood graph in z-space (UMAP-ish)
      - Compute fuzzy weights w_{t,s} from distances in z-space
      - Aggregate values over neighbors: out_t = Σ_s w_{t,s} V(h_s)

    Strict-causal by default: neighbors are restricted to s <= t.
    """

    def __init__(self, d: int, topo_dim: int = 16, k: int = 16, causal: bool = True):
        super().__init__()
        self.d = int(d)
        self.topo_dim = int(topo_dim)
        self.k = int(k)
        self.causal = bool(causal)

        self.topo_proj = nn.Linear(d, topo_dim, bias=False)  # learned embedding manifold coords
        self.val_proj = nn.Linear(d, d, bias=False)          # value morphism
        self.out = nn.Linear(d, d, bias=False)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        x: (B,T,d)
        returns: (B,T,d)
        """
        B, T, d = x.shape
        device = x.device

        Z = self.topo_proj(x)     # (B,T,topo_dim)
        V = self.val_proj(x)      # (B,T,d)

        # Pairwise squared distances in topo space: (B,T,T)
        # torch.cdist returns L2 distance; square it.
        dist2 = torch.cdist(Z, Z) ** 2

        # Enforce strict causality: disallow s > t by setting distance = +inf
        if self.causal:
            # allow[t,s] = True iff s <= t
            t_idx = torch.arange(T, device=device)[:, None]
            s_idx = torch.arange(T, device=device)[None, :]
            allow = (s_idx <= t_idx)  # (T,T)
            dist2 = dist2.masked_fill(~allow[None, :, :], float("inf"))
        else:
            # still typically exclude self as trivial neighbor
            pass

        # Exclude self from neighbors (helps prevent degenerate identity copying)
        eye = torch.eye(T, device=device).bool()
        dist2 = dist2.masked_fill(eye[None, :, :], float("inf"))

        # Choose k nearest neighbors for each t (per batch)
        k_eff = min(self.k, T - 1) if T > 1 else 0
        if k_eff <= 0:
            return self.out(V)  # trivial

        # idx: (B,T,k)
        idx = torch.topk(dist2, k_eff, dim=-1, largest=False).indices

        # Gather neighbor distances: (B,T,k)
        dist2_knn = dist2.gather(dim=-1, index=idx)

        # UMAP-ish local scaling: use kth neighbor distance as sigma^2 proxy
        # Avoid divide-by-zero
        sigma2 = dist2_knn[..., -1].clamp_min(1e-8)  # (B,T)
        logits = -dist2_knn / sigma2.unsqueeze(-1)   # (B,T,k)

        w = F.softmax(logits, dim=-1)                # (B,T,k)

        # Gather neighbor values: Vn = V[b, idx[b,t,j], :]
        Vn = V.gather(dim=1, index=idx.unsqueeze(-1).expand(-1, -1, -1, d))  # (B,T,k,d)

        out = (w.unsqueeze(-1) * Vn).sum(dim=2)      # (B,T,d)
        return self.out(out)
```

Extending a Functor

Unlike extension of functions

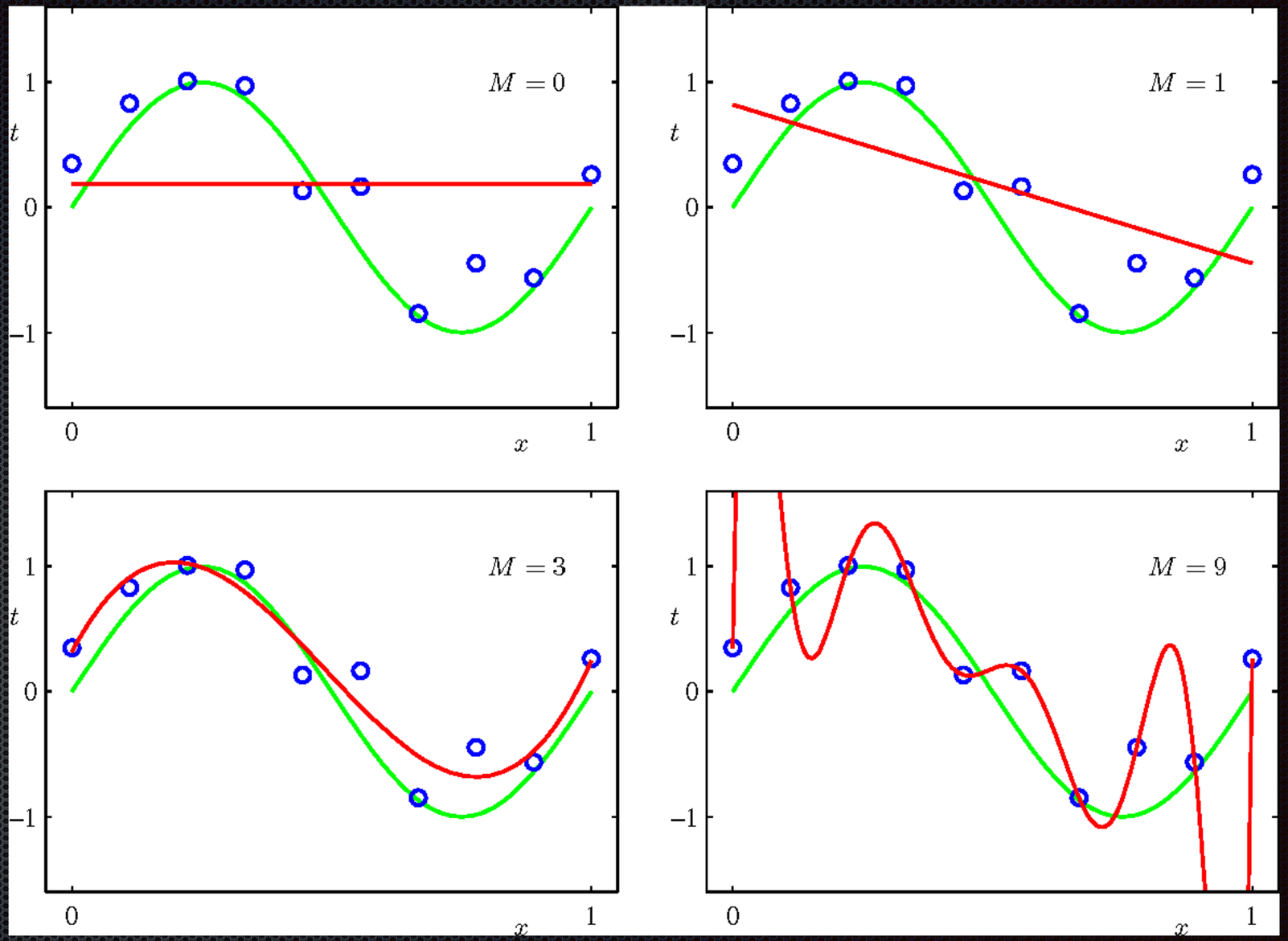
Canonical solutions exist!



Curve Fitting

Left Kan: most general

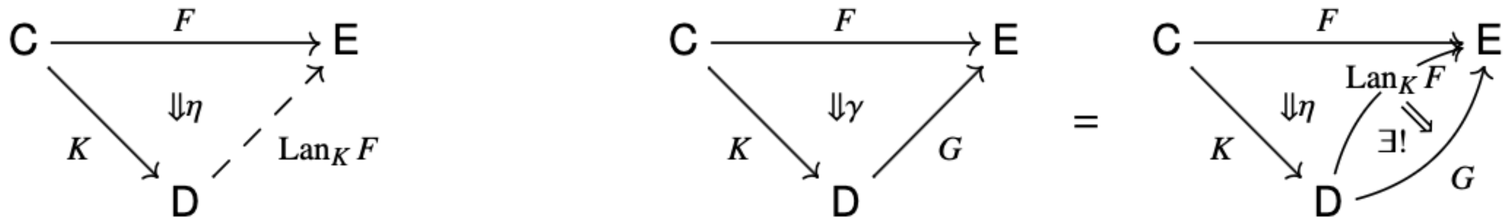
Right Kan: most conservative



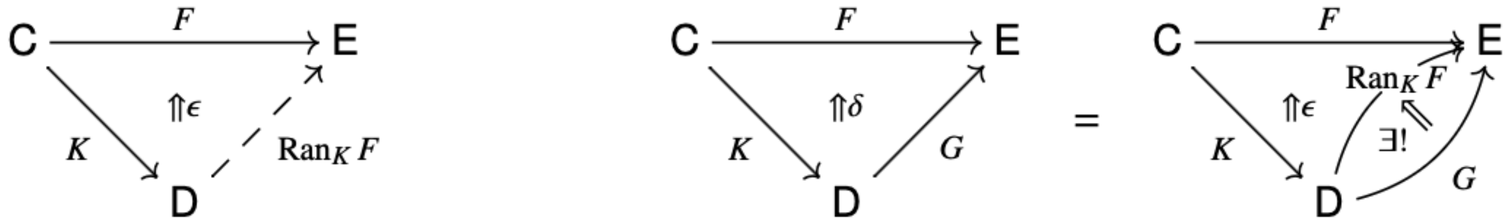
What are Kan Extensions?

- Given a functor F, from some domain category C to co-domain E, “extend” F to some (usually larger) category D
  - Example: C = finite sets, D = all (infinite) sets
  - ML application: C = actual dataset, D = entire space
- We will apply this concept to design a new class of Transformers

DEFINITION 6.1.1. Given functors $F\colon \mathsf{C} \to \mathsf{E}$, $K\colon \mathsf{C} \to \mathsf{D}$, a **left Kan extension** of F along K is a functor $\mathrm{Lan}_K F\colon \mathsf{D} \to \mathsf{E}$ together with a natural transformation $\eta\colon F \Rightarrow \mathrm{Lan}_K F \cdot K$ such that for any other such pair $(G\colon \mathsf{D} \to \mathsf{E}, \gamma\colon F \Rightarrow GK)$, γ factors uniquely through η as illustrated.[1]



Dually, a **right Kan extension** of F along K is a functor $\mathrm{Ran}_K F\colon \mathsf{D} \to \mathsf{E}$ together with a natural transformation $\epsilon\colon \mathrm{Ran}_K F \cdot K \Rightarrow F$ such that for any $(G\colon \mathsf{D} \to \mathsf{E}, \delta\colon GK \Rightarrow F)$, δ factors uniquely through ϵ as illustrated.



Every Concept is a Kan Extension

A unified way to understand all of category theory

When do Kan Extensions exist?

- Given a functor F: C -> E, it can be left-Kan-extended along K: C -> D if
  - C is small, D is locally small, and E is co-complete (so colimits exist)
- Similarly, F can be right-Kan-extended along K: C->D if
  - C is small, D is locally small, and E is complete (so limits exist)

Computing Left Kan Extensions

- We have a functor F: C -> E
  - We know its effect on objects c in C, as well as arrows f: c -> c’
- We have another functor K: C -> D (possibly larger than C)
  - How can we compute the effect of F on this larger category?

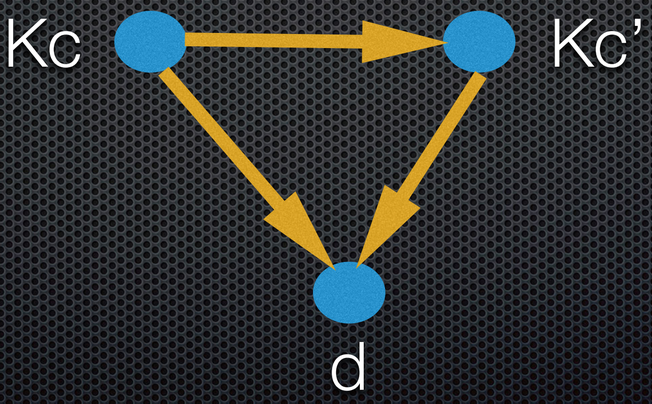
Left Kan Extension computation

- Given any object d in D, to compute the left Kan Extension Lan_K F(d)
  - Consider all possible approximations to d “from the left” using the category C, that is all arrows $Kc \rightarrow d$
  - “Weight” each of these arrows using a “colimit” (gluing the objects)
- So, approximate the objects d in D using the structure of C
- Project back to category C, and then apply the old functor F: C -> E

Comma Categories

- Given a functor K: C -> D, the induced comma category $K \downarrow d$ is defined as the category whose objects are morphisms $Kc \rightarrow d$, and whose arrows are commuting triangles

- For any morphism f: c -> c', the induced triangle $Kc \rightarrow d$, $Kc' \rightarrow d$, and $Kc \rightarrow Kc'$ must commute for it to be an arrow in the $K \downarrow d$ category



Computing Kan Extensions

THEOREM 6.2.1. *Given functors* $F\colon \mathsf{C} \to \mathsf{E}$ *and* $K\colon \mathsf{C} \to \mathsf{D}$, *if for every* $d \in \mathsf{D}$ *the colimit*

$$\operatorname{Lan}_K F(d) := \operatorname{colim}(K\downarrow d \xrightarrow{\Pi^d} \mathsf{C} \xrightarrow{F} \mathsf{E}) \tag{6.2.2}$$

exists, then they define the left Kan extension $\operatorname{Lan}_K F\colon \mathsf{D} \to \mathsf{E}$, *in which case the unit transformation* $\eta\colon F \Rightarrow \operatorname{Lan}_K F \cdot K$ *can be extracted from the colimit cone. Dually, if for every* $d \in \mathsf{D}$ *the limit*

$$\operatorname{Ran}_K F(d) := \lim(d\downarrow K \xrightarrow{\Pi_d} \mathsf{C} \xrightarrow{F} \mathsf{E}) \tag{6.2.3}$$

exists, then they define the right Kan extension $\operatorname{Ran}_K F\colon \mathsf{D} \to \mathsf{E}$, *in which case the counit transformation* $\epsilon\colon \operatorname{Ran}_K F \cdot K \Rightarrow F$ *can be extracted from the limit cone.*

Figure source: Riehl, Chapter 6, Category Theory in Context

Enriched Kan Extensions

- We usually want “enriched” categories in AGI
  - The hom set C(c,d) is itself an object in some category V
  - V is a symmetric monoidal category (e.g., vector spaces)
  - Enriched Kan Extension as a coend: $\mathrm{Lan}_K F = \int^c D(Kc, d) \otimes F(c)$
    - Here D(Kc, d) acts as a “weight”
    - $\otimes$ is a monoidal product like scalar multiplication
    - The coend integrates contributions across all objects c

Left Kan as a Coend!

$$\mathrm{Lan}_K F = \int^{c} D(Kc, d) \otimes F(c)$$

- We can translate the abstract coend in neural networks as:
  - D(Kc, d) is a learned attention score
  - Monoidal product is scalar multiplication
  - Coend is a weighted sum
  - Self-attention itself is simply a weighted sum (like a coend)

Kan Extensions for Transformers

- View attention and geometric mixing (in GT), both as instances of (enriched) left-Kan extensions along a “simplicial incidence” functor
  - Attention is a Kan-type of weighted colimit over tokens
  - Geometric mixing: extend from simplices (tokens, edges, motifs) back to token updates
- Two types of Kan Extension Transformers:
  - Quadratic variant performs global pooling over simplices
  - Incidence-restricted variant aggregates only over incident simplices

Two types of Kan Extension Transformers

Quadratic

Incidence-restricted

[Mahadevan, Categories for AGI]

Quadratic (Global) Kan Pooling

Extending $\mathcal{I}$ to include edges (and optionally higher simplices) yields a *quadratic* Kan update in which each token aggregates from all simplices.

In practice,

$$h'_t = h_t + \sum_{\sigma \in \mathcal{I}} w(t,\sigma)\, V(\sigma), \qquad w(t,\sigma) = \mathrm{softmax}\big(Q_t^\top K_\sigma\big), \tag{8}$$

where queries Q_t and keys K_σ are learned projections.

This introduces a second attention-like kernel over simplices and scales as $O(S^2)$ in sequence length. A simplicial causal mask may optionally enforce

$$\max(\sigma) \le t,$$

preventing future-token leakage.

Quadratic Kan therefore corresponds directly to the enriched coend in Eq. 6 with learned weights.

Incidence-Restricted (Linear-Time) Kan Pooling

If we restrict the coend to *representable incidence weights*, i.e., $\mathrm{Hom}(K\sigma, t)$ is nonzero only when $\sigma \ni t$, then Eq. 6 reduces to

$$(\mathrm{Lan}_K F)(t) \;\approx\; \sum_{\sigma \ni t} \phi\big(F(\sigma)\big), \tag{9}$$

for a learnable message map ϕ.

For edge-only simplices, this becomes:

$$e_t = \psi\big([v_{t-1}, v_t]\big), \tag{10}$$

$$h'_t = h_t + \phi(e_t) \quad \text{(causal incidence)}. \tag{11}$$

Here v_t denotes the value base (hidden states or detached predictive values). The update uses only edges incident to t and scales as $O(S)$.

Thus, incidence-restricted Kan is a left Kan extension along F with representable weights determined by simplex membership.

Information Regimes

We need to be careful in allowing information access

Causal vs. non-causal

Information Regimes

We distinguish three regimes:

- **Strict causal:** updates at time t depend only on positions $\leq t$.
- **Gold noncausal (invalid):** updates at time t depend directly on gold tokens at positions $> t$.
- **Self-conditioned noncausal (predict/detach):** updates at time t may use predicted information about positions $> t$, computed from prefix context and detached from gradients.

The distinction between the latter two regimes is central to this work.

Kan Extensions and Causality

Let K denote a simplicial incidence functor mapping simplices to token positions, and let F denote a value functor assigning feature vectors to simplices. A Kan-style update takes the form

$$(\mathrm{Lan}_K F)(t) = \int^{\sigma} \mathrm{Hom}(K\sigma, t) \otimes F(\sigma).$$

Causality depends entirely on how $F(\sigma)$ is constructed.

Gold Noncausal Regime (Invalid)

Suppose $F(\sigma)$ is constructed from hidden states h computed under teacher forcing. If σ contains a future position $t+1$, then h_{t+1} encodes gold token x_{t+1}.

If noncausal incidence is allowed, then $(\mathrm{Lan}_K F)(t)$ may directly depend on h_{t+1}, and thus on gold x_{t+1}, while predicting x_{t+1}.

Predict and Detach

We can try to predict the next token, but must guard against leakage

Strict Causal Regime

Strict causality may be enforced by masking simplices:

$$\max(\sigma) \leq t.$$

Only prefix simplices contribute to the update at t. This ensures autoregressive validity but restricts geometric expressivity, since no structural information beyond the prefix may be used.

Predict-and-Detach Self-Conditioning

To allow noncausal geometric structure without gold leakage, we replace hidden-state values with detached predictive embeddings.

Let $\ell_t = W_o h_t$ denote logits at position t. Define:

$$\hat{e}_t = \text{detach}(\text{softmax}(\ell_t / T)\, E)\,, \tag{12}$$

where:

- E is the embedding matrix,
- T is a temperature parameter,
- $\text{detach}(\cdot)$ blocks gradient flow.

The predictive embedding $\hat{e}_t$ depends only on prefix tokens $x_{\leq t}$. It represents the model's own belief about x_{t+1}.

Predict and Detach Self-Conditioning

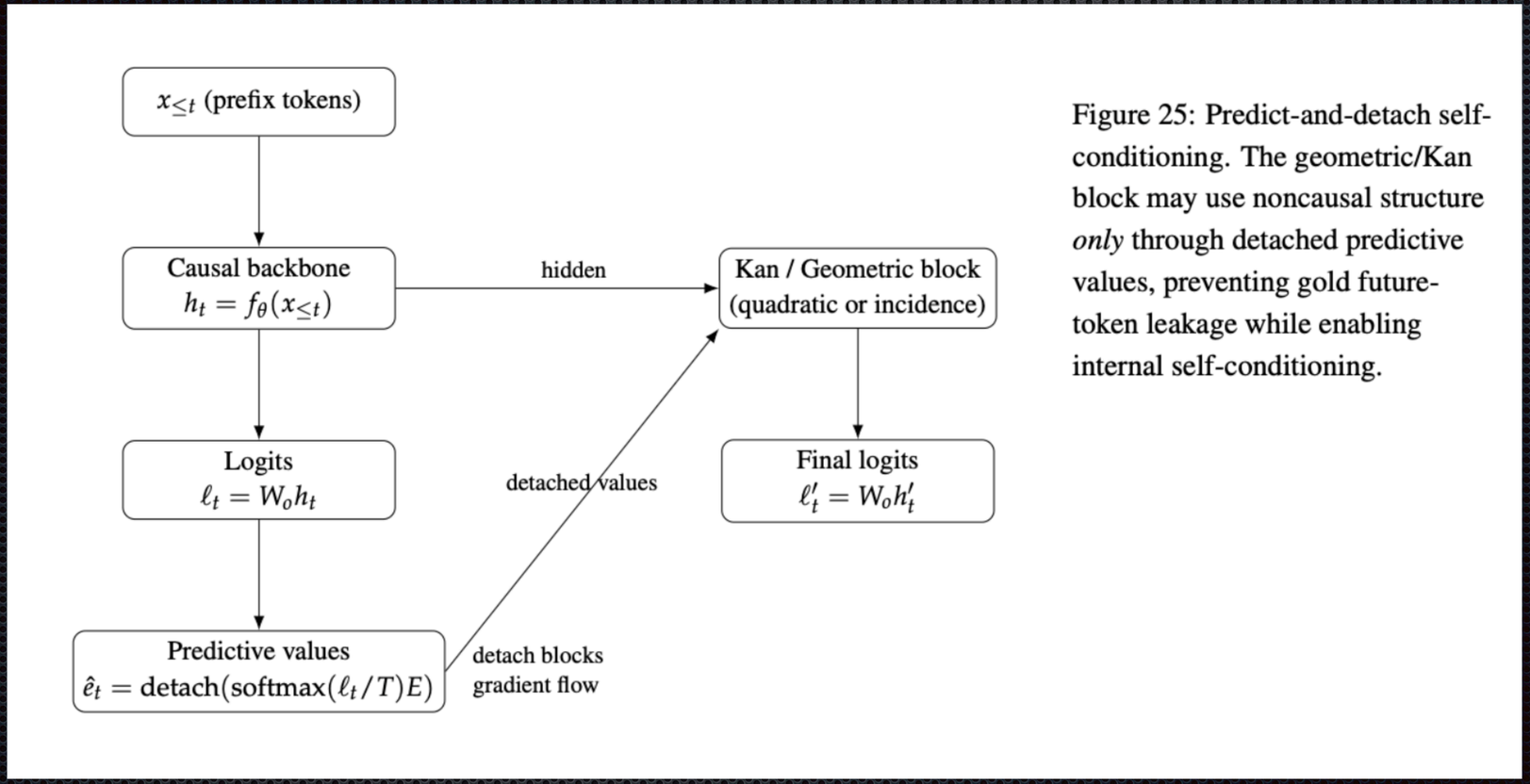


Figure 25: Predict-and-detach self-conditioning. The geometric/Kan block may use noncausal structure *only* through detached predictive values, preventing gold future-token leakage while enabling internal self-conditioning.

[Figure source: Mahadevan, Categories for AI, 2026]

Predict and Detach Algorithm

Algorithm 11: Predict-and-Detach Self-Conditioning

1. Compute causal hidden states $h_t = f_\theta(x_{\le t})$.
2. Compute logits $\ell_t = W_o h_t$.
3. Form predictive embeddings

$$\hat{e}_t = \mathrm{detach}(\mathrm{softmax}(\ell_t / T) E).$$

4. Use $\hat{e}$ as simplex values in noncausal Kan aggregation.
5. Compute final logits from updated hidden states.

Relation to Prompt Repetition

Predict-and-detach is structurally equivalent to prompt repetition. In prompt repetition, generated tokens are fed back into the context for a second pass. Here, predictive embeddings derived from the prefix are incorporated into geometric aggregation within a single forward pass.

Thus, self-conditioned Kan extensions can be interpreted as internalized prompt repetition under categorical control.

Algorithm 12: Predict-and-Detach Value Construction

Input: Prefix tokens $x_{\le t}$, backbone f_θ, output head W_o, embedding matrix E, temperature T

Output: Detached predictive values $\hat{e}_t$

$h_t \leftarrow f_\theta(x_{\le t})$
$\ell_t \leftarrow W_o h_t$
$p_t \leftarrow \mathrm{softmax}(\ell_t / T)$
$\hat{e}_t \leftarrow \mathrm{detach}(p_t E)$ ▷ no gradients through this branch
return $\hat{e}_t$

Quadratic KET

Uses two self-attention loops

Quadratic instead of linear

Algorithm 13: Quadratic Kan Extension Transformer (one block)

Input: Token states $h_{0:S-1}$, simplex set $\mathcal{I}$, value base v (hidden or $\hat{e}$), causal simplex mask optional

Output: Updated token states h'

Construct simplex values $V(\sigma)$ from v (vertices, edges, optional higher simplices)

for $t = 0$ to $S - 1$ **do**

  Compute kernel weights $w(t, \sigma) \propto \exp(Q_t^\top K_\sigma)$

  if causal simplex mask **then**

    Set $w(t, \sigma) \leftarrow 0$ for simplices with $\max(\sigma) > t$

  end if

  $m_t \leftarrow \sum_{\sigma \in \mathcal{I}} w(t, \sigma)\, V(\sigma)$

  $h'_t \leftarrow \mathrm{LN}(h_t + \mathrm{MLP}(m_t))$

end for

return h'

Incidence KET

Faster linear time version

Algorithm 14: Incidence Kan Extension (edge-only, linear-time)

Input: Token states $h_{0:S-1}$, value base v (hidden or $\hat{e}$), causal incidence flag

Output: Updated token states h'

$e_t \leftarrow \psi([v_{t-1}, v_t])$ for $t = 1, \ldots, S-1$ ▷ edge features

for $t = 0$ to $S-1$ **do**

  if causal incidence **then**

    $m_t \leftarrow \phi(e_t)$ using only edge $(t-1, t)$ (for $t \geq 1$)

  else

    $m_t \leftarrow \phi(e_t) + \phi(e_{t+1})$ ▷ uses both incident edges

  end if

  $h'_t \leftarrow \mathrm{LN}(h_t + m_t)$

end for

return h'

Causal TopoCoend

TopoCoend replaces the Kan simplex-indexed pooling with a topological neighborhood embedding in a learned embedding space

Algorithm 15: Strict-Causal TopoCoend (one block)

Input: Token states $h_{0:S-1}$, topo projection π, value map V, neighbor count k, (optional) window W

Output: Updated token states h'

$z_t \leftarrow \pi(h_t)$ for $t = 0, \ldots, S-1$ ▷ topological coordinates

$v_t \leftarrow V(h_t)$ for $t = 0, \ldots, S-1$ ▷ value features

for $t = 0$ to $S-1$ **do**

  Candidate set $\mathcal{C}(t) \leftarrow \{s : \max(0, t-W) \le s \le t\}$ ▷ optional window; else $\{0, \ldots, t\}$

  $\mathcal{N}_k(t) \leftarrow k$-NN of z_t within $\mathcal{C}(t) \setminus \{t\}$

  $\sigma_t^2 \leftarrow \|z_t - z_{s_k}\|^2$ where s_k is the k-th neighbor

  $w_{t,s} \propto \exp(-\|z_t - z_s\|^2 / \sigma_t^2)$ for $s \in \mathcal{N}_k(t)$

  Normalize $w_{t,\cdot}$ over $\mathcal{N}_k(t)$

  $m_t \leftarrow \sum_{s \in \mathcal{N}_k(t)} w_{t,s}\, v_s$

  $h'_t \leftarrow \mathrm{LN}(h_t + \mathrm{MLP}(m_t))$

end for

return h'

TopoCoend with Predict Detach

Noncausal neighbors are now allowed in the neighborhood structure, but only through detached predicted carriers

Algorithm 16: TopoCoend with Predict-and-Detach (one block)

Input: Token states $h_{0:S-1}$, logits $\ell_t = W_o h_t$, embedding matrix E, topo projection π, value map V, neighbors k, (optional) window W

Output: Updated token states h'

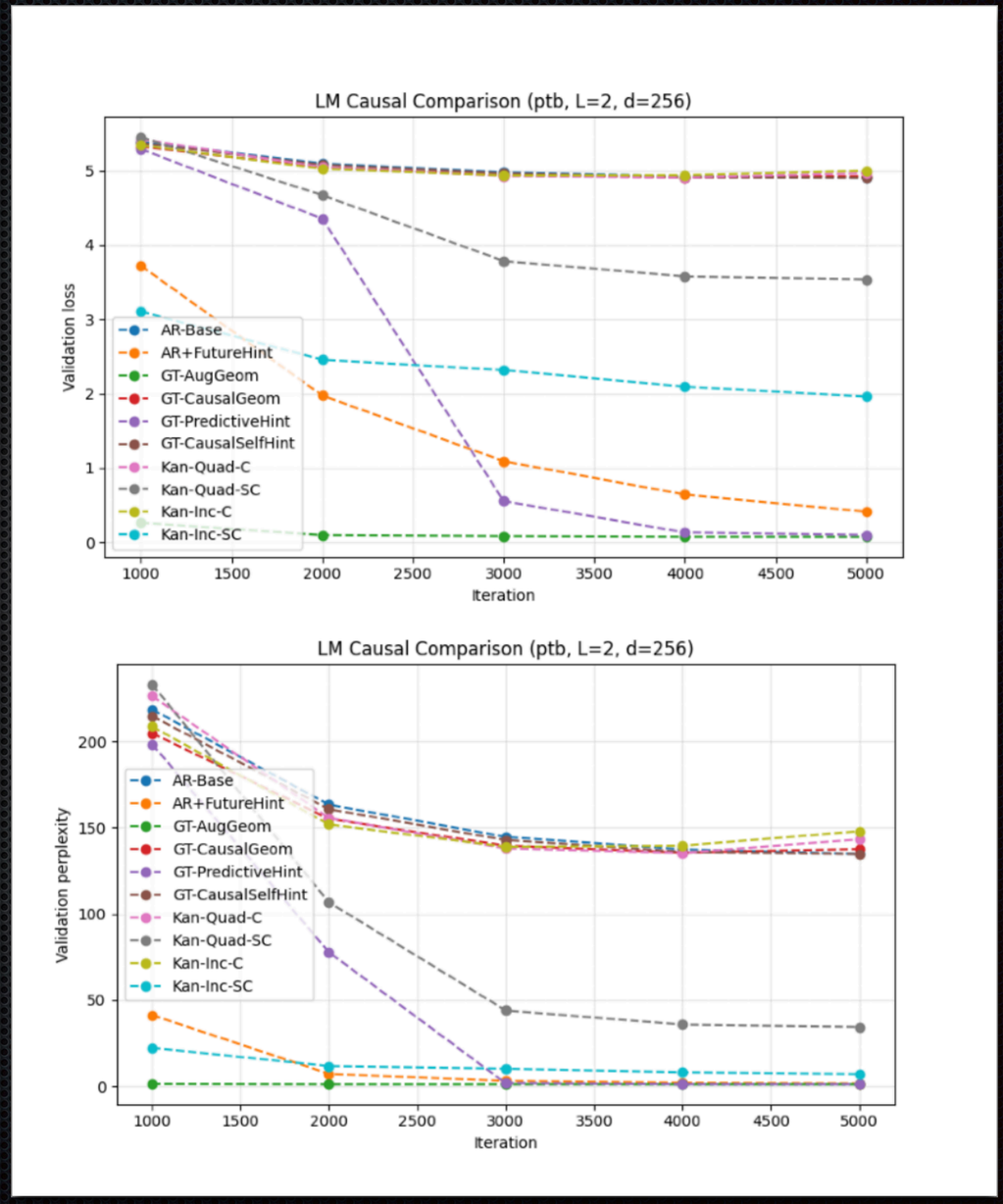
Compute detached predictive carriers $\hat{e}_t$ via Algorithm 12
$z_t^{\text{past}} \leftarrow \pi(h_t)$ and $v_t^{\text{past}} \leftarrow V(h_t)$
$z_t^{\text{fut}} \leftarrow \pi(\hat{e}_t)$ and $v_t^{\text{fut}} \leftarrow V(\hat{e}_t)$
for $t = 0$ to $S-1$ **do**
    Candidate set $\mathcal{C}(t) \leftarrow \{s : \max(0, t-W) \le s \le \min(S-1, t+W)\}$ ▷ optional; else all s
    For $s \in \mathcal{C}(t)$ define $z_s \leftarrow z_s^{\text{past}}$ if $s \le t$, else $z_s \leftarrow z_s^{\text{fut}}$
    $\mathcal{N}_k(t) \leftarrow k$-NN of z_t^{past} among $\{z_s\}_{s \in \mathcal{C}(t) \setminus \{t\}}$
    $\sigma_t^2 \leftarrow \|z_t^{\text{past}} - z_{s_k}\|^2$ (local scale)
    $w_{t,s} \propto \exp(-\|z_t^{\text{past}} - z_s\|^2 / \sigma_t^2)$ for $s \in \mathcal{N}_k(t)$
    Normalize $w_{t,\cdot}$ over $\mathcal{N}_k(t)$
    Define values $v_s \leftarrow v_s^{\text{past}}$ if $s \le t$, else $v_s \leftarrow v_s^{\text{fut}}$
    $m_t \leftarrow \sum_{s \in \mathcal{N}_k(t)} w_{t,s} v_s$
    $h'_t \leftarrow \text{LN}(h_t + \text{MLP}(m_t))$
end for
return h'

Penn Tree Bank

Strict causal models cluster at the top

Noncausal models show perplexity collapse at the bottom

Self-conditioning combine the strengths of both and straddle the middle



PTB Perplexity

Results show stark differences between causal and non-causal models

Regime	Model	Val PPL	Test PPL
Strict-causal comparable			
	AR-Base	135.35	126.31
	GT-CausalSelfHint	135.53	126.33
	GT-CausalGeom	137.36	128.26
	Kan-Quad-C	139.33	130.28
	Kan-Inc-C	146.17	135.55
Self-conditioned (predict/detach)			
	Kan-Inc-SC	5.89	5.37
	Kan-Quad-SC	7.34	6.76
Invalid (gold-leak) — shown for diagnosis			
	GT-AugGeom	1.08	1.05
	GT-PredictiveHint	1.10	1.06
	AR+FutureHint	1.55	1.40

KET vs TopoCoend

TopoCoend is competitive in the causal regime, but not in the predict-detach regime.

Method	PTB Perplexity	Wiki-2 Perplexity	Wiki-103 Perplexity
TopoCoend	131	161	215
Baseline AR Transformer	135	164	230
GT-Causal Geom	137	156	223
KanQuad-C	139	161	216

Running Times

KET models are slower by a factor of 2

Results on nVidia DGX Spark

Model	Test PPL	Iter/s
AR-Base	**124.47**	**34.53**
GT-CausalGeom	127.17	31.87
GT-CausalSelfHint	126.50	14.14
Kan-Quad-C	133.37	15.31
Kan-Inc-C	137.19	26.05

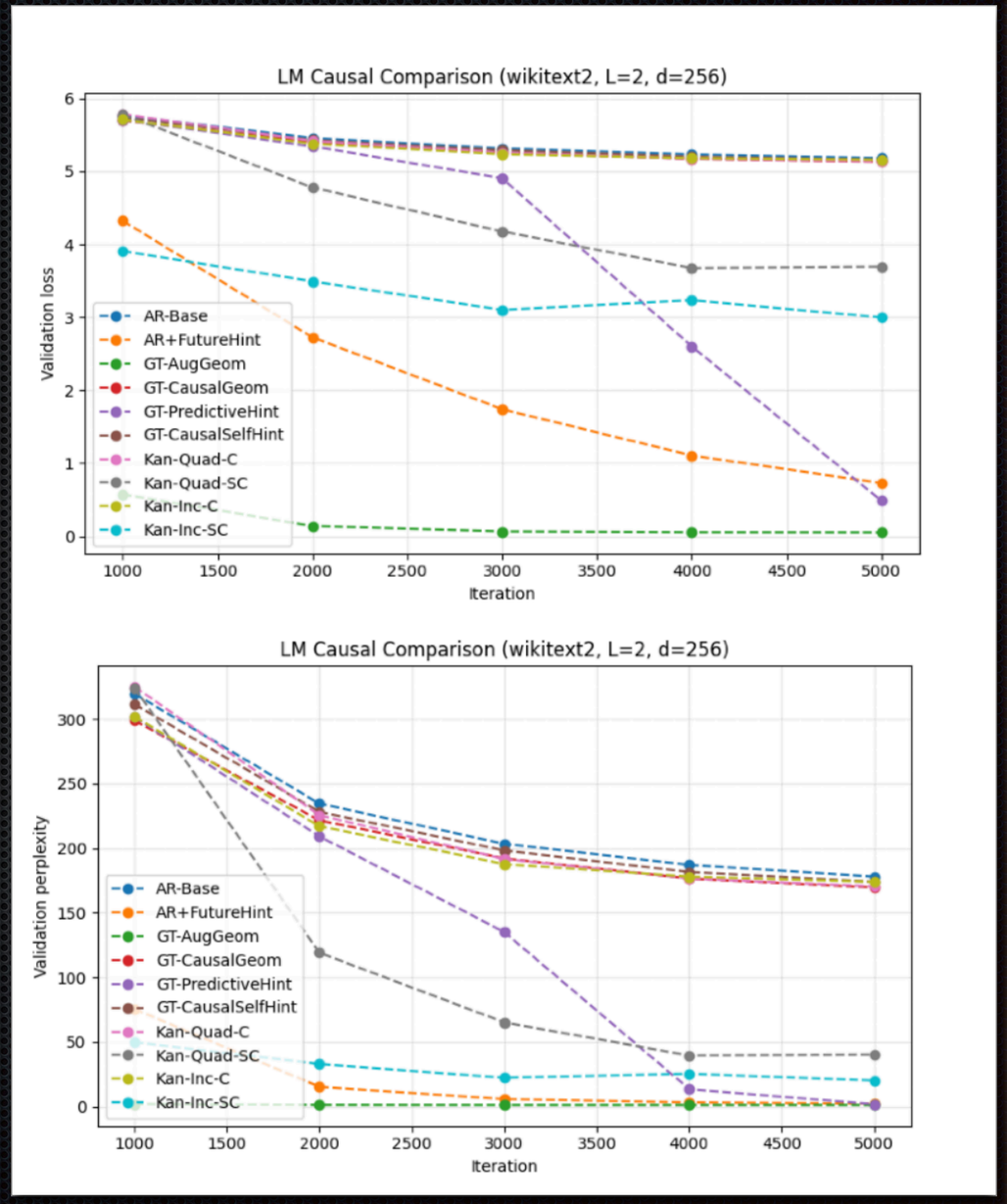
Wiki-2 Dataset

Similar behavior to PTB

Causal methods on top

Self-conditioning in the middle

Anti-causal at the bottom



Wiki-2 Speed

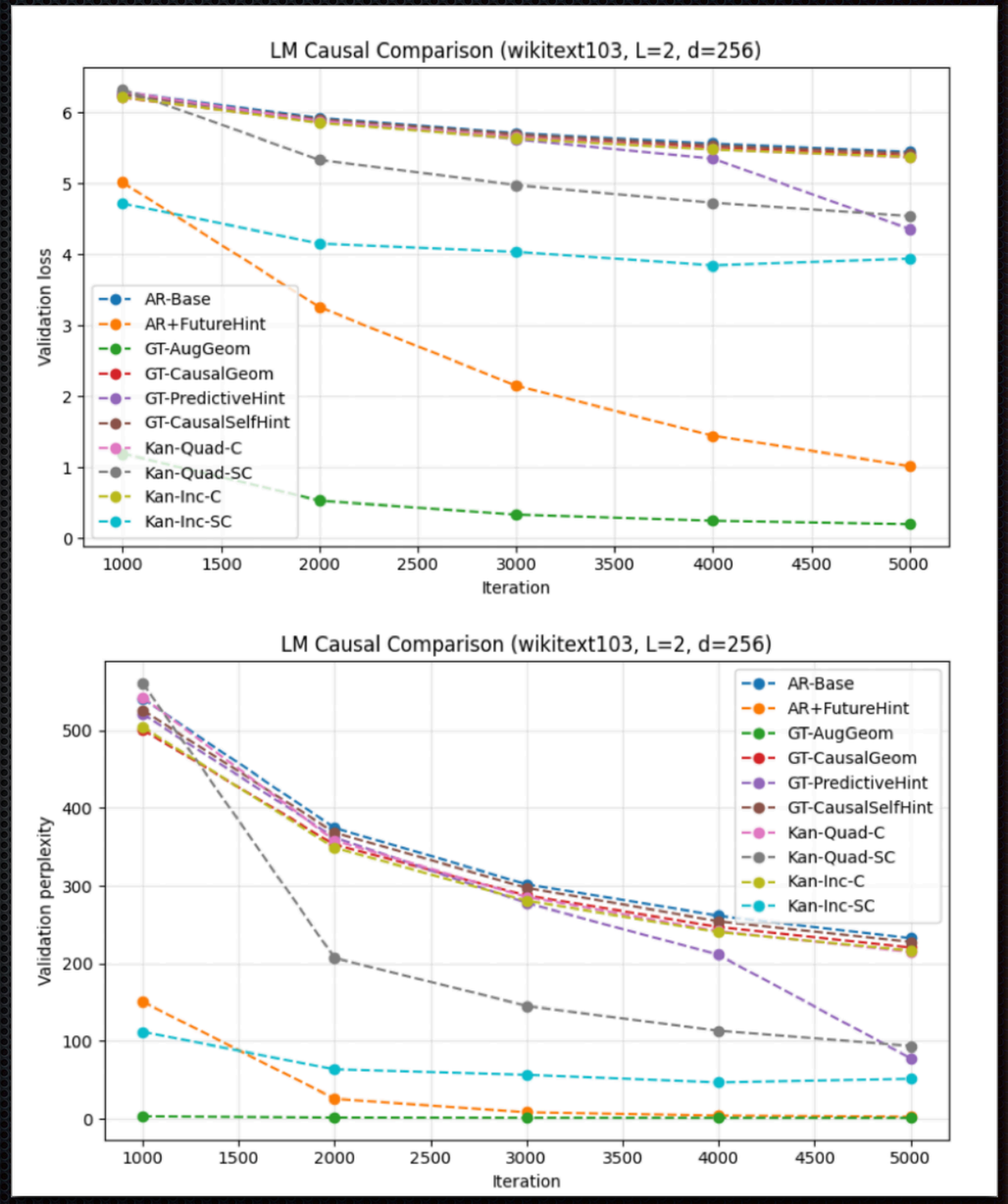
Larger dataset, models are all slower

KET is still more expensive

Model	Test PPL	Iter/s
AR-Base	163.92	**7.53**
GT-CausalGeom	157.74	7.16
Kan-Quad-C	**156.42**	4.01
Kan-Inc-C	161.12	5.19

Wikitext-103

Largest dataset: 100 million tokens



Wiki-103 Speed

Largest dataset

KET is slower by 1.5x

Model	Test PPL	Iter/s
AR-Base	230.12	7.48
GT-CausalGeom	222.64	7.08
Kan-Quad-C	**215.83**	4.02
Kan-Inc-C	224.71	5.12

Summary and Further Reading

- Chapter 6 on Kan Extensions: Riehl, Category Theory in Context
- Chapter on Kan Extension Transformers: Mahadevan, Categories for AGI
- Suggestions for exploration
  - Try sample KET models on GitHub repo
  - How to define Kan Extensions in AGI applications, like RL?
  - Kan Extensions in Causal Inference (later lecture in course)