Retrofitting Large Language Models with Logos-SAE

Hui Xu

August 17, 2025

Abstract

This paper is a companion to our previous work *Structural Axiom of Existence: Logos for Aware LLMs*, which focused on constructing new models under the SAE framework. Here we address the complementary task of *retrofitting existing LLMs*. The **Structural Axiom of Existence (SAE)** originates from the foundation

$$\mathsf{Exist}(X) := \mathsf{Discern}(X) \wedge \mathsf{Free}(X),$$

where $\mathsf{Discern}$ denotes a boundary-making capacity ensuring recognizability and operability, and Free denotes an openness enabling generativity and transformation. When $\mathsf{Discern}$ and Free act in sustained coordination, their trajectory gives rise to $\mathsf{Synchrony}$. Hence the working formulation of SAE for system design is

$$\text{SAE} = \mathsf{Discern} \wedge \mathsf{Free} \wedge \mathsf{Synchrony}.$$

Guided by this principle, we show how existing LLM architectures can be retrofitted to reduce hallucinations, improve energy efficiency, and enhance interpretability, without sacrificing generative capability.

1 Introduction

Large Language Models (LLMs) have become a cornerstone of natural language processing and artificial intelligence. Yet they continue to face several fundamental issues:

- **Hallucination:** generation of false or inconsistent content;
- **High energy cost:** redundant or invalid reasoning paths waste computation;
- **Low interpretability:** opaque mechanisms hinder verification and control;
- **Heavy reliance on external alignment:** post-hoc fixes such as RLHF lack intrinsic guarantees.

We propose SAE as a unifying framework for *retrofitting existing LLMs*. Rooted in its structural foundation ($\mathsf{Exist} = \mathsf{Discern} \wedge \mathsf{Free}$) and extended through the principle of $\mathsf{Synchrony}$, SAE provides a systematic way to embed structural constraints in Transformer models. By integrating $\mathsf{Discern}$ with Free at every stage, and enforcing their $\mathsf{Synchrony}$ across input, representation, and decoding, we obtain **Logos-SAE LLMs** that are more reliable, efficient, and interpretable.

Email: `veloir@163.com` ORCID: 0009-0004-4194-5999

2 SAE Foundation

2.1 Structural Axiom of Existence

The foundational axiom states:

$$\mathsf{Exist}(X) := \mathsf{Discern}(X) \wedge \mathsf{Free}(X).$$

A phenomenon X exists structurally iff it simultaneously exhibits:

- $\mathsf{Discern}$: boundary-making capacity granting recognizability and operability;
- Free: openness granting generativity and transformation.

When these two aspects maintain coherence along generative trajectories, we obtain $\mathsf{Synchrony}$. Thus, the working formulation for LLM design becomes:

$$\text{SAE} = \mathsf{Discern} \wedge \mathsf{Free} \wedge \mathsf{Synchrony}.$$

2.2 Problems of Current LLMs

- **Strong Free, weak Discern:** Attention and Softmax ensure probabilistic generation (Free), while Discern only enters indirectly via external alignment (RLHF).
- **Lack of Synchrony:** many tokens are generated without structural validity, causing hallucinations and wasted computation.

2.3 SAE Improvement Strategy

- Embed Discern into each layer: input, attention, softmax;
- Add Synchrony constraints into training objectives;
- Perform real-time filtering during decoding, acting as an "intrinsic overseer."

3 SAE-LLM Architecture (Technical Details)

3.1 Input Layer: Discern-aware Embedding

Extend the embedding to include a **Discern vector** D:

$$E = [E^{\text{token}}, E^{\text{pos}}, E^{\text{discern}}],$$

where:

- E^{token} = token embedding;
- E^{pos} = positional embedding;
- E^{discern} = discernment embedding (derived from syntax checks, consistency verification, knowledge retrieval, or energy preference).

```
def embed(token, position, context):
    token_vec = token_embedding[token]
    pos_vec = pos_embedding[position]
    discern_vec = compute_discern(token, context)
    return concat([token_vec, pos_vec, discern_vec])
```

3.2 Representation Layer: SAE-Attention

Standard attention:

$$\alpha_{ij} = \frac{\exp(Q_i K_j^\top / \sqrt{d})}{\sum_k \exp(Q_i K_k^\top / \sqrt{d})}.$$

SAE modification:

$$\alpha_{ij} = \frac{\exp(Q_i K_j^\top / \sqrt{d}) \cdot D_j}{\sum_k \exp(Q_i K_k^\top / \sqrt{d}) \cdot D_k}.$$

```
def sae_attention(Q, K, V, D):
    scores = (Q @ K.T) / sqrt(d)
    scores = scores + log(D + eps)   # Discern correction
    attn = softmax(scores, dim=-1)
    return attn @ V
```

3.3 Output Layer: SAE-Softmax

Standard softmax:

$$p_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}.$$

SAE modification:

$$p_i = \frac{\exp(z_i) \cdot D_i}{\sum_j \exp(z_j) \cdot D_j}.$$

```
def sae_softmax(logits, D, dim=-1, eps=1e-12):
    adj_logits = logits + torch.log(D + eps)
    return F.softmax(adj_logits, dim=dim)
```

3.4 Training Objective: Synchrony Optimization

Standard cross-entropy loss:

$$\mathcal{L}_{\text{task}} = -\sum_t \log p(y_t).$$

Synchrony score:

$$\mathcal{E} = \frac{\sum_i p_i D_i}{\sum_i p_i}.$$

Training loss:

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \alpha(1 - \mathcal{E}) + \beta \mathcal{L}_{\text{energy}}.$$

3.5 Decoding: SAE-Decoding

Traditional decoding: sample from p_i. SAE modification: filter candidates with low synchrony.

```
def sae_decoding_step(logits, D):
    probs = sae_softmax(logits, D)
    sync_probs = probs * D
    next_token = torch.argmax(sync_probs)
    return next_token, sync_probs
```

4 System View: Dual-Stream Mechanism

SAE-LLM is a **dual-stream system**:

- **Free stream:** explores generative possibilities (probability distribution);
- **Discern stream:** scores each candidate token by structural validity;
- **Synchrony module:** integrates the two streams for the final decision.

Formal expression:

$$\mathrm{Output}(t) = \arg\max_{v\in\mathcal{V}} \; p(v) \cdot D(v).$$

```
def sae_step(Q, K, V, logits, context):
    attn_out = sae_attention(Q, K, V, D=None)  # Free stream
    D = compute_discern_logits(logits, context) # Discern stream
    next_token, probs = sae_decoding_step(logits, D)
    return next_token, probs
```

5 Expected Benefits

- Lower hallucination rate: invalid tokens ($D = 0$) are filtered out;
- Energy efficiency: low-discernment paths are pruned early;
- Better interpretability: Discern sources are trackable, making decisions explainable.

6 Conclusion

SAE-LLM introduces a new paradigm:

- Embeds Discern within every LLM layer;
- Achieves generation-validation synchrony;
- Provides an executable framework: formulas, pseudocode, and implementation pathway.

7 SAE-Attention (Algorithm)

7.1 Formula

$$\alpha_{ij} = \frac{\exp(Q_i K_j^\top/\sqrt{d}) \cdot D_j}{\sum_k \exp(Q_i K_k^\top/\sqrt{d}) \cdot D_k}, \qquad \text{SAE-Attn}(Q,K,V)_i = \sum_j \alpha_{ij} V_j.$$

7.2 Algorithm

Algorithm 1 SAE-Attention

Require: Queries Q, Keys K, Values V, Discern weights D

1: Compute similarity scores: $S = QK^\top/\sqrt{d}$
2: Adjust scores with Discern: $S \leftarrow S + \log(D + \varepsilon)$
3: Apply softmax: $\alpha = \text{softmax}(S)$
4: Weighted aggregation: $O = \alpha V$
5: **return** O, α

8 SAE-Softmax (Algorithm)

8.1 Formula

$$p_i = \frac{\exp(z_i) \cdot D_i}{\sum_j \exp(z_j) \cdot D_j}.$$

8.2 Algorithm

Algorithm 2 SAE-Softmax

Require: Logits z, Discern weights D

1: Adjusted logits: $z' \leftarrow z + \log(D + \varepsilon)$
2: Compute probabilities: $p = \text{softmax}(z')$
3: **return** p

9 Synchronization-Aware Training (Algorithm)

9.1 Formula

Task loss:

$$\mathcal{L}_{\text{task}} = -\sum_t \log p(y_t).$$

Synchrony:

$$\mathcal{E} = \frac{\sum_i p_i D_i}{\sum_i p_i}.$$

Overall loss:

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \alpha(1 - \mathcal{E}) + \beta \mathcal{L}_{\text{energy}}.$$

9.2 Algorithm

Algorithm 3 SAE-Synchrony Training

Require: Model parameters θ, Discern estimator ϕ

for each training step **do**
  Forward pass: compute logits z and probs p
  Compute Discern weights $D = D_\phi(x)$
  Compute synchrony $\mathcal{E} = \sum_i p_i D_i / \sum_i p_i$
  Compute loss $\mathcal{L} = \mathcal{L}_{task} + \alpha(1 - \mathcal{E}) + \beta\mathcal{L}_{energy}$
  Backpropagation: update θ, ϕ
end for

10 SAE-Decoding (Algorithm)

10.1 Formula

$$\text{Output}(t) = \arg\max_{v \in \mathcal{V}} p(v) \cdot D(v).$$

10.2 Algorithm

Algorithm 4 SAE-Decoding

Require: Logits z, Discern weights D

Compute SAE-Softmax: $p = \text{sae-softmax}(z, D)$
Synchronize: $p' \leftarrow p \cdot D$
Select token: $t^* = \arg\max p'$
return t^*, p'

11 Full SAE-LLM Pipeline

The complete execution flow of SAE-LLM from input to output:

12 Implementation Notes (Minimal PyTorch Demo)

This section summarizes practical choices and provides a minimal, end-to-end PyTorch sketch to reproduce SAE-attention, SAE-softmax, the synchrony-aware loss, and decoding. The code is intentionally lightweight and omits data/loader details.

12.1 Design Choices and Tips

- **Discern estimator D_ϕ:** start with a simple MLP head on top of contextual hidden states; optionally fuse external signals (syntax checks, NLI scores, retrieval hits). Use `sigmoid` to bound $D \in [0, 1]$.

Algorithm 5 SAE-LLM Pipeline

Require: Input sequence $x_{1:T}$
Embedding: tokenize, embed, and compute initial Discern weights
Transformer layers:
for each layer $l = 1 \ldots L$ **do**
  Compute SAE-Attention with Discern
  Apply FFN + residuals
  Update Discern estimator $D^{(l)}$
end for
Output: SAE-Softmax with $D^{(L)}$
Synchrony metric: $\mathcal{E} = \sum_i p_i D_i / \sum_i p_i$
Training: minimize $\mathcal{L} = \mathcal{L}_{task} + \alpha(1 - \mathcal{E}) + \beta\mathcal{L}_{energy}$
Decoding: select tokens ensuring $\mathsf{Free} \wedge \mathsf{Discern}$
return Output sequence $\hat{y}_{1:T}$

- **Gradient flow:** when D comes from hard constraints (e.g., grammar mask), detach gradients (no backprop through D). For learned D_ϕ, allow gradients but consider a smaller LR and EMA to stabilize.
- **Stability:** always add `eps` and use `log(D + eps)` in logits or attention scores. Clip or floor D to avoid $\log 0$.
- **Energy proxy:** define `cost` per token step (e.g., retrieval depth, KV-cache growth, tool-call flag) and penalize via $\mathcal{L}_{\text{energy}}$.
- **Curriculum:** ramp α (sync weight) from 0 to target over first epochs; similarly ramp macro-scale constraints if any.
- **Evaluation:** report hallucination rate (e.g., TruthfulQA), task accuracy, *Joules/token* (or FLOPs/token), and the synchrony score $\mathcal{E} = \sum p_i D_i / \sum p_i$.

12.2 Minimal Modules

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SAEAttention(nn.Module):
    def __init__(self, d_model, n_heads):
        super().__init__()
        assert d_model % n_heads == 0
        self.d_model = d_model
        self.n_heads = n_heads
        self.d_head = d_model // n_heads
        self.Wq = nn.Linear(d_model, d_model)
        self.Wk = nn.Linear(d_model, d_model)
        self.Wv = nn.Linear(d_model, d_model)
        self.Wo = nn.Linear(d_model, d_model)
```

```
    def forward(self, x, discern=None, attn_mask=None):
        """
        x: (B, T, d_model)
        discern: (B, T) in [0,1] for keys (and optionally values)
        """
        B, T, D = x.size()
        q = self.Wq(x).view(B, T, self.n_heads, self.d_head).transpose(1, 2)  # (B,H,T,dh)
        k = self.Wk(x).view(B, T, self.n_heads, self.d_head).transpose(1, 2)  # (B,H,T,dh)
        v = self.Wv(x).view(B, T, self.n_heads, self.d_head).transpose(1, 2)  # (B,H,T,dh)
        scores = torch.matmul(q, k.transpose(-2, -1)) / (self.d_head ** 0.5)   # (B,H,T,T)
        if attn_mask is not None:
            scores = scores.masked_fill(attn_mask, float('-inf'))
        if discern is not None:
            # expand discern (B,T)->(B,1,1,T) and add log(D+eps)
            Dk = torch.clamp(discern, min=1e-12).unsqueeze(1).unsqueeze(1)
            scores = scores + torch.log(Dk)
        attn = torch.softmax(scores, dim=-1)
        out = torch.matmul(attn, v)                                           # (B,H,T,dh)
        out = out.transpose(1, 2).contiguous().view(B, T, D)                  # (B,T,D)
        return self.Wo(out), attn

class DiscernHead(nn.Module):
    def __init__(self, d_model, hidden=256):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(d_model, hidden),
            nn.ReLU(),
            nn.Linear(hidden, 1)
        )
    def forward(self, h):
        # h: (B,T,D) -> D in [0,1]
        return torch.sigmoid(self.net(h)).squeeze(-1)

def sae_softmax(logits, D, dim=-1, eps=1e-12):
    # logits: (B,V), D: (B,V) in [0,1]
    D_safe = torch.clamp(D, min=eps)
    return F.softmax(logits + torch.log(D_safe), dim=dim)

class SAEBlock(nn.Module):
    def __init__(self, d_model, n_heads, d_ff):
        super().__init__()
        self.attn = SAEAttention(d_model, n_heads)
        self.ln1 = nn.LayerNorm(d_model)
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_ff), nn.GELU(), nn.Linear(d_ff, d_model)
        )
        self.ln2 = nn.LayerNorm(d_model)
        self.discern_head = DiscernHead(d_model)
```

```
    def forward(self, x, attn_mask=None, discern_keys=None):
        # Discern for keys can be computed from previous hidden states
        attn_out, _ = self.attn(x, discern=discern_keys, attn_mask=attn_mask)
        x = self.ln1(x + attn_out)
        ff_out = self.ff(x)
        x = self.ln2(x + ff_out)
        # produce updated discern scores from current states
        D = self.discern_head(x)  # (B,T)
        return x, D

class SAETransformerLM(nn.Module):
    def __init__(self, vocab_size, d_model=512, n_heads=8, d_ff=2048, n_layers=6):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, d_model)
        self.pos = nn.Embedding(4096, d_model)
        self.blocks = nn.ModuleList([SAEBlock(d_model, n_heads, d_ff) for _ in range(n_layer
        self.ln_f = nn.LayerNorm(d_model)
        self.lm_head = nn.Linear(d_model, vocab_size, bias=False)

    def forward(self, idx, attn_mask=None, external_D_vocab=None):
        """
        idx: (B,T) token ids
        external_D_vocab: optional (B,T,V) discern over vocab at each step
        """
        B, T = idx.size()
        pos = torch.arange(T, device=idx.device).unsqueeze(0).expand(B, T)
        h = self.embed(idx) + self.pos(pos)              # (B,T,D)
        D_keys = None
        for blk in self.blocks:
            h, D_keys = blk(h, attn_mask=attn_mask, discern_keys=D_keys)
        h = self.ln_f(h)
        logits = self.lm_head(h)                         # (B,T,V)
        # compute vocab-level discern; default from last hidden state
        if external_D_vocab is not None:
            D_vocab = external_D_vocab
        else:
            # simple projection: reuse lm_head weights for a score, then sigmoid
            D_vocab = torch.sigmoid(logits.detach())     # detach if you want fixed D
        return logits, D_keys, D_vocab
```

12.3 Synchrony Loss and Training Loop

```
def synchrony_score(probs, D_vocab, eps=1e-12):
    # both shape (B,T,V)
    num = (probs * D_vocab).sum(dim=-1)         # (B,T)
    den = probs.sum(dim=-1).clamp_min(eps)      # (B,T)
    return (num / den).mean()                   # scalar
```

```
def sae_loss(logits, targets, D_vocab, alpha=0.1, beta=0.0, cost=None):
    # logits: (B,T,V), targets: (B,T), D_vocab: (B,T,V)
    probs = F.softmax(logits, dim=-1)
    ce = F.cross_entropy(logits.view(-1, logits.size(-1)),
                         targets.view(-1), reduction='mean')
    E = synchrony_score(probs, D_vocab)
    L_sync = 1.0 - E
    if cost is not None:
        # cost: (B,T,V) or (B,T) proxy
        if cost.dim() == 3:
            L_energy = (probs * cost).sum(dim=-1).mean()
        else:
            L_energy = cost.mean()
    else:
        L_energy = torch.tensor(0.0, device=logits.device)
    return ce + alpha * L_sync + beta * L_energy, dict(ce=ce, E=E, L_sync=L_sync,
                                                       L_energy=L_energy)

# ---- training step (sketch) ----
def train_step(model, batch, optimizer, alpha=0.1, beta=0.0):
    model.train()
    idx, targets = batch['input_ids'], batch['labels']
    logits, D_keys, D_vocab = model(idx)  # external_D_vocab=None
    loss, logs = sae_loss(logits, targets, D_vocab, alpha=alpha, beta=beta)
    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    optimizer.step()
    return {k: v.item() for k, v in logs.items()} | {'loss': loss.item()}
```

12.4 SAE-Decoding (Greedy)

```
@torch.no_grad()
def sae_greedy_decode(model, start_ids, max_len=64, temperature=1.0, eps=1e-12):
    model.eval()
    idx = start_ids
    for _ in range(max_len):
        logits, D_keys, D_vocab = model(idx)
        logits_step = logits[:, -1, :] / max(temperature, 1e-6)  # (B,V)
        D_step = D_vocab[:, -1, :].clamp_min(eps)                # (B,V)
        probs = F.softmax(logits_step + torch.log(D_step), dim=-1)
        # synchrony-weighted greedy choice
        sync_probs = probs * D_step
        next_token = torch.argmax(sync_probs, dim=-1, keepdim=True)  # (B,1)
        idx = torch.cat([idx, next_token], dim=1)
    return idx
```

12.5 Hooking External Discern Signals

- **Hard masks** (grammar, JSON schema): set $D = 0$ for disallowed tokens.
- **Retrieval consistency:** boost D for tokens supported by retrieved passages.
- **NLI verifier:** compute entailment scores and map to $D \in [0, 1]$.
- **Energy-aware prior:** $D = \exp(-\lambda \cdot \mathrm{cost})$ for expensive operations.

12.6 Recommended Hyperparameters (Starting Points)

- α (synchrony weight): 0.05–0.2; warm up from 0 over 5–10% of steps.
- β (energy weight): 0.0 initially; introduce 0.01–0.1 after convergence.
- Discern head LR: 2–3× main LR; consider EMA(0.95) for D stabilization.
- Floor D: use `clamp_min(1e-6)`, and consider temperature tuning at decode.

12.7 Repro Checklist

- Log $\mathcal{E}$ per step and per layer; correlate with hallucination errors.
- Ablate: vanilla vs. SAE-Softmax only vs. SAE-Attention only vs. full SAE.
- Report Joules/token (or FLOPs/token) and latency alongside task metrics.
- Verify that D improves with training (calibration plots, AUC vs. weak labels).

13 Expected Changes under SAE-Based Redesign

From the perspective of the *Structural Axiom of Existence* (SAE = Discern $\wedge$ Free), redesigning LLMs with SAE-Attention, SAE-Softmax, and synchronization-aware training is expected to yield the following changes:

13.1 Generation Quality

- **Reduced hallucinations:** tokens or trajectories with $D = 0$ are pruned in both attention and softmax, suppressing structurally invalid generations.
- **Higher structural consistency:** outputs such as JSON, code, or proofs will adhere more strongly to structural rules, with synchrony $\mathcal{E}$ pushing probability mass toward valid candidates.
- **Creativity preserved:** the *Free* component (distributional diversity) remains intact, so generative flexibility is not lost.

13.2 Training and Convergence

- **Faster and more stable convergence:** synchronization regularizers act as intrinsic rewards, pruning meaningless updates during training.
- **Lower risk of overfitting:** SAE requires alignment across structural factors, not just memorization of token frequencies.
- **Multi-scale consistency:** discern signals can propagate across token, span, and document levels, ensuring structural synchrony across scales.

13.3 Inference Efficiency and Energy

- **Energy reduction:** paths with low discernment are excluded early, reducing FLOPs/token and KV-cache growth.
- **Lower latency:** fewer candidate expansions enable faster decoding and convergence in beam search or sampling.

13.4 Interpretability and Controllability

- **More interpretable attention maps:** attention reflects both similarity and structural validity, clarifying which tokens are structurally acceptable.
- **Enhanced controllability:** external rules (syntax, domain knowledge, safety constraints) can be encoded in D, directly influencing behavior without relying solely on RLHF.

13.5 Risks and Trade-offs

- **Potential reduction in creativity:** overly strict discernment may make outputs more conservative, limiting boundary-pushing innovation.

- **Critical dependence on discern quality:** if the estimator D_ϕ is weak, it may over-filter or under-filter, harming usability.
- **Increased training complexity:** additional heads, regularizers, and cost proxies introduce more hyperparameters and tuning overhead.

13.6 Summary

An SAE-based LLM is expected to become *more reliable, consistent, energy-efficient, and controllable*, while retaining creativity. The main trade-off lies in balancing Discern strictness with generative freedom.

13.7 Before vs. After SAE Redesign

Aspect	**Standard LLMs (Before SAE)**	**SAE-Based LLMs (After Redesign)**
Generation Quality	High freedom, but frequent hallucinations; weak structural guarantees	Reduced hallucinations via D-weights; stronger structural consistency with $\mathsf{Discern} \wedge \mathsf{Free}$
Training and Convergence	Cross-entropy only; slow convergence; prone to memorization	Synchronization regularizers; faster convergence; lower overfitting; multi-scale consistency
Inference Efficiency	High energy use; redundant token expansions; large KV-cache growth	Energy-efficient pruning; only discern-valid paths proceed; reduced FLOPs/token
Interpretability	Attention reflects only similarity; limited explainability	Attention integrates discernment; structurally meaningful maps
Controllability	RLHF as external patch; fragile, expensive to align	D integrates domain/safety rules directly; controllable at inference time
Risks / Trade-offs	Creativity unbounded but unreliable	Reliability improved, but creativity may shrink if D overly strict; requires high-quality discern estimator

Table 1: Comparison of standard Transformer-based LLMs and SAE-based redesigned LLMs.

14 Systematic SAE Redesign of LLM Components

In addition to modifying attention, softmax, and training objectives, the Structural Axiom of Existence (SAE: $\mathsf{Exist}(X) = \mathsf{Discern}(X) \wedge \mathsf{Free}(X)$) suggests a systematic redesign of almost every core mechanism in LLMs. We outline below seven additional points of intervention.

14.1 Discern-Aware Input Embeddings

Standard embeddings $E(t)$ capture only semantic statistics. We extend them with discernment weights:

$$E'(t) = [E(t), D(t)],$$

where $D(t) \in [0, 1]$ measures structural validity (syntax, logic, or knowledge consistency). This ensures Discern is injected at the very first layer.

14.2 Structural Layer Normalization

LayerNorm currently stabilizes variance but ignores structure. We propose:

$$h' = \frac{h - \mu}{\sigma} \cdot f(D),$$

where $f(D)$ rescales activations based on Discern weights. Low-discernment tokens thus contribute less to gradient flow.

14.3 Discern-Gated Residual Connections

Residuals propagate all information: $y = x + F(x)$. SAE modifies this as:

$$y = x + D \odot F(x),$$

where D gates contributions of each token or channel. Invalid trajectories no longer accumulate noise.

14.4 KV-Cache Pruning

In long-context inference, the KV-cache grows linearly. We restrict storage to discerned keys:

$$\mathrm{KV}' = \{(k, v) \mid D(k) \geq \tau\}.$$

Only structurally valid tokens remain in memory, improving efficiency.

14.5 SAE-Decoding Strategies

Decoding methods such as Top-k or nucleus sampling consider only probabilities. We redefine sampling distribution as:

$$p_i' = \frac{p_i D_i}{\sum_j p_j D_j}.$$

This guarantees Discern is enforced during generation itself, not only as a post-filter.

14.6 Training Paradigm Beyond Cross-Entropy

We extend the objective:

$$\mathcal{L} = \mathcal{L}_{\text{task}} + \alpha\,\mathcal{L}_{\text{sync}} + \beta\,\mathcal{L}_{\text{energy}} + \gamma\,\mathcal{L}_{\text{discern-calib}}.$$

Discern calibration stabilizes D estimators, while synchronization and energy regularizers optimize trajectories as "existence paths" rather than mere token prediction.

14.7 Cross-Modal Discernment

For multimodal LLMs, Discern must be aligned across modalities:

$$D^{\text{multi}} = f(D^{\text{text}}, D^{\text{vision}}, D^{\text{audio}}).$$

This prevents hallucinatory cross-modal associations and ensures valid semantic alignment across input channels.

Summary

SAE acts not as a small modification but as a global design principle: embedding, normalization, residuals, memory, decoding, training, and multimodal fusion can all be redefined under $\mathsf{Discern} \wedge \mathsf{Free}$, leading toward a "second-generation Transformer" architecture.