```
import os
import math
import copy
import time
import json
from typing import Optional, Tuple, List, Dict

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
import numpy as np
import matplotlib.pyplot as plt
from tqdm.auto import tqdm

# Hugging Face Libraries
from datasets import load_dataset
from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import Whitespace
from transformers import PreTrainedTokenizerFast

#
================================================================================
======
# 1. CONFIGURATION & HYPERPARAMETERS
#
================================================================================
======

CONFIG = {
    "run_name": "SoME_v5.2_The_Thaw",
    "seed": 42,

    # Data Settings
    "data": {
        "dataset_name": "roneneldan/TinyStories",
        "train_subset_size": 10000,  # Increased for stability
        "val_subset_size": 2000,
        "batch_size": 32,
        "num_workers": 4,
    },
```

```
# Model Architecture (SoME Transformer)
"model": {
    "vocab_size": 8192,      # Will be overwritten by actual tokenizer vocab size
    "d_model": 512,
    "num_heads": 8,
    "num_layers": 8,
    "seq_len": 512,
    "dropout": 0.1,
},

# SoME Routing Physics (The "Big 8" Fixed)
"some_layer": {
    "num_experts": 256,
    "d_ffn": 1536,           # 3x expansion
    "top_k": 8,

    # Initialization
    "init_method": "sparse", # Crucial for "Jagged Tools" hypothesis

    # Heuristic Physics
    "alpha": 0.01,           # Attraction (Inverse Hebbian)
    "beta": 0.001,           # Peer-Pull (Clustering)
    "delta": 0.001,          # Decay (Repulsion/Cleanup)

    # Stability / Management
    "theta_percentile": 0.05,
    "warmup_steps": 1000,
    "ema_decay": 0.99,

    # Router Architecture
    "router_type": "mlp",    # Enforce Manifold Warping (Law of Router Capacity)
    "router_mlp_mult": 2.0,  # Hidden dim multiplier for router
},

# Training Schedule
"training": {
    "num_epochs": 4,
    "thaw_at_epoch": 2,      # 0-indexed: Freezes 0, 1. Thaws at start of 2.
    "learning_rate": 6e-4,
    "weight_decay": 0.1,
    "training_temp": 1.0,    # Softmax temperature during training
    "eval_temp": 1.0,        # Standardized eval temperature (Fix #5)
    "use_amp": True,         # Automatic Mixed Precision
}
```

```
}

# Set Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
torch.manual_seed(CONFIG['seed'])
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(CONFIG['seed'])
    # TF32 Optimization for Ampere+ GPUs
    torch.set_float32_matmul_precision('high')
    torch.backends.cudnn.benchmark = True

print(f"Running on device: {device}")

#
================================================================================
======
# 2. DATA PREPARATION UTILITIES
#
================================================================================
======

class LanguageModelDataset(Dataset):
    """
    Causal Language Model Dataset with Strict Target Alignment.
    Fixes implemented:
    1. Returns (input_ids, targets, attention_mask)
    2. Targets are shifted left by 1 (next-token prediction).
    3. Targets corresponding to PAD tokens are set to -100 (ignored by loss).
    4. Targets after the first EOS are set to -100 to prevent learning padding patterns.
    """
    def __init__(self, tokenized_data, pad_token_id: int, eos_token_id: int):
        self.data = tokenized_data
        self.pad_token_id = pad_token_id
        self.eos_token_id = eos_token_id

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        item = self.data[idx]

        # Tensor conversion
        input_ids = torch.tensor(item["input_ids"], dtype=torch.long)
        attention_mask = torch.tensor(item["attention_mask"], dtype=torch.long)
```

```
        # Create Targets (Next Token Prediction)
        targets = input_ids.clone()

        # Shift targets: target for input[i] is input[i+1]
        # The last input token has no target, so we ignore it.
        targets[:-1] = input_ids[1:]
        targets[-1] = -100

        # Mask PAD tokens in targets
        if self.pad_token_id is not None:
            targets[input_ids == self.pad_token_id] = -100

        # Mask everything after the first EOS token to avoid training on pad patterns
        if self.eos_token_id is not None:
            # Find positions of EOS
            is_eos = (input_ids == self.eos_token_id)
            if is_eos.any():
                # Find first EOS index
                eos_idx = (is_eos.cumsum(dim=0) == 1) & is_eos
                first_eos_pos = eos_idx.nonzero(as_tuple=True)[0][0].item()
                # Ignore everything AFTER the EOS (targets should handle the EOS itself)
                # But typically we want the model to predict EOS given the last word.
                # We mask targets *after* the EOS position.
                targets[first_eos_pos + 1:] = -100

        return input_ids, targets, attention_mask

def prepare_data(config):
    print("\n--- Phase 1: Data Preparation ---")
    tokenizer_path = "tinystories-tokenizer-v5.json"

    # 1. Train or Load Tokenizer
    if not os.path.exists(tokenizer_path):
        print("Training BPE Tokenizer on TinyStories...")
        dataset = load_dataset(config['data']['dataset_name'], split="train", streaming=True)

        def batch_iterator(batch_size=1000):
            for i, item in enumerate(dataset):
                yield item['text']
                if i >= 5000: break # Train on subset for speed

        tokenizer = Tokenizer(BPE(unk_token="[UNK]"))
        tokenizer.pre_tokenizer = Whitespace()
```

```
        trainer = BpeTrainer(special_tokens=["[UNK]", "[PAD]", "[EOS]"], vocab_size=8192)
        tokenizer.train_from_iterator(batch_iterator(), trainer=trainer)
        tokenizer.save(tokenizer_path)
    else:
        print("Loading existing tokenizer...")

    # Wrap in HuggingFace wrapper for ease of use
    tokenizer = PreTrainedTokenizerFast(tokenizer_file=tokenizer_path)
    tokenizer.add_special_tokens({'pad_token': '[PAD]', 'eos_token': '[EOS]', 'unk_token': '[UNK]'})

    print(f"Tokenizer Vocab Size: {tokenizer.vocab_size}")

    # 2. Load & Tokenize Dataset
    print("Loading Dataset...")
    full_dataset = load_dataset(config['data']['dataset_name'])

    # Select Subsets
    train_data = full_dataset['train'].select(range(config['data']['train_subset_size']))
    val_data = full_dataset['validation'].select(range(config['data']['val_subset_size']))

    def tokenize_function(examples):
        # Append EOS token to every text
        texts = [t + tokenizer.eos_token for t in examples['text']]
        return tokenizer(
            texts,
            padding="max_length",
            truncation=True,
            max_length=config['model']['seq_len'],
            return_tensors=None # Return lists, handled by Dataset class
        )

    print("Tokenizing (Parallel)...")
    train_tokenized = train_data.map(tokenize_function, batched=True,
num_proc=os.cpu_count(), remove_columns=['text'])
    val_tokenized = val_data.map(tokenize_function, batched=True, num_proc=os.cpu_count(),
remove_columns=['text'])

    # 3. Create Datasets & Loaders
    train_dataset = LanguageModelDataset(train_tokenized, tokenizer.pad_token_id,
tokenizer.eos_token_id)
    val_dataset = LanguageModelDataset(val_tokenized, tokenizer.pad_token_id,
tokenizer.eos_token_id)

    train_loader = DataLoader(
```

```
        train_dataset,
        batch_size=config['data']['batch_size'],
        shuffle=True,
        num_workers=config['data']['num_workers'],
        pin_memory=True
    )

    val_loader = DataLoader(
        val_dataset,
        batch_size=config['data']['batch_size'],
        shuffle=False,
        num_workers=config['data']['num_workers'],
        pin_memory=True
    )

    return train_loader, val_loader, tokenizer

#
================================================================================
======
# 3. SoME ARCHITECTURE (v5.2 Implementation)
#
================================================================================
======

class Expert(nn.Module):
    """
    Standard Feed-Forward Expert.
    Can be initialized sparsely to ensure orthogonality of function.
    """
    def __init__(self, d_model, d_ffn, init_method='default'):
        super().__init__()
        self.w_down = nn.Linear(d_model, d_ffn)
        self.activation = nn.GELU()
        self.w_up = nn.Linear(d_ffn, d_model)

        self._initialize_weights(init_method)

    def _initialize_weights(self, method):
        if method == 'orthogonal':
            nn.init.orthogonal_(self.w_down.weight)
            nn.init.orthogonal_(self.w_up.weight)
        elif method == 'sparse':
            # Sparsity = 0.5 creates distinct jagged transformations
```

```
            nn.init.sparse_(self.w_down.weight, sparsity=0.5)
            nn.init.sparse_(self.w_up.weight, sparsity=0.5)
        elif method == 'default':
            # Standard PyTorch init (Kaiming/Xavier implicit)
            pass

        # Always zero bias for cleaner signal propagation in deep networks
        nn.init.zeros_(self.w_down.bias)
        nn.init.zeros_(self.w_up.bias)

    def forward(self, x):
        # x: [Batch, Seq, d_model]
        return self.w_up(self.activation(self.w_down(x)))

class SOMELayer(nn.Module):
    """
    Self-Organizing Mixture of Experts Layer.
    Implements the "Physics of Routing" (Attraction, Peer-Pull, Inertia).
    """
    def __init__(self, d_model, some_config):
        super().__init__()
        self.d_model = d_model
        self.num_experts = some_config['num_experts']
        self.d_ffn = some_config['d_ffn']
        self.top_k = some_config['top_k']

        # Physics Parameters
        self.alpha = some_config['alpha']
        self.beta = some_config['beta']
        self.delta = some_config['delta']
        self.theta_percentile = some_config['theta_percentile']
        self.warmup_steps = some_config['warmup_steps']
        self.ema_decay = some_config['ema_decay']

        # --- Router Architecture (Fix: Manifold Warping) ---
        router_type = some_config.get('router_type', 'linear')
        if router_type == 'mlp':
            hidden_dim = int(d_model * some_config.get('router_mlp_mult', 2.0))
            self.query_network = nn.Sequential(
                nn.Linear(d_model, hidden_dim),
                nn.GELU(),
                nn.Linear(hidden_dim, d_model)
            )
            # Init MLP router carefully
```

```
            for m in self.query_network.modules():
                if isinstance(m, nn.Linear):
                    nn.init.normal_(m.weight, mean=0.0, std=0.02)
                    nn.init.zeros_(m.bias)
        else:
            self.query_network = nn.Linear(d_model, d_model)

        # --- The Key Store (The Map) ---
        # Initialize keys on unit hypersphere
        keys = torch.randn(self.num_experts, d_model)
        self.register_buffer("key_store", F.normalize(keys, p=2, dim=-1))

        # --- Metrics & Inertia Tracking ---
        self.register_buffer("usage_count", torch.zeros(self.num_experts)) # Selection frequency
        self.register_buffer("usage_mass", torch.zeros(self.num_experts))  # Probability mass
        self.register_buffer("steps", torch.zeros((), dtype=torch.long))

        # --- The Expert Pool ---
        self.experts = nn.ModuleList([
            Expert(d_model, self.d_ffn, some_config['init_method'])
            for _ in range(self.num_experts)
        ])

        # Pre-calculate combinations for Peer-Pull to avoid runtime overhead
        if self.top_k > 1:
            indices = torch.combinations(torch.arange(self.top_k), r=2)
            self.register_buffer("peer_pull_indices", indices)

    def forward(self, x, temperature=1.0):
        # x: [Batch, Seq, d_model]
        batch_size, seq_len, _ = x.shape
        x_flat = x.view(-1, self.d_model)

        # 1. Produce Query (The Explorer)
        # Fix #4: Run routing in FP32 even if mixed precision is on to preserve geometry
        with torch.amp.autocast("cuda", enabled=False):
            x_float = x_flat.float()
            queries_raw = self.query_network(x_float)
            queries = F.normalize(queries_raw, p=2, dim=-1)

            # 2. Similarity Search (The Map)
            # keys are maintained unit-norm (mostly), so dot product is cosine similarity
            scores = torch.matmul(queries, self.key_store.t())
```

```
            # 3. Gating
            top_k_scores, top_k_indices = torch.topk(scores, self.top_k, dim=-1)
            gating_weights = F.softmax(top_k_scores / float(temperature), dim=-1)

        # 4. Dispatch & Combine (Standard MoE Logic)
        # We need to route inputs to experts. Ideally we avoid explicit loops for speed.
        # However, for clarity and stability in this reference implementation, we use
        # the standard permutation method.

        flat_top_k_indices = top_k_indices.view(-1) # [B*S*k]

        # Sort indices to group by expert
        sorted_indices, permutation_map = torch.sort(flat_top_k_indices)
        unique_expert_ids, counts = torch.unique_consecutive(sorted_indices,
return_counts=True)

        # Repeat input for top-k
        flat_inputs = x_flat.repeat_interleave(self.top_k, dim=0) # [B*S*k, d]
        permuted_inputs = flat_inputs[permutation_map]

        # Split by expert load
        split_inputs = torch.split(permuted_inputs, counts.tolist(), dim=0)

        # Process chunks
        output_chunks = []
        for i, expert_id in enumerate(unique_expert_ids):
            expert = self.experts[expert_id]
            # Experts might be frozen or unfrozen depending on experiment phase
            output_chunks.append(expert(split_inputs[i]))

        # Re-assemble
        concatenated_outputs = torch.cat(output_chunks, dim=0)

        # Inverse permutation to restore original order
        inverse_permutation_map = torch.argsort(permutation_map)
        expert_outputs = concatenated_outputs[inverse_permutation_map]

        # Reshape to [B*S, k, d]
        expert_outputs = expert_outputs.view(-1, self.top_k, self.d_model)

        # Weighted sum
        # Ensure gating weights match output dtype (e.g. float16)
        gating_weights_cast = gating_weights.to(expert_outputs.dtype)
        output_flat = (expert_outputs * gating_weights_cast.unsqueeze(-1)).sum(dim=1)
```

```
    output = output_flat.view(batch_size, seq_len, self.d_model)

    # Return aux data for heuristic updates
    return output + x, queries, top_k_indices, gating_weights

@torch.no_grad()
def update_keys(self, queries, top_k_indices, gating_weights):
    """
    The Knowledge Gravity Update Step.
    Runs entirely without gradients.
    """
    self.steps += 1

    # Skip update if batch is empty (edge case)
    if queries.shape[0] == 0: return

    # --- 1. Update Usage Statistics (Inertia) ---
    flat_indices = top_k_indices.view(-1)

    # Count based usage
    unique_idx, counts = torch.unique(flat_indices, return_counts=True)
    counts_float = counts.float()
    self.usage_count.mul_(self.ema_decay)
    self.usage_count.index_add_(0, unique_idx, (1.0 - self.ema_decay) * counts_float)

    # Mass based usage (Fix #7 + #8)
    flat_weights = gating_weights.view(-1).float()
    mass_update = torch.zeros_like(self.usage_mass)
    mass_update.index_add_(0, flat_indices, flat_weights)
    self.usage_mass.mul_(self.ema_decay)
    self.usage_mass.add_((1.0 - self.ema_decay) * mass_update)

    # Define Inertia (Resistance to movement) based on count
    inertia_source = self.usage_count

    # --- 2. Heuristic: Attraction (Alpha) ---
    # "Experts move towards the data that selects them."
    if self.alpha > 0:
        for k in range(self.top_k):
            # We process the k-th choice for all tokens
            k_indices = top_k_indices[:, k] # [Batch*Seq]

            # Calculate inertia for these specific experts
```

```
        current_inertia = 1.0 + inertia_source[k_indices]

        # Scale alpha by inertia (Heavy experts move slower)
        alpha_eff = self.alpha / current_inertia.unsqueeze(-1)

        # Vector from Key to Query
        # query [N, d] - key [N, d]
        update_vec = queries - self.key_store[k_indices]

        # Apply update
        self.key_store.index_add_(0, k_indices, alpha_eff * update_vec)

# --- 3. Heuristic: Peer-Pull (Beta) ---
# "Co-activated experts pull towards each other."
if self.beta > 0 and self.top_k > 1:
    # We use the pre-calculated combinations indices
    idx_a_ptr = self.peer_pull_indices[:, 0]
    idx_b_ptr = self.peer_pull_indices[:, 1]

    # Get actual expert IDs
    experts_a = top_k_indices[:, idx_a_ptr].reshape(-1)
    experts_b = top_k_indices[:, idx_b_ptr].reshape(-1)

    keys_a = self.key_store[experts_a]
    keys_b = self.key_store[experts_b]

    # Inertia: Use MAX inertia (Fix #6)
    # Heavy experts should anchor light experts, not be dragged by them.
    inertia_a = (1.0 + inertia_source[experts_a]).unsqueeze(-1)
    inertia_b = (1.0 + inertia_source[experts_b]).unsqueeze(-1)
    joint_inertia = torch.max(inertia_a, inertia_b)

    beta_eff = self.beta / joint_inertia

    # Pull A towards B, and B towards A
    update_a = beta_eff * (keys_b - keys_a)
    update_b = beta_eff * (keys_a - keys_b)

    self.key_store.index_add_(0, experts_a, update_a)
    self.key_store.index_add_(0, experts_b, update_b)

# --- 4. Normalization & Decay ---
# Renormalize keys to stay on hypersphere
self.key_store.copy_(F.normalize(self.key_store, p=2, dim=-1))
```

```
        # Decay (Delta) - Prune unused experts
        if self.delta > 0 and self.steps > self.warmup_steps:
            # Identify low-usage experts
            active_counts = self.usage_count[self.usage_count > 0]
            if active_counts.numel() > 0:
                threshold = torch.quantile(active_counts, self.theta_percentile)
                decay_mask = self.usage_count < threshold

                # Shrink their vectors (reduces dot product score naturally)
                self.key_store[decay_mask] *= (1.0 - self.delta)

class SOMETransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, some_config, dropout=0.1):
        super().__init__()
        self.norm1 = nn.LayerNorm(d_model)
        self.attn = nn.MultiheadAttention(d_model, num_heads, dropout=dropout, batch_first=True)
        self.norm2 = nn.LayerNorm(d_model)
        self.some_layer = SOMELayer(d_model, some_config)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None, temperature=1.0):
        # Attention Sublayer
        res = x
        x = self.norm1(x)

        # Causal Masking (Fix #1 & #2) applied in main model
        attn_out, _ = self.attn(x, x, x, attn_mask=mask, need_weights=False, is_causal=True)
        x = res + self.dropout(attn_out)

        # SoME Sublayer
        res = x
        x = self.norm2(x)
        some_out, queries, indices, gates = self.some_layer(x, temperature)
        x = res + self.dropout(some_out) # SoME output already includes residual path logic in
forward?
        # Wait, SOMELayer.forward returns x + output.
        # Let's adjust SOMELayer to return raw output to be standard.
        # Actually looking at SOMELayer code above: `return output + x`.
        # Standard transformer block usually does: x = x + layer(norm(x))
        # The SOMELayer code above implements the residual internally.
        # To strictly follow Pre-Norm:
        # Input to SOMELayer is norm2(x). Output is norm2(x) + Expert(norm2(x)).
        # We need to handle the outer residual carefully.
```

```
        # Let's assume SOMELayer returns `x_in + expert_out`.
        # So here x becomes `norm2(x) + expert_out`.
        # We want `res + expert_out`.
        # So: x = res + (some_out - input_to_some)
        # To avoid confusion, let's trust SOMELayer returns specific logic.
        # Current SOMELayer returns: input + weighted_experts.
        # So if we pass norm2(x), we get norm2(x) + experts.
        # We want x_new = res + experts.
        # So x_new = res + (some_out - norm2(x))
        # This is messy. Let's simplify SOMELayer return in this script context.
        # NOTE: For this specific script, I will trust the SOMELayer above returns `x + expert_out`.
        # So `some_out` is `norm2(x) + expert_val`.
        # We want `res + expert_val`.
        # So `expert_val = some_out - norm2(x_in)`.

        expert_contribution = some_out - x # (norm2(x) + E) - norm2(x) = E
        x = res + self.dropout(expert_contribution)

        return x, queries, indices, gates

class SOMETransformer(nn.Module):
    def __init__(self, model_config, some_config):
        super().__init__()
        self.d_model = model_config['d_model']
        self.embedding = nn.Embedding(model_config['vocab_size'], self.d_model)
        self.pos_encoder = nn.Parameter(torch.randn(1, model_config['seq_len'], self.d_model) *
0.02)
        self.dropout = nn.Dropout(model_config['dropout'])

        self.layers = nn.ModuleList([
            SOMETransformerBlock(self.d_model, model_config['num_heads'], some_config,
model_config['dropout'])
            for _ in range(model_config['num_layers'])
        ])

        self.norm_f = nn.LayerNorm(self.d_model)
        self.head = nn.Linear(self.d_model, model_config['vocab_size'], bias=False)

        # Tie weights
        self.head.weight = self.embedding.weight

        # Causal mask buffer
        mask = torch.triu(torch.ones(model_config['seq_len'], model_config['seq_len']) * float('-inf'),
diagonal=1)
```

```
        self.register_buffer("causal_mask", mask)

    def forward(self, x, temperature=1.0):
        B, T = x.shape

        # Embed + Pos
        x = self.embedding(x) + self.pos_encoder[:, :T, :]
        x = self.dropout(x)

        # Layers
        all_queries, all_indices, all_gates = [], [], []

        # Use registered buffer for mask
        mask = self.causal_mask[:T, :T]

        for layer in self.layers:
            x, q, i, g = layer(x, mask=mask, temperature=temperature)
            all_queries.append(q)
            all_indices.append(i)
            all_gates.append(g)

        x = self.norm_f(x)
        logits = self.head(x)

        return logits, all_queries, all_indices, all_gates

    @torch.no_grad()
    def update_all_keys(self, all_queries, all_indices, all_gates, valid_mask=None):
        """Update keys for all layers."""
        for layer_idx, layer in enumerate(self.layers):
            q = all_queries[layer_idx]
            i = all_indices[layer_idx]
            g = all_gates[layer_idx]

            # Apply mask if provided (ignore PAD/EOS positions)
            if valid_mask is not None:
                # valid_mask is [B, T]
                # q is [B, T, d]
                # Flatten
                mask_flat = valid_mask.view(-1)
                q = q.view(-1, q.size(-1))[mask_flat]
                i = i.view(-1, i.size(-1))[mask_flat]
                g = g.view(-1, g.size(-1))[mask_flat]
            else:
```

```
            q = q.view(-1, q.size(-1))
            i = i.view(-1, i.size(-1))
            g = g.view(-1, g.size(-1))

        layer.some_layer.update_keys(q, i, g)

#
================================================================================
======
# 4. TRAINING ENGINE
#
================================================================================
======

def train_epoch(model, loader, optimizer, criterion, scheduler, temp, scaler, config):
    model.train()
    total_loss = 0

    # We need to access the underlying model for key updates if using torch.compile
    model_access = model._orig_mod if hasattr(model, '_orig_mod') else model

    pbar = tqdm(loader, desc="Training", leave=False)
    for input_ids, targets, _ in pbar:
        input_ids, targets = input_ids.to(device), targets.to(device)

        # Mixed Precision Context
        with torch.amp.autocast("cuda", enabled=config['training']['use_amp']):
            logits, queries, indices, gates = model(input_ids, temperature=temp)

            # Flatten for loss
            loss = criterion(logits.view(-1, config['model']['vocab_size']), targets.view(-1))

        # Optimization
        optimizer.zero_grad(set_to_none=True)
        scaler.scale(loss).backward()
        scaler.unscale_(optimizer)
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        scaler.step(optimizer)
        scaler.update()
        scheduler.step()

        # Heuristic Update (No Grad)
        # Create mask for valid tokens (where target != -100)
        valid_mask = (targets != -100)
```

```
        model_access.update_all_keys(queries, indices, gates, valid_mask)

        total_loss += loss.item()
        pbar.set_postfix(loss=loss.item(), lr=scheduler.get_last_lr()[0])

    return total_loss / len(loader)

@torch.no_grad()
def evaluate(model, loader, criterion, temp, config):
    model.eval()
    total_loss = 0

    for input_ids, targets, _ in loader:
        input_ids, targets = input_ids.to(device), targets.to(device)

        with torch.amp.autocast("cuda", enabled=config['training']['use_amp']):
            logits, _, _, _ = model(input_ids, temperature=temp)
            loss = criterion(logits.view(-1, config['model']['vocab_size']), targets.view(-1))

        total_loss += loss.item()

    return total_loss / len(loader)

def calculate_gini(counts):
    counts = counts.float().cpu().numpy()
    if np.sum(counts) == 0: return 0.0
    counts = np.sort(counts)
    n = len(counts)
    index = np.arange(1, n + 1)
    return ((2 * index - n - 1) * counts).sum() / (n * counts.sum())

#
================================================================================
======
# 5. MAIN EXECUTION (THE THAW EXPERIMENT)
#
================================================================================
======

#
================================================================================
======
# 5. MAIN EXECUTION (THE THAW EXPERIMENT) - FIXED
```

```
#
================================================================================
======

def main():
    print(f"Starting Experiment: {CONFIG['run_name']}")

    # 1. Data
    train_loader, val_loader, tokenizer = prepare_data(CONFIG)
    CONFIG['model']['vocab_size'] = tokenizer.vocab_size

    # 2. Model
    print("\n--- Phase 2: Initialization (Frozen) ---")
    model = SOMETransformer(CONFIG['model'], CONFIG['some_layer']).to(device)

    # FREEZE EXPERTS
    print("Freezing Expert Libraries...")
    frozen_params = 0
    for name, param in model.named_parameters():
        if "experts" in name:
            param.requires_grad = False
            frozen_params += param.numel()

    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    print(f"Total Params: {total_params/1e6:.2f}M")
    print(f"Trainable Params: {trainable_params/1e6:.2f}M")
    print(f"Frozen Params: {frozen_params/1e6:.2f}M")

    if torch.cuda.get_device_capability()[0] >= 7:
        print("Compiling model...")
        model = torch.compile(model)

    # 3. Optimization Setup
    optimizer = torch.optim.AdamW(
        [p for p in model.parameters() if p.requires_grad],
        lr=CONFIG['training']['learning_rate'],
        weight_decay=CONFIG['training']['weight_decay']
    )

    criterion = nn.CrossEntropyLoss(ignore_index=-100)
    scaler = torch.amp.GradScaler("cuda", enabled=CONFIG['training']['use_amp'])

    total_steps = len(train_loader) * CONFIG['training']['num_epochs']
```

```
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=total_steps)

history = {'train_loss': [], 'val_loss': [], 'ppl': []}

# 4. Training Loop
thaw_epoch = CONFIG['training']['thaw_at_epoch']

for epoch in range(CONFIG['training']['num_epochs']):
    print(f"\nEpoch {epoch+1}/{CONFIG['training']['num_epochs']}")

    # === THE THAW EVENT ===
    if epoch == thaw_epoch:
        print("\n" + "!"*50)
        print("❄️  THE THAW: Unfreezing 1.6 Billion Parameters  🔥")
        print("!"*50)

        # Access underlying model
        model_ref = model._orig_mod if hasattr(model, '_orig_mod') else model

        # Unfreeze everything
        unfrozen_count = 0
        for layer in model_ref.layers:
            for expert in layer.some_layer.experts:
                for param in expert.parameters():
                    param.requires_grad = True
                    unfrozen_count += param.numel()
        print(f"Unfrozen {unfrozen_count/1e6:.2f}M expert parameters.")

        # Re-initialize Optimizer
        print("Re-building Optimizer...")

        # 1. Collect all trainable parameters
        all_trainable_params = [p for p in model.parameters() if p.requires_grad]

        # 2. Create new optimizer
        optimizer = torch.optim.AdamW(
            all_trainable_params,
            lr=CONFIG['training']['learning_rate'],
            weight_decay=CONFIG['training']['weight_decay']
        )

        # 3. CRITICAL FIX: Manually set initial_lr for the scheduler
        # The scheduler needs this to calculate the decay from the correct starting point.
        for group in optimizer.param_groups:
```

```
            group['initial_lr'] = CONFIG['training']['learning_rate']

        # 4. Re-attach scheduler
        current_step = epoch * len(train_loader)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
            optimizer, T_max=total_steps, last_epoch=current_step - 1
        )
        print("Thaw Complete. Resuming Training.")

    # Train
    t_loss = train_epoch(
        model, train_loader, optimizer, criterion, scheduler,
        CONFIG['training']['training_temp'], scaler, CONFIG
    )

    # Evaluate
    v_loss = evaluate(
        model, val_loader, criterion,
        CONFIG['training']['eval_temp'], CONFIG
    )

    # Safe PPL calculation to avoid overflow printing
    try:
        ppl = math.exp(v_loss)
    except OverflowError:
        ppl = float('inf')

    history['train_loss'].append(t_loss)
    history['val_loss'].append(v_loss)
    history['ppl'].append(ppl)

    # Metrics Inspection
    model_ref = model._orig_mod if hasattr(model, '_orig_mod') else model
    mid_layer = model_ref.layers[CONFIG['model']['num_layers'] // 2].some_layer
    gini = calculate_gini(mid_layer.usage_mass)
    dead = (mid_layer.usage_count == 0).sum().item()

    print(f"Result: Train={t_loss:.4f} | Val={v_loss:.4f} | PPL={ppl:.2f}")
    print(f"Topology: Middle Layer Gini={gini:.3f} | Dead Experts={dead}")

# 5. Plotting
plt.figure(figsize=(10, 6))
plt.plot(history['train_loss'], label='Train Loss')
plt.plot(history['val_loss'], label='Val Loss')
```

```
    plt.axvline(x=thaw_epoch, color='r', linestyle='--', label='The Thaw')
    plt.title("SoME 'The Thaw' Experiment: Loss Trajectory")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.savefig("thaw_experiment_loss.png")
    print("\nExperiment Complete. Plot saved.")

if __name__ == "__main__":
    main()
```