

## SoME v5 Official Ablations Code (IPYNB)

---

Cell One:

#

=====

=====

# Cell 1: Setup and Dependencies

#

=====

=====

print("--- Part 1: Setup and Dependencies ---")

!pip install torch datasets transformers huggingface\_hub tokenizers matplotlib -q

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.utils.data import DataLoader, Dataset

from transformers import PreTrainedTokenizerFast

from tokenizers import Tokenizer

from tokenizers.models import BPE

from tokenizers.trainers import BpeTrainer

from tokenizers.pre\_tokenizers import Whitespace

from datasets import load\_dataset

import copy

from tqdm import tqdm

import math

import os

import numpy as np

import matplotlib.pyplot as plt

# Verify that a GPU is available and set the device

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

print(f"Using device: {device}")

# Enable TF32 for A100 GPUs for a free performance boost

if torch.cuda.is\_available() and torch.cuda.get\_device\_capability(0)[0] >= 8:

print("A100 GPU detected. Enabling TF32.")

torch.set\_float32\_matmul\_precision('high')

# Enable benchmark mode for cuDNN

torch.backends.cudnn.benchmark = True

---

Cell Two:

```

#
=====
=====
# Cell 2: Core SoME Framework (FIXED)
# (Classes, Data Functions, Training Loop)
#
=====
=====
#
# Fixes implemented (the "big 8"):
# 1) Mask PAD tokens in loss targets
# 2) Prevent attention from attending to PAD (key_padding_mask)
# 3) Force a single vocab-size source of truth (tokenizer -> config -> model/loss)
# 4) Run routing + key updates in FP32 under AMP (preserve SoME geometry)
# 5) Standardize eval temperature (no hidden sharpening)
# 6) Beta inertia uses MAX mass (heavy experts resist movement)
# 7) Normalize usage/inertia updates by token flux (batch/seq/top_k comparable)
# 8) Track inequality both by selection-count and by gating-weight mass
#
# (Everything else is kept intentionally close to your baseline.)
# -----

```

# --- 1. Model Component Classes ---

```

class Expert(nn.Module):
    """An expert module with configurable random weight initialization."""
    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)

        if init_method == 'orthogonal':
            nn.init.orthogonal_(self.w_down.weight)
            nn.init.orthogonal_(self.w_up.weight)
        elif init_method == 'sparse':
            nn.init.sparse_(self.w_down.weight, sparsity=0.5)
            nn.init.sparse_(self.w_up.weight, sparsity=0.5)
        elif init_method != 'default':
            raise ValueError(f"Unknown initialization method: {init_method}")

        nn.init.zeros_(self.w_down.bias)
        nn.init.zeros_(self.w_up.bias)

```

```
def forward(self, x):
    return self.w_up(self.activation(self.w_down(x)))
```

```
class SOMELayer(nn.Module):
    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']

        # Heuristic update parameters
        self.alpha = some_config['alpha']
        self.beta = some_config['beta']
        self.delta = some_config['delta']

        # Key management parameters
        self.theta_percentile = some_config['theta_percentile']
        self.warmup_steps = some_config['warmup_steps']
        self.ema_decay = some_config['ema_decay']

        # Ablation flags
        self.ablation_flags = some_config.get(
            'ablation_flags',
            {'use_alpha': True, 'use_beta': True, 'use_delta': True}
        )
        print(f"SoME Layer Ablation Flags: {self.ablation_flags}")

        # Router / query network (trainable)
        # Toggle via config['model']['router_type'] in the Ablation/Config cell.
        self.router_type = (some_config.get("router_type", "linear") or "linear").lower()
        if self.router_type == "linear":
            self.query_network = nn.Linear(d_model, d_model, bias=True)
        elif self.router_type in ("mlp", "mlp_router", "mlp-router"):
            mult = float(some_config.get("router_mlp_mult", 1.0))
            hidden = max(1, int(d_model * mult))
            dropout = float(some_config.get("router_dropout", 0.0))
            layers = [nn.Linear(d_model, hidden, bias=True), nn.GELU()]
            if dropout > 0.0:
                layers.append(nn.Dropout(dropout))
            layers.append(nn.Linear(hidden, d_model, bias=True))
            self.query_network = nn.Sequential(*layers)
        else:
```

```

        raise ValueError(f"Unknown router_type={self.router_type!r} (use 'linear' or 'mlp')")
    # Init query network (match common transformer init for each Linear)
    for m in self.query_network.modules():
        if isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, mean=0.0, std=0.02)
            if m.bias is not None:
                nn.init.zeros_(m.bias)

    keys = torch.randn(self.num_experts, d_model)
    self.register_buffer("key_store", F.normalize(keys, p=2, dim=-1))

    # Fix #8 + #7: track both selection-frequency and gating-mass, normalized by token flux
    self.register_buffer("usage_count", torch.zeros(self.num_experts)) # EMA of selection
probability mass
    self.register_buffer("usage_mass", torch.zeros(self.num_experts)) # EMA of gating-weight
probability mass

    # Fix #14 from earlier notes is not part of the "big 8", but we make steps a scalar for clarity.
    self.register_buffer("steps", torch.zeros((), dtype=torch.long))

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

    # Freeze all expert parameters
    for expert in self.experts:
        for param in expert.parameters():
            param.requires_grad = False

    if self.top_k > 1:
        self.register_buffer("peer_pull_indices", torch.combinations(torch.arange(self.top_k),
r=2))

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

        # Fix #4: routing computations in fp32 even under AMP
        with torch.cuda.amp.autocast(enabled=False):
            queries_raw = self.query_network(x_flat.float())
            queries = F.normalize(queries_raw, p=2, dim=-1)

```

```

        # cosine-style similarity (keys are unit norm unless delta is enabled; baseline behavior
        preserved)
        scores = torch.matmul(queries, self.key_store.t())

        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)

    # Expert dispatch (can run under AMP)
    flat_top_k_indices = top_k_indices.view(-1)
    sorted_indices, permutation_map = torch.sort(flat_top_k_indices)
    unique_expert_ids, counts = torch.unique_consecutive(sorted_indices,
    return_counts=True)

    flat_inputs = x_flat.repeat_interleave(self.top_k, dim=0)
    permuted_inputs = flat_inputs[permutation_map]
    split_inputs = torch.split(permuted_inputs, counts.tolist(), dim=0)

    output_chunks = []
    for i, expert_id in enumerate(unique_expert_ids):
        output_chunks.append(self.experts[expert_id](split_inputs[i]))

    concatenated_outputs = torch.cat(output_chunks, dim=0)
    inverse_permutation_map = torch.argsort(permutation_map)
    expert_outputs = concatenated_outputs[inverse_permutation_map]

    # Ensure dtype compatibility in the weighted sum
    gating_weights_for_mul = gating_weights.to(expert_outputs.dtype)
    weighted_outputs = (expert_outputs.view(-1, self.top_k, self.d_model) *
    gating_weights_for_mul.unsqueeze(-1)).sum(dim=1)
    final_output = weighted_outputs.view(batch_size, seq_len, self.d_model)

    return x + final_output, queries, top_k_indices, gating_weights

@torch.no_grad()
def update_keys(self, queries, top_k_indices, gating_weights):
    self.steps += 1

    # queries: [T, d_model], top_k_indices: [T, top_k], gating_weights: [T, top_k]
    num_tokens = top_k_indices.shape[0]
    if num_tokens == 0:
        return

    # -----
    # Usage EMAs (raw counts + gating-weight mass)

```

```

# -----
flat_idx = top_k_indices.reshape(-1)
unique_indices, counts = torch.unique(flat_idx, return_counts=True)
counts_f = counts.to(torch.float32)

self.usage_count.mul_(self.ema_decay)
self.usage_count.index_add_(0, unique_indices, (1.0 - self.ema_decay) * counts_f)

if gating_weights is not None:
    flat_w = gating_weights.reshape(-1).to(torch.float32)
    mass_sums = torch.zeros_like(self.usage_mass)
    mass_sums.index_add_(0, flat_idx, flat_w)

    self.usage_mass.mul_(self.ema_decay)
    self.usage_mass.add_((1.0 - self.ema_decay) * mass_sums)

# Inertia source for key updates (count-based)
inertia_source = self.usage_count

# --- Heuristic 1: Alpha (Attraction) ---
if self.ablation_flags.get('use_alpha', True):
    for i in range(self.top_k):
        indices = top_k_indices[:, i]
        inertia = 1.0 + inertia_source[indices]
        alpha_effective = self.alpha / inertia.unsqueeze(-1)
        update_vec = queries - self.key_store[indices]
        self.key_store.index_add_(0, indices, alpha_effective * update_vec)

# --- Heuristic 2: Beta (Peer Pull / Clustering) ---
if self.top_k > 1 and self.ablation_flags.get('use_beta', True):
    indices_i = top_k_indices[:, self.peer_pull_indices[:, 0]].reshape(-1)
    indices_j = top_k_indices[:, self.peer_pull_indices[:, 1]].reshape(-1)
    keys_i, keys_j = self.key_store[indices_i], self.key_store[indices_j]

    inertia_i = (1.0 + inertia_source[indices_i]).unsqueeze(-1)
    inertia_j = (1.0 + inertia_source[indices_j]).unsqueeze(-1)

    # Fix #6: heavy experts resist movement (use MAX inertia, not MIN)
    beta_effective = self.beta / torch.max(inertia_i, inertia_j)

    update_vec_i = beta_effective * (keys_j - keys_i)
    update_vec_j = beta_effective * (keys_i - keys_j)
    self.key_store.index_add_(0, indices_i, update_vec_i)
    self.key_store.index_add_(0, indices_j, update_vec_j)

```

```

# Keep baseline normalization behavior
self.key_store.copy_(F.normalize(self.key_store, p=2, dim=-1))

# --- Heuristic 3: Delta (Decay / "Use it or Lose it") ---
# (Left intentionally consistent with your baseline; note: this changes key norms.)
if self.steps.item() > self.warmup_steps and self.ablation_flags.get('use_delta', True):
    active_usage_counts = self.usage_count[self.usage_count > 0]
    if active_usage_counts.numel() > 0:
        dynamic_theta = torch.quantile(active_usage_counts.float(), self.theta_percentile)
        low_usage_mask = self.usage_count < dynamic_theta
        self.key_store[low_usage_mask] *= (1.0 - self.delta)

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

    def forward(self, x, attention_mask=None, temperature=1.0):
        seq_len = x.size(1)
        causal_mask = torch.triu(torch.ones(seq_len, seq_len, device=x.device),
diagonal=1).bool()

        # Fix #2: prevent attention from seeing PAD positions
        key_padding_mask = None
        if attention_mask is not None:
            key_padding_mask = (attention_mask == 0)

        attn_output, _ = self.attention(x, x, x, attn_mask=causal_mask,
key_padding_mask=key_padding_mask)
        x = self.norm1(x + attn_output)

        some_output, queries, top_k_indices, gating_weights = self.some_layer(x,
temperature=temperature)
        x = self.norm2(some_output)
        return x, queries, top_k_indices, gating_weights

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):

```

```

super().__init__()
position = torch.arange(max_len).unsqueeze(1)
div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
pe = torch.zeros(1, max_len, d_model)
pe[0, :, 0::2] = torch.sin(position * div_term)
pe[0, :, 1::2] = torch.cos(position * div_term)
self.register_buffer('pe', pe)

```

```

def forward(self, x):
    return x + self.pe[:, :x.size(1)]

```

```

class SOMETransformer(nn.Module):
    def __init__(self, model_config, some_config):
        super().__init__()
        self.embedding = nn.Embedding(model_config['vocab_size'], model_config['d_model'])
        self.pos_encoder = PositionalEncoding(model_config['d_model'], model_config['seq_len'])
        self.layers = nn.ModuleList([
            SOMETransformerBlock(model_config['d_model'], model_config['num_heads'],
some_config)
            for _ in range(model_config['num_layers'])
        ])
        self.fc_out = nn.Linear(model_config['d_model'], model_config['vocab_size'])
        self.d_model = model_config['d_model']

    def forward(self, x, attention_mask=None, temperature=1.0):
        x = self.embedding(x) * math.sqrt(self.d_model)
        x = self.pos_encoder(x)

        all_queries, all_indices, all_gates = [], [], []
        for layer in self.layers:
            x, queries, top_k_indices, gating_weights = layer(x, attention_mask=attention_mask,
temperature=temperature)
            all_queries.append(queries)
            all_indices.append(top_k_indices)
            all_gates.append(gating_weights)

        return self.fc_out(x), all_queries, all_indices, all_gates

```

```

@torch.no_grad()
def update_all_keys(self, all_queries, all_indices, all_gates, token_mask=None):
    """Update each layer's key store.

```



```

token_mask (optional):
    - shape [B, T] or [B*T]
    - True where the token position is valid for key updates (e.g., targets != -100).
    """
    if token_mask is not None:
        if token_mask.dim() == 2:
            token_mask = token_mask.reshape(-1)
            token_mask = token_mask.to(dtype=torch.bool, device=all_indices[0].device)

    for layer, q, idx, g in zip(self.layers, all_queries, all_indices, all_gates):
        if token_mask is not None:
            if q is not None:
                q = q[token_mask]
            if idx is not None:
                idx = idx[token_mask]
            if g is not None:
                g = g[token_mask]
            layer.some_layer.update_keys(q, idx, g)

```

# --- 2. Data Preparation ---

```

class LanguageModelDataset(Dataset):
    """Causal LM dataset with STRICT targets:
    - next-token prediction
    - ignore PAD targets
    - ignore all positions at/after the first EOS in the input (so we don't learn to predict
    PAD/garbage after EOS)
    Returns: (input_ids, targets, attention_mask)
    """
    def __init__(self, tokenized_data, pad_token_id: int, eos_token_id: int = None):
        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]
        input_ids = torch.tensor(item["input_ids"], dtype=torch.long)
        # Use provided attention_mask if available; otherwise derive from PAD.
        if "attention_mask" in item:

```

```

        attention_mask = torch.tensor(item["attention_mask"], dtype=torch.long)
    else:
        attention_mask = (input_ids != self.pad_token_id).long()

    # Next-token targets (shift left)
    targets = input_ids.clone()
    targets[:-1] = input_ids[1:]
    targets[-1] = -100 # last position has no next token

    # Ignore PAD targets (prevents PAD-dominated objective)
    if self.pad_token_id is not None:
        targets[targets == self.pad_token_id] = -100

    # Ignore targets at/after first EOS in the input (keeps EOS prediction itself!)
    if self.eos_token_id is not None:
        eos_cum = (input_ids == self.eos_token_id).cumsum(dim=0)
        targets[eos_cum > 0] = -100

    return input_ids, targets, attention_mask

```

```

def prepare_data(config):
    print("\n--- Part 2: Data Preparation & Configuration ---")
    tokenizer_path = "tinystories-tokenizer-v2.json"

    if not os.path.exists(tokenizer_path):
        print("Training custom tokenizer...")
        dataset = load_dataset("roneneldan/TinyStories", split="train")

        def get_training_corpus():
            for i in range(0, len(dataset), 1000):
                yield dataset[i: i + 1000]["text"]

        tokenizer_model = Tokenizer(BPE(unk_token="[UNK]"))
        tokenizer_model.pre_tokenizer = Whitespace()
        trainer = BpeTrainer(
            special_tokens=["[UNK]", "[PAD]", "[EOS]"],
            vocab_size=config['model']['vocab_size']
        )
        tokenizer_model.train_from_iterator(get_training_corpus(), trainer=trainer)
        tokenizer_model.save(tokenizer_path)
    else:
        print("Tokenizer already exists. Loading from file.")

```

```

tokenizer = PreTrainedTokenizerFast(tokenizer_file=tokenizer_path)
tokenizer.add_special_tokens({'pad_token': '[PAD]', 'eos_token': '[EOS]'})
print(f'Custom tokenizer loaded with vocab size: {tokenizer.vocab_size}')

# Fix #3: single source of truth for vocab size
config['model']['vocab_size'] = tokenizer.vocab_size

print("\nTokenizing dataset...")
full_dataset = load_dataset("roneneldan/TinyStories", streaming=False)
train_subset = full_dataset['train'].select(range(config['data']['train_subset_size']))
val_subset = full_dataset['validation'].select(range(config['data']['val_subset_size']))

def tokenize_function(examples):
    text_with_eos = [s + tokenizer.eos_token for s in examples["text"]]
    return tokenizer(
        text_with_eos,
        truncation=True,
        padding="max_length",
        max_length=config['model']['seq_len'],
        return_tensors="pt"
    )

tokenized_train = train_subset.map(tokenize_function, batched=True,
remove_columns=["text"], num_proc=os.cpu_count())
tokenized_val = val_subset.map(tokenize_function, batched=True, remove_columns=["text"],
num_proc=os.cpu_count())

train_dataset = LanguageModelDataset(tokenized_train,
pad_token_id=tokenizer.pad_token_id)
validation_dataset = LanguageModelDataset(tokenized_val,
pad_token_id=tokenizer.pad_token_id)

num_workers = max(2, os.cpu_count() // 2 if os.cpu_count() else 2)
train_loader = DataLoader(
    train_dataset,
    batch_size=config['data']['batch_size'],
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    pin_memory=True
)
validation_loader = DataLoader(
    validation_dataset,
    batch_size=config['data']['batch_size'],

```

```

        drop_last=False, # keep full validation set
        num_workers=num_workers,
        pin_memory=True
    )

    print(f"\nTrain dataset size (subset): {len(train_dataset)}")
    print(f"Using {num_workers} workers for DataLoader.")
    return train_loader, validation_loader, tokenizer

```

# --- 3. Training & Evaluation Functions ---

```

def calculate_gini(usage_counts):
    counts = usage_counts.cpu().to(torch.float32).numpy()
    if np.sum(counts) == 0:
        return 0.0
    counts = np.sort(counts)
    n = len(counts)
    index = np.arange(1, n + 1)
    return (np.sum((2 * index - n - 1) * counts)) / (n * np.sum(counts))

```

```

def calculate_entropy(usage_counts):
    total_usage = usage_counts.sum()
    if total_usage == 0:
        return 0.0
    probs = usage_counts / total_usage
    probs = probs[probs > 0]
    return -torch.sum(probs * torch.log2(probs)).item()

```

```

def train_epoch(model, dataloader, optimizer, criterion, scheduler, current_temp, vocab_size):
    model.train()
    total_loss = 0
    scaler = torch.cuda.amp.GradScaler()
    progress_bar = tqdm(dataloader, desc="Training", leave=False)

    for input_ids, targets, attention_mask in progress_bar:
        input_ids = input_ids.to(device, non_blocking=True)
        targets = targets.to(device, non_blocking=True)
        attention_mask = attention_mask.to(device, non_blocking=True)

        with torch.cuda.amp.autocast():

```

```

        logits, queries, indices, gates = model(input_ids, attention_mask=attention_mask,
        temperature=current_temp)
        loss = criterion(logits.view(-1, vocab_size), targets.view(-1))

        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()

        # Fix #4: ensure key updates happen on the underlying module when compiled
        model_state = model._orig_mod if hasattr(model, "_orig_mod") else model
        model_state.update_all_keys(queries, indices, gates, token_mask=(targets.view(-1) !=
        -100))

        total_loss += loss.item()
        progress_bar.set_postfix({'loss': f'{loss.item():.4f}', 'lr': f'{scheduler.get_last_lr()[0]:.1e}'})

    return total_loss / len(dataloader)

```

```

def evaluate_epoch(model, dataloader, criterion, vocab_size, eval_temp: float):
    model.eval()
    total_loss = 0
    progress_bar = tqdm(dataloader, desc="Evaluating", leave=False)

    with torch.no_grad():
        for input_ids, targets, attention_mask in progress_bar:
            input_ids = input_ids.to(device, non_blocking=True)
            targets = targets.to(device, non_blocking=True)
            attention_mask = attention_mask.to(device, non_blocking=True)

            with torch.cuda.amp.autocast():
                # Fix #5: eval temperature is explicit and standardized
                logits, _, _ = model(input_ids, attention_mask=attention_mask,
                temperature=eval_temp)
                loss = criterion(logits.view(-1, vocab_size), targets.view(-1))

            total_loss += loss.item()
            progress_bar.set_postfix({'loss': f'{loss.item():.4f}'})

    return total_loss / len(dataloader)

```

```

def plot_losses(train_losses, val_losses, config):
    epochs = len(train_losses)
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, epochs + 1), train_losses, 'b-o', label='Training Loss')
    plt.plot(range(1, epochs + 1), val_losses, 'r-o', label='Validation Loss')
    title = f"SoME v3.0 (FIXED) Run: {config['run_name']}"
    plt.title(title)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.xticks(range(1, epochs + 1))
    filename = f"loss_curve_{config['run_name']}_fixed.png"
    plt.savefig(filename)
    print(f"\nLoss curve plot saved to {filename}")
    plt.show()

```

# --- 4. Main Execution Function ---

```

def main(config):
    """Main function to run a SoME experiment."""
    print(f"\n--- Starting Experiment: {config['run_name']} ---")

    # 1. Data
    train_loader, val_loader, tokenizer = prepare_data(config)

    # 2. Model Initialization
    print("\n--- Part 3: Model Definition ---")
    model = SOMETransformer(config['model'], config['some_layer']).to(device)

    if hasattr(torch, 'compile'):
        print("\nCompiling the model for faster training...")
        model = torch.compile(model)

    # 3. Training Setup
    print("\n--- Part 4: Training, Evaluation, and Metrics ---")
    optimizer = torch.optim.AdamW(
        [p for p in model.parameters() if p.requires_grad],
        lr=config['training']['learning_rate'],
        betas=(0.9, 0.95),
        weight_decay=0.1
    )

```

```

)
criterion = nn.CrossEntropyLoss(ignore_index=-100)
total_steps = len(train_loader) * config['training']['num_epochs']
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=total_steps)

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"\nTotal parameters: {total_params/1e6:.2f}M")
print(f"Trainable parameters: {trainable_params/1e6:.2f}M ({100 * trainable_params /
total_params:.2f}%)")
print(f"Total training steps: {total_steps}")
print(f"Using expert initialization method: {config['some_layer']['init_method']}")

# Fix #5: eval temperature is explicit (defaults to training temperature)
eval_temp = float(config['training'].get('eval_temp', config['training']['training_temp']))

# 4. Training Loop
train_losses, val_losses = [], []
best_val_loss = float('inf')
model_save_path = f"best_model_{config['run_name']}_fixed.pth"

for epoch in range(config['training']['num_epochs']):
    print(f"\n--- Epoch {epoch + 1}/{config['training']['num_epochs']} ---")

    train_loss = train_epoch(
        model, train_loader, optimizer, criterion, scheduler,
        config['training']['training_temp'],
        vocab_size=config['model']['vocab_size']
    )
    val_loss = evaluate_epoch(model, val_loader, criterion,
vocab_size=config['model']['vocab_size'], eval_temp=eval_temp)
    perplexity = math.exp(val_loss)

    train_losses.append(train_loss)
    val_losses.append(val_loss)

    model_to_inspect = model._orig_mod if hasattr(model, '_orig_mod') else model
    middle = config['model']['num_layers'] // 2
    layer = model_to_inspect.layers[middle].some_layer

# Fix #8: report both inequality views
gini_count = calculate_gini(layer.usage_count)
entropy_count = calculate_entropy(layer.usage_count)

```

```

gini_mass = calculate_gini(layer.usage_mass)
entropy_mass = calculate_entropy(layer.usage_mass)

print(f"Epoch {epoch + 1}: Train Loss = {train_loss:.4f}, Val Loss = {val_loss:.4f}, Val
Perplexity = {perplexity:.2f}")
print(f" Middle Layer (count): Gini = {gini_count:.3f}, Entropy = {entropy_count:.3f}")
print(f" Middle Layer (mass): Gini = {gini_mass:.3f}, Entropy = {entropy_mass:.3f}")

if val_loss < best_val_loss:
    best_val_loss = val_loss
    torch.save(model_to_inspect.state_dict(), model_save_path)
    print(f" Model saved as {model_save_path}")

# 5. Finalization
print(f"\n--- Training Complete for {config['run_name']} ---")
plot_losses(train_losses, val_losses, config)

```

---

Cell Three:

```

#
=====
=====
# Cell 3: Experiment Configuration & Execution (FIXED)
#
=====
=====

# --- Router selection (toggle) ---
ROUTER_TYPE = "mlp" # "linear" or "mlp"
ROUTER_MLP_MULT = 2.0 # only used for MLP: hidden = int(d_model *
ROUTER_MLP_MULT)
ROUTER_DROPOUT = 0.0 # optional dropout inside the router MLP

config = {
    "run_name": "v3_baseline_orthogonal",

    "data": {
        "train_subset_size": 10000,
        "val_subset_size": 1000,
        "batch_size": 24,
    },

    "model": {
        # NOTE: vocab_size will be overwritten to tokenizer.vocab_size in prepare_data() (Fix #3)
        "vocab_size": 8192,

```



```

    "d_model": 512,
    "num_heads": 8,
    "num_layers": 10,
    "seq_len": 1024,
},

"some_layer": {
    "num_experts": 128,
    "d_ffn": 2048,
    "top_k": 8,
    "init_method": "sparse",

    "alpha": 0.005,
    "beta": 0.001,
    "delta": 0.001,

    "theta_percentile": 0.05,
    "warmup_steps": 400,
    "ema_decay": 0.995,
    "router_type": ROUTER_TYPE,
    "router_mlp_mult": ROUTER_MLP_MULT,
    "router_dropout": ROUTER_DROPOUT,

    "ablation_flags": {
        "use_alpha": True,
        "use_beta": True,
        "use_delta": True
    }
},

"training": {
    "num_epochs": 4,
    "learning_rate": 6e-4,
    "training_temp": 1.0,
    # Fix #5: explicit eval temperature (defaults to training_temp if omitted)
    "eval_temp": 1.0,
}
}

```

main(config)

---

Cell Four:

```

#
=====
=====
# Cell 4: Multi-Layer SoME Analysis & Diagnostics Dashboard v1.3 (FIXED)
#
# RUN THIS CELL AFTER A TRAINING RUN (CELL 3) IS COMPLETE
#
=====
=====

print("\n--- Part 1: Dashboard Setup ---")
!pip install umap-learn seaborn -q
import umap
import seaborn as sns

model_path_to_load = f"best_model_{config['run_name']}_fixed.pth"
tokenizer_path = "tinystories-tokenizer-v2.json"

if os.path.exists(model_path_to_load) and os.path.exists(tokenizer_path):
    print(f"Loading best model from: {model_path_to_load}")
    print(f"Loading tokenizer from: {tokenizer_path}")

    tokenizer = PreTrainedTokenizerFast(tokenizer_file=tokenizer_path)
    tokenizer.add_special_tokens({'pad_token': '[PAD]', 'eos_token': '[EOS]'})

    # Fix #3: keep analysis model vocab consistent with tokenizer
    config['model']['vocab_size'] = tokenizer.vocab_size

    analysis_model = SOMETransformer(config['model'], config['some_layer']).to(device)
    analysis_model.load_state_dict(torch.load(model_path_to_load, map_location=device))
    analysis_model.eval()
else:
    print("ERROR: Could not find a saved model or tokenizer. Please run a training cell first.")
    analysis_model = None
    tokenizer = None

if analysis_model and tokenizer:
    print("\n\n--- Part 2: Aggregate Utilization Analysis (from Middle Layer) ---")

    middle_layer_idx = config['model']['num_layers'] // 2
    middle_layer = analysis_model.layers[middle_layer_idx].some_layer

    usage_count = middle_layer.usage_count.detach().cpu()
    usage_mass = middle_layer.usage_mass.detach().cpu()

```

```

total_experts = middle_layer.num_experts

used_experts = torch.sum(usage_count > 0).item()
usage_percentage = (used_experts / total_experts) * 100
print(f'Expert Usage (Layer {middle_layer_idx}): {used_experts}/{total_experts}
({usage_percentage:.2f}%)')

gini_count = calculate_gini(usage_count)
ent_count = calculate_entropy(usage_count)
gini_mass = calculate_gini(usage_mass)
ent_mass = calculate_entropy(usage_mass)

print(f'Final Metrics (count) Layer {middle_layer_idx}: Gini={gini_count:.4f},
Entropy={ent_count:.4f} (Max={math.log2(total_experts):.4f})')
print(f'Final Metrics (mass) Layer {middle_layer_idx}: Gini={gini_mass:.4f},
Entropy={ent_mass:.4f} (Max={math.log2(total_experts):.4f})')

# Histogram (count)
plt.figure(figsize=(12, 5))
plt.hist(usage_count.numpy(), bins=50)
plt.yscale('log')
plt.title(f'Log-Scale Histogram of Expert Usage (COUNT) ({config['run_name']} - Layer
{middle_layer_idx})')
plt.xlabel('EMA selection mass')
plt.ylabel('Number of Experts (Log Scale)')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()

# Histogram (mass)
plt.figure(figsize=(12, 5))
plt.hist(usage_mass.numpy(), bins=50)
plt.yscale('log')
plt.title(f'Log-Scale Histogram of Expert Usage (MASS) ({config['run_name']} - Layer
{middle_layer_idx})')
plt.xlabel('EMA gating-weight mass')
plt.ylabel('Number of Experts (Log Scale)')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
plt.show()

# --- 3. Key Store Structure Visualization (middle layer) ---
print("\n\n--- Part 3: Key Store Structure Visualization (from Middle Layer) ---")
key_store_data = middle_layer.key_store.detach().cpu().numpy()
print("Running UMAP projection on the key store... (this may take a moment)")

```

```

reducer = umap.UMAP(n_neighbors=15, min_dist=0.1, n_components=2, metric='cosine',
random_state=42)
embedding = reducer.fit_transform(key_store_data)

```

```

plt.figure(figsize=(12, 10))
scatter = plt.scatter(embedding[:, 0], embedding[:, 1], c=usage_mass, s=20, alpha=0.7)
plt.title(f"UMAP Projection of Expert Key Store ({config['run_name']} - Layer
{middle_layer_idx})")
plt.xlabel('UMAP Dimension 1')
plt.ylabel('UMAP Dimension 2')
plt.colorbar(scatter, label='Expert Usage (EMA gating-weight mass)')
plt.grid(True, linestyle='--', linewidth=0.5)
plt.show()

```

# --- 4. Multi-Layer Generative Analysis with Expert Tracing ---

```
print("\n\n--- Part 4: Multi-Layer Generative Analysis with Expert Tracing ---")
```

```

def generate_with_multi_layer_trace(model, tokenizer, prompt, layer_indices,
max_new_tokens=50, temperature=1.0):

```

```

    model.eval()
    input_ids = tokenizer.encode(prompt, return_tensors="pt").to(device)
    generated_ids = input_ids.clone()

```

```
    expert_trace = {layer: [] for layer in layer_indices}
```

```
    print(f"\n--- Prompt ---\n{prompt}", end="")
```

```

    for _ in range(max_new_tokens):
        with torch.no_grad(), torch.amp.autocast("cuda"):
            outputs, _, all_indices, _ = model(generated_ids, attention_mask=None,
temperature=temperature)

```

```

        for layer_idx in layer_indices:
            last_token_indices = all_indices[layer_idx][-1, :].detach().cpu().numpy().tolist()
            expert_trace[layer_idx].append(last_token_indices)

```

```

        next_token_logits = outputs[:, -1, :]
        next_token_id = torch.argmax(next_token_logits, dim=-1).unsqueeze(0)

```

```

        if next_token_id.item() == tokenizer.eos_token_id:
            break

```

```

        generated_ids = torch.cat((generated_ids, next_token_id), dim=1)
        new_token = tokenizer.decode(next_token_id[0])

```

```

print(new_token, end="", flush=True)

print("\n--- End of Generation ---")

print("\n--- Multi-Layer Expert Activation Trace ---")
num_generated_tokens = len(expert_trace[layer_indices[0]])
for i in range(num_generated_tokens):
    token_text = tokenizer.decode(generated_ids[0, input_ids.shape[1] + i]).replace(' ', '_')
    print(f"\nToken '{token_text}':")
    for layer_idx in layer_indices:
        print(f"  Layer {layer_idx}: \tUsed Experts -> {expert_trace[layer_idx][i]}")

layers_to_trace = [1, 5, 9]
prompts = [
    "Once upon a time, there was a little fox who",
    "The recipe for the perfect cake is to first",
    "The robot opened its eyes and saw",
]
for p in prompts:
    generate_with_multi_layer_trace(analysis_model, tokenizer, p,
    layer_indices=layers_to_trace, temperature=1.0)

```