SoME v8 Test Grafting

```
# Cell 1: Setup and Dependencies
print("--- Part 1: Setup and Dependencies ---")
!pip install -q transformers datasets tokenizers accelerate

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
from transformers import AutoModelForCausalLM, AutoConfig
from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import Whitespace
from transformers import PreTrainedTokenizerFast
from datasets import load_dataset
import copy
from tqdm import tqdm
import math
import os
import numpy as np
import gc

# Verify GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Enable TF32/BF16 optimizations for A100
if torch.cuda.is_available() and torch.cuda.get_device_capability(0)[0] >= 8:
    print("Ampere+ GPU detected. Enabling TF32.")
    torch.set_float32_matmul_precision('high')
    torch.backends.cudnn.benchmark = True

# Cell 2: The Harvester (Neuro-Grafting Logic)

def harvest_experts_from_model(model_name="Qwen/Qwen2.5-0.5B", slice_size=128):
    """
    Downloads a model, slices its SwiGLU FFNs into chunks, and returns a library of frozen
weights.
    """
    print(f"\n🔪 Harvesting organs from {model_name}...")

    # Load model on CPU to save GPU RAM
    hf_model = AutoModelForCausalLM.from_pretrained(
        model_name,
        torch_dtype=torch.float16,
        device_map="cpu",
        trust_remote_code=True
    )
    hf_config = hf_model.config

    # Qwen uses d_model=896. We must match this.
    print(f"Model Architecture: d_model={hf_config.hidden_size},
layers={hf_config.num_hidden_layers}")

    expert_library = []

    # Iterate through layers
    layers = hf_model.model.layers

    for i, layer in enumerate(tqdm(layers, desc="Slicing Layers")):
        # Extract MLP weights
        # Qwen/Llama use: gate_proj, up_proj, down_proj
        mlp = layer.mlp

        # Get raw weights
        # gate/up shape: [intermediate_size, d_model]
        # down shape:    [d_model, intermediate_size]
        w_gate = mlp.gate_proj.weight.data
        w_up = mlp.up_proj.weight.data
        w_down = mlp.down_proj.weight.data

        # Calculate how many experts we can cut from this layer
        intermediate_size = w_gate.shape[0]
        num_slices = intermediate_size // slice_size

        for j in range(num_slices):
            start_idx = j * slice_size
            end_idx = start_idx + slice_size

            # Slice the SwiGLU components
            # Gate/Up: Slice output dim (rows)
            slice_gate = w_gate[start_idx:end_idx, :].clone()
            slice_up = w_up[start_idx:end_idx, :].clone()

            # Down: Slice input dim (columns)
            slice_down = w_down[:, start_idx:end_idx].clone()
            # We store them as FP16 to save RAM, will cast to FP32 during init if needed
            expert_library.append({
                'w_gate': slice_gate,
                'w_up': slice_up,
                'w_down': slice_down,
                'source_layer': i
            })

    print(f"✅ Harvest Complete. Extracted {len(expert_library)} experts of hidden_dim
{slice_size}.")

    # Clean up to free RAM
    del hf_model
    gc.collect()

    return expert_library, hf_config.hidden_size

# Cell 3: Data Preparation

class LanguageModelDataset(Dataset):
    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)

        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()

        targets = input_ids.clone()
        targets[:-1] = input_ids[1:]
        targets[-1] = -100

        if self.pad_token_id is not None:
            targets[targets == self.pad_token_id] = -100
        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"

    # 1. Tokenizer Logic
    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)
        tokenizer = PreTrainedTokenizerFast(tokenizer_file=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]'})
    config['model']['vocab_size'] = tokenizer.vocab_size
    print(f"Custom tokenizer loaded with vocab size: {tokenizer.vocab_size}")

    # 2. Dataset Tokenization
    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)

    train_loader = DataLoader(train_dataset, batch_size=config['data']['batch_size'],
                              shuffle=True, drop_last=True, num_workers=2, pin_memory=True)
    validation_loader = DataLoader(validation_dataset, batch_size=config['data']['batch_size'],
                                 drop_last=False, num_workers=2, pin_memory=True)

    return train_loader, validation_loader, tokenizer

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))

# Cell 4: SoME v8 Architecture (Grafted)

class GraftedExpert(nn.Module):
    """
    An expert that initializes from pre-trained SwiGLU weights (Gate, Up, Down).
    Frozen by default.
    """
    def __init__(self, d_model, d_ffn, weights):
        super().__init__()

        # SwiGLU: Gate (d_model->d_ffn), Up (d_model->d_ffn), Down (d_ffn->d_model)
        self.w_gate = nn.Linear(d_model, d_ffn, bias=False)
        self.w_up = nn.Linear(d_model, d_ffn, bias=False)
        self.w_down = nn.Linear(d_ffn, d_model, bias=False)

        # Load weights
        # Note: nn.Linear stores as (Out, In).
        # Our Harvester provided slice_gate as (Out, In), slice_up as (Out, In).
        # Our Harvester provided slice_down as (Out, In).
        self.w_gate.weight.data = weights['w_gate'].to(torch.float32)
        self.w_up.weight.data = weights['w_up'].to(torch.float32)
        self.w_down.weight.data = weights['w_down'].to(torch.float32)

        # Freeze immediately
        for param in self.parameters():
            param.requires_grad = False

    def forward(self, x):
        # SwiGLU: (Swish(Gate(x)) * Up(x)) @ Down
        return self.w_down(F.silu(self.w_gate(x)) * self.w_up(x))

class SOMELayer(nn.Module):
    def __init__(self, d_model, some_config, expert_library_subset):
        super().__init__()
        self.d_model = d_model
        self.num_experts = len(expert_library_subset)
        self.top_k = some_config['top_k']

        # Heuristics
        self.alpha = some_config['alpha']
        self.beta = some_config['beta']
        self.delta = some_config['delta']
        self.respawn_threshold = some_config.get('respawn_threshold', 0.1)
        self.theta_percentile = some_config['theta_percentile']
        self.warmup_steps = some_config['warmup_steps']
        self.ema_decay = some_config['ema_decay']
        self.ablation_flags = some_config.get('ablation_flags', {'use_alpha': True, 'use_beta': True,
'use_delta': True})

        # Router (Trainable) - MLP Router
        mult = float(some_config.get("router_mlp_mult", 2.0))
        hidden = int(d_model * mult)
        self.query_network = nn.Sequential(
            nn.Linear(d_model, hidden),
            nn.GELU(),
            nn.Linear(hidden, d_model)
        )

        # Initialize Query Net
        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)

        # Key Store (Trainable via Heuristics)
        # Init keys on sphere
        keys = torch.randn(self.num_experts, d_model)
        self.register_buffer("key_store", F.normalize(keys, p=2, dim=-1))

        # Usage Stats
        self.register_buffer("usage_count", torch.zeros(self.num_experts))
        self.register_buffer("usage_mass", torch.zeros(self.num_experts))
        self.register_buffer("steps", torch.zeros((), dtype=torch.long))
        self.register_buffer("dead_expert_count", torch.zeros((), dtype=torch.long))

        # The Experts (Frozen & Grafted)
        self.experts = nn.ModuleList()
        for weights in expert_library_subset:
            # d_ffn is determined by the slice size inside the weights (rows of gate)
            d_ffn = weights['w_gate'].shape[0]
            self.experts.append(GraftedExpert(d_model, d_ffn, weights))

        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)

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

            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)

        # Dispatch
        flat_indices = top_k_indices.view(-1)
        sorted_indices, perm_map = torch.sort(flat_indices)
        unique_ids, counts = torch.unique_consecutive(sorted_indices, return_counts=True)

        flat_x = x_flat.repeat_interleave(self.top_k, dim=0)
        perm_x = flat_x[perm_map]
        split_x = torch.split(perm_x, counts.tolist())

        outputs = []
        for i, expert_id in enumerate(unique_ids):
            outputs.append(self.experts[expert_id](split_x[i]))

        cat_out = torch.cat(outputs, dim=0)

        # Reorder
        inv_perm = torch.argsort(perm_map)
        ordered_out = cat_out[inv_perm]

        # Weight
        gating_weights_f = gating_weights.to(ordered_out.dtype)
        weighted = (ordered_out.view(-1, self.top_k, self.d_model) *
gating_weights_f.unsqueeze(-1)).sum(1)

        final_output = weighted.view(batch_size, seq_len, self.d_model)

        # Return x + residual (Pre-Norm style handled in block)
        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
        num_tokens = top_k_indices.shape[0]
        if num_tokens == 0: return

        # 1. Update Usage Stats
        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)

        inertia_source = self.usage_count

        # 2. Calculate Updates (Attraction + Peer Pull)
        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)

        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)
            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)

        # 3. The Clamp Fix
        current_norms = self.key_store.norm(p=2, dim=-1, keepdim=True)
        clamp_mask = (current_norms > 1.0)
        if clamp_mask.any():
            self.key_store = torch.where(clamp_mask, self.key_store / current_norms,
self.key_store)

        # 4. Decay
        if self.steps > 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)

        # 5. Phoenix Mechanism
        key_norms = self.key_store.norm(p=2, dim=-1)
        dead_mask = key_norms < self.respawn_threshold
        num_dead = dead_mask.sum().item()
        self.dead_expert_count.fill_(num_dead)

        if num_dead > 0:
            flat_queries = queries.view(-1, self.d_model)
            if flat_queries.size(0) >= num_dead:
```

```
            rand_indices = torch.randperm(flat_queries.size(0),
device=queries.device)[:num_dead]
            new_keys = flat_queries[rand_indices].clone()
            new_keys = F.normalize(new_keys + torch.randn_like(new_keys) * 0.01, p=2, dim=-1)
            self.key_store[dead_mask] = new_keys
            self.usage_count[dead_mask] = 0.0
            self.usage_mass[dead_mask] = 0.0

class SOMETransformer(nn.Module):
  def __init__(self, model_config, some_config, expert_library):
    super().__init__()
    self.d_model = model_config['d_model']
    self.num_layers = model_config['num_layers']

    # New embeddings trained from scratch (mapped to Qwen semantic space by router)
    self.embedding = nn.Embedding(model_config['vocab_size'], self.d_model)
    self.pos_encoder = nn.Embedding(model_config['seq_len'], self.d_model)

    self.layers = nn.ModuleList()

    # Distribute experts across layers
    total_experts = len(expert_library)
    experts_per_layer = total_experts // self.num_layers

    # Shuffle library to ensure randomness in distribution
    import random
    random.shuffle(expert_library)

    for i in range(self.num_layers):
      start = i * experts_per_layer
      end = start + experts_per_layer
      subset = expert_library[start:end]

      # Construct Block
      block = nn.ModuleDict({
        'attn': nn.MultiheadAttention(self.d_model, model_config['num_heads'],
batch_first=True),
        'norm1': nn.LayerNorm(self.d_model),
        'norm2': nn.LayerNorm(self.d_model),
        'some': SOMELayer(self.d_model, some_config, subset)
      })
      self.layers.append(block)

    self.fc_out = nn.Linear(self.d_model, model_config['vocab_size'])
```

```
        self.norm_f = nn.LayerNorm(self.d_model)

    def forward(self, x, attention_mask=None, temperature=1.0):
        b, seq = x.shape
        positions = torch.arange(seq, device=x.device).unsqueeze(0)
        x = self.embedding(x) + self.pos_encoder(positions)

        all_queries, all_indices, all_gates = [], [], []
        causal_mask = torch.triu(torch.ones(seq, seq, device=x.device), diagonal=1).bool()

        if attention_mask is not None:
            key_padding_mask = (attention_mask == 0)
        else:
            key_padding_mask = None

        for layer in self.layers:
            # Attention Block
            norm_x = layer['norm1'](x)
            attn_out, _ = layer['attn'](norm_x, norm_x, norm_x,
                                        attn_mask=causal_mask,
                                        key_padding_mask=key_padding_mask,
                                        is_causal=True)
            x = x + attn_out

            # SoME Block
            norm_x = layer['norm2'](x)
            # SOMELayer returns (x + residual), but since we passed norm_x, it returns (norm_x +
residual)
            # We want x + residual.
            # some_out_raw = layer['some'](norm_x) -> returns norm_x + expert_out
            some_out, q, idx, g = layer['some'](norm_x, temperature=temperature)

            # Extract pure residual: (norm_x + expert_out) - norm_x = expert_out
            expert_residual = some_out - norm_x

            # Add to main stream
            x = x + expert_residual

            all_queries.append(q)
            all_indices.append(idx)
            all_gates.append(g)

        return self.fc_out(self.norm_f(x)), all_queries, all_indices, all_gates
    @torch.no_grad()
    def update_all_keys(self, all_queries, all_indices, all_gates, token_mask=None):
        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'].update_keys(q, idx, g)

# Cell 5: Main Execution

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()

        # Heuristic Updates
        model_state = model._orig_mod if hasattr(model, "_orig_mod") else model
        valid_mask = (targets.view(-1) != -100)
        model_state.update_all_keys(queries, indices, gates, token_mask=valid_mask)

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

    return total_loss / len(dataloader)

def evaluate_epoch(model, dataloader, criterion, vocab_size, eval_temp):
    model.eval()
    total_loss = 0
    with torch.no_grad():
        for input_ids, targets, attention_mask in tqdm(dataloader, desc="Eval", leave=False):
            input_ids = input_ids.to(device)
            targets = targets.to(device)
            attention_mask = attention_mask.to(device)

            with torch.cuda.amp.autocast():
                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()
    return total_loss / len(dataloader)

def main():
    # 1. Harvest Experts
    # We slice to 128 dim. Qwen 0.5B intermediate is 4864.
    # 4864 / 128 = 38 experts per layer.
    # 24 layers * 38 = 912 total experts.
    slice_size = 128
    expert_library, qwen_d_model = harvest_experts_from_model("Qwen/Qwen2.5-0.5B",
slice_size=slice_size)

    # 2. Config
    config = {
        "run_name": "v8_LlamaGraft_Qwen0.5B",
        "data": { "train_subset_size": 20000, "val_subset_size": 2000, "batch_size": 16 },
        "model": {
            "vocab_size": 8192,
            "d_model": qwen_d_model, # Must match Qwen (896)
            "num_heads": 14,         # 896 / 64 = 14 heads
            "num_layers": 6,         # Stacking 6 layers of SoME
            "seq_len": 512
        },
        "some_layer": {
            "top_k": 4,
            "alpha": 0.015,
            "beta": 0.001,
            "delta": 0.005,
            "respawn_threshold": 0.1,
            "theta_percentile": 0.05,
            "warmup_steps": 200,
            "ema_decay": 0.995,
            "router_mlp_mult": 2.0,
            "ablation_flags": {"use_alpha": True, "use_beta": True, "use_delta": True}
        },
        "training": { "num_epochs": 2, "learning_rate": 3e-4, "training_temp": 1.0, "eval_temp": 1.0 }
    }

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

    # 4. Init Model
    print("Initializing Grafted SoME...")
    model = SOMETransformer(config['model'], config['some_layer'], expert_library).to(device)

    # Stats
    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
({trainable_params/total_params*100:.2f}%)")

    # 5. Train
    optimizer = torch.optim.AdamW([p for p in model.parameters() if p.requires_grad],
lr=config['training']['learning_rate'])
    criterion = nn.CrossEntropyLoss(ignore_index=-100)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer,
T_max=len(train_loader)*config['training']['num_epochs'])

    print("\n--- Starting Training ---")
    for epoch in range(config['training']['num_epochs']):
        train_loss = train_epoch(model, train_loader, optimizer, criterion, scheduler, 1.0,
config['model']['vocab_size'])
        val_loss = evaluate_epoch(model, val_loader, criterion, config['model']['vocab_size'], 1.0)

        # Log Gini for middle layer
        mid_layer = model.layers[config['model']['num_layers'] // 2]['some']
        gini = calculate_gini(mid_layer.usage_count)

        print(f"Epoch {epoch+1}: Train={train_loss:.4f}, Val={val_loss:.4f},
PPL={math.exp(val_loss):.2f}")
        print(f" Middle Layer Gini={gini:.3f}, Dead Experts={mid_layer.dead_expert_count.item()}")

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