

SoME v3 Colab Code Notebook

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
# (Classes, Data Functions, Training Loop)
#
=====
=====
```

--- 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']

# --- NEW: 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}")

self.query_network = nn.Linear(d_model, d_model)

keys = torch.randn(self.num_experts, d_model)
self.register_buffer("key_store", F.normalize(keys, p=2, dim=-1))
self.register_buffer("usage_count", torch.zeros(self.num_experts))
self.register_buffer("steps", torch.tensor([0], 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)

    queries_raw = self.query_network(x_flat)
    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 / temperature, dim=-1)

    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]

weighted_outputs = (expert_outputs.view(-1, self.top_k, self.d_model) *
gating_weights.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

```

```

@torch.no_grad()
def update_keys(self, queries, top_k_indices):
    self.steps += 1
    unique_indices, counts = torch.unique(top_k_indices, return_counts=True)
    self.usage_count.mul_(self.ema_decay)
    self.usage_count.index_add_(0, unique_indices, (1.0 - self.ema_decay) * counts.float())

    # --- 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 + self.usage_count[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 + self.usage_count[indices_i]).unsqueeze(-1)

```

```

inertia_j = (1.0 + self.usage_count[indices_j]).unsqueeze(-1)
beta_effective = self.beta / torch.min(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)

```

```

self.key_store.data = F.normalize(self.key_store.data, p=2, dim=-1)

```

```

# --- Heuristic 3: Delta (Decay / "Use it or Lose it") ---
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)

```

```

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 = SOME_LAYER(d_model, some_config)

```

```

    def forward(self, x, temperature=1.0):
        seq_len = x.size(1)
        mask = torch.triu(torch.ones(seq_len, seq_len, device=x.device) * float('-inf'), diagonal=1)
        attn_output, _ = self.attention(x, x, x, attn_mask=mask)
        x = self.norm1(x + attn_output)
        some_output, queries, top_k_indices = self.some_layer(x, temperature=temperature)
        x = self.norm2(some_output)
        return x, queries, top_k_indices

```

```

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, temperature=1.0):
    x = self.embedding(x) * math.sqrt(self.d_model)
    x = self.pos_encoder(x)
    all_queries, all_indices = [], []
    for layer in self.layers:
        x, queries, top_k_indices = layer(x, temperature=temperature)
        all_queries.append(queries)
        all_indices.append(top_k_indices)
    return self.fc_out(x), all_queries, all_indices
```

```
@torch.no_grad()
def update_all_keys(self, all_queries, all_indices):
    for i, layer_block in enumerate(self.layers):
        queries = all_queries[i].view(-1, layer_block.some_layer.d_model)
        indices = all_indices[i].view(-1, layer_block.some_layer.top_k)
        layer_block.some_layer.update_keys(queries, indices)
```

--- 2. Data Preparation ---

```
class LanguageModelDataset(Dataset):
    def __init__(self, tokenized_data):
        self.data = tokenized_data
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        item = self.data[idx]
        inputs = torch.tensor(item['input_ids'])
```

```

    targets = inputs.clone()
    targets[:-1] = inputs[1:]
    targets[-1] = -100 # Ignore loss for the last token prediction
    return inputs, targets

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}")

    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)
    validation_dataset = LanguageModelDataset(tokenized_val)

```

```

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=True, 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,
tokenizer_vocab_size):
    model.train()
    total_loss = 0
    scaler = torch.cuda.amp.GradScaler()
    progress_bar = tqdm(dataloader, desc="Training", leave=False)

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

        with torch.cuda.amp.autocast():
            logits, queries, indices = model(inputs, temperature=current_temp)
            loss = criterion(logits.view(-1, tokenizer_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()

model.update_all_keys(queries, indices)
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, tokenizer_vocab_size):
    model.eval()
    total_loss = 0
    progress_bar = tqdm(dataloader, desc="Evaluating", leave=False)

    with torch.no_grad():
        for inputs, targets in progress_bar:
            inputs, targets = inputs.to(device, non_blocking=True), targets.to(device,
non_blocking=True)
            with torch.cuda.amp.autocast():
                logits, _, _ = model(inputs, temperature=0.5) # Sharpen during eval
                loss = criterion(logits.view(-1, tokenizer_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 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']}.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']}")

    # 4. Training Loop
    train_losses, val_losses = [], []
    best_val_loss = float('inf')
    model_save_path = f"best_model_{config['run_name']}.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'], tokenizer.vocab_size)
    val_loss = evaluate_epoch(model, val_loader, criterion, tokenizer.vocab_size)
    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
    usage_counts = model_to_inspect.layers[config['model']]['num_layers'] //
2].some_layer.usage_count
    gini_coeff = calculate_gini(usage_counts)
    entropy_val = calculate_entropy(usage_counts)

    print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Val Loss = {val_loss:.4f}, Val
Perplexity = {perplexity:.2f}")
    print(f" Middle Layer Expert Metrics: Gini = {gini_coeff:.3f}, Entropy = {entropy_val:.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

#

=====

=====

--- Define the configuration for the experiment ---

config = {

Unique name for this run, used for saving files

"run_name": "v3_baseline_orthogonal",

--- Data Settings ---

"data": {

```

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

# --- Model Dimensions ---
"model": {
    "vocab_size": 8192,
    "d_model": 768,
    "num_heads": 12,
    "num_layers": 12,
    "seq_len": 1536,
},

# --- SoME Layer Hyperparameters ---
"some_layer": {
    "num_experts": 256,
    "d_ffn": 1536,
    "top_k": 8,
    "init_method": "sparse", # Options: 'default', 'orthogonal', 'sparse'

    # Heuristic update rules
    "alpha": 0.015, # Attraction
    "beta": 0.001, # Peer Pull / Clustering
    "delta": 0.001, # Decay

    # Key management
    "theta_percentile": 0.05,
    "warmup_steps": 400,
    "ema_decay": 0.995,

    # ===== #
    # === ABLATION STUDY CONTROL PANEL ===
    # Set these to True/False to enable/disable heuristics
    "ablation_flags": {
        "use_alpha": True, # Master switch for the attraction rule
        "use_beta": True, # Master switch for the peer pull rule
        "use_delta": True # Master switch for the decay rule
    }
    # ===== #
},

# --- Training Schedule ---
"training": {

```

```

        "num_epochs": 4,
        "learning_rate": 6e-4,
        "training_temp": 1.0,
    }
}

# --- Run the experiment ---
main(config)



---


Cell Four
#
=====

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

=====

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

# Ensure the latest configuration is loaded
model_path_to_load = f"best_model_{config['run_name']}.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]'})

    analysis_model = SOMETransformer(config['model'], config['some_layer']).to(device)
    analysis_model.load_state_dict(torch.load(model_path_to_load))
    analysis_model.eval()

# We will inspect multiple layers, so no single 'layer_to_inspect' is needed here.

```

```

else:
    print(f"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:
    # --- 2. Aggregate Utilization Analysis (from a representative middle layer) ---
    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_counts = middle_layer.usage_count.cpu()
    total_experts = middle_layer.num_experts

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

    final_gini = calculate_gini(usage_counts)
    final_entropy = calculate_entropy(usage_counts)
    print(f"Final Gini Coefficient (Layer {middle_layer_idx}): {final_gini:.4f}")
    print(f"Final Shannon Entropy (Layer {middle_layer_idx}): {final_entropy:.4f} (Max: {math.log2(total_experts):.4f})")

    plt.figure(figsize=(12, 6))
    plt.hist(usage_counts.numpy(), bins=50, color='skyblue', edgecolor='black')
    plt.yscale('log')
    plt.title(f"Log-Scale Histogram of Expert Usage Counts ({config['run_name']} - Layer {middle_layer_idx})")
    plt.xlabel('Usage Count (EMA)')
    plt.ylabel('Number of Experts (Log Scale)')
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
    plt.show()

    # --- 3. Key Store Structure Visualization (from middle layer) ---
    print("\n\n--- Part 3: Key Store Structure Visualization (from Middle Layer) ---")
    key_store_data = middle_layer.key_store.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_counts, cmap='viridis', 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 Count (EMA)')
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 ---")

```

```

# === MODIFIED FUNCTION: Accepts a list of layer indices ===
def generate_with_multi_layer_trace(model, tokenizer, prompt, layer_indices,
max_new_tokens=50):
    model.eval()
    input_ids = tokenizer.encode(prompt, return_tensors="pt").to(device)
    generated_ids = input_ids.clone()

    # New data structure to hold traces for multiple layers
    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_queries, all_indices = model(generated_ids)

            # For each layer we are watching, capture the expert indices for the last token
            for layer_idx in layer_indices:
                last_token_indices = all_indices[layer_idx][-1, :].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(f"\n--- Multi-Layer Expert Activation Trace ---")
num_generated_tokens = len(expert_trace[layer_indices[0]])

# New print loop to group by token for easy comparison
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:
        token_experts = expert_trace[layer_idx][i]
        print(f" Layer {layer_idx}: \tUsed Experts -> {token_experts}")

# === MODIFIED CALL: Specify the layers to trace ===
layers_to_trace = [1, 5, 9] # Early, Middle, and Late layers
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)

```
