

```

#
=====
=====
# SoME Transformer V5.2 - Final V2 Baseline with All Phase 1 Fixes
# Implements:
# 1. DEFINITIVE CAUSAL MASKING: Explicitly creates and registers a causal mask.
# 2. CORRECT TARGET CREATION: Standard next-token prediction target shifting. (V2.2
FIX)
# 3. CORRECT LR SCHEDULER ORDER: The scheduler is now called after the optimizer.
(V2.2 FIX)
# 4. All other V2 Baseline improvements (EOS handling, Cosine Routing, EMA Inertia).
#
=====
=====

# 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')

```

```
torch.backends.cudnn.benchmark = True
```

```
# Part 2: Data Preparation (Fixed)
```

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

```
SEQ_LEN = 512
```

```
BATCH_SIZE = 64
```

```
VOCAB_SIZE = 8192
```

```
print("--- Loading/Training Custom Tokenizer ---")
```

```
tokenizer_path = "tinystories-tokenizer-v2.json"
```

```
if not os.path.exists(tokenizer_path):
```

```
    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 = Tokenizer(BPE(unk_token="[UNK]"))
```

```
    tokenizer.pre_tokenizer = Whitespace()
```

```
    trainer = BpeTrainer(special_tokens=["[UNK]", "[PAD]", "[EOS]"], vocab_size=VOCAB_SIZE)
```

```
    tokenizer.train_from_iterator(get_training_corpus(), trainer=trainer)
```

```
    tokenizer.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("\n--- Tokenizing Dataset ---")
```

```
full_dataset = load_dataset("roneneldan/TinyStories")
```

```
train_subset_size = 40000
```

```
val_subset_size = 10000
```

```
train_subset = full_dataset["train"].select(range(train_subset_size))
```

```
val_subset = full_dataset["validation"].select(range(val_subset_size))
```

```
## V2 FIX: EOS TOKEN HANDLING
```

```
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=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())

```

```

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'])

```

```

        ## V2.2 FIX: Correct Target Creation

```

```

        # The target for token at position `i` should be the token at position `i+1`.

```

```

        targets = inputs.clone()

```

```

        targets[:-1] = inputs[1:]

```

```

        targets[-1] = -100 # Ignore the loss for the last token prediction.

```

```

        return inputs, targets

```

```

train_dataset = LanguageModelDataset(tokenized_train)

```

```

validation_dataset = LanguageModelDataset(tokenized_val)

```

```

CPU_COUNT = os.cpu_count()

```

```

NUM_WORKERS = max(2, CPU_COUNT // 2 if CPU_COUNT else 2)

```

```

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True,

```

```

drop_last=True, num_workers=NUM_WORKERS, pin_memory=True)

```

```

validation_loader = DataLoader(validation_dataset, batch_size=BATCH_SIZE, drop_last=True,

```

```

num_workers=NUM_WORKERS, pin_memory=True)

```

```

print(f"Train dataset size (subset): {len(train_dataset)}")

```

```

print(f"Using {NUM_WORKERS} workers for DataLoader.")

```

```

# Part 3: Model Definition (Optimized & Fixed)

```

```

# =====

```

```

class Expert(nn.Module):

```

```

def __init__(self, d_model, d_ffn):
    super().__init__()
    self.w_down = nn.Linear(d_model, d_ffn)
    self.activation = nn.GELU()
    self.w_up = nn.Linear(d_ffn, d_model)
def forward(self, x):
    return self.w_up(self.activation(self.w_down(x)))

class SOMELayer(nn.Module):
    def __init__(self, d_model, num_experts, d_ffn, top_k, alpha=0.01, beta=0.001, delta=0.001,
theta_percentile=0.05, warmup_steps=1000, ema_decay=0.99):
        super().__init__()
        self.d_model, self.num_experts, self.d_ffn, self.top_k = d_model, num_experts, d_ffn, top_k
        self.alpha, self.beta, self.delta = alpha, beta, delta
        self.theta_percentile = theta_percentile
        self.warmup_steps = warmup_steps
        self.query_network = nn.Linear(d_model, d_model)
        keys = torch.randn(num_experts, d_model)
        self.register_buffer("key_store", F.normalize(keys, p=2, dim=-1))
        self.register_buffer("usage_count", torch.zeros(num_experts))
        self.register_buffer("steps", torch.tensor([0], dtype=torch.long))
        self.experts = nn.ModuleList([Expert(d_model, d_ffn) for _ in range(num_experts)])
        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))
        self.ema_decay = ema_decay

    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())

    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)

    if self.top_k > 1:
        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 = self.key_store[indices_i]
        keys_j = 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)

if self.steps > self.warmup_steps:
    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_layer):
        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
        ## V2.1 FIX: Create and register a permanent causal mask.
        mask = torch.triu(torch.ones(SEQ_LEN, SEQ_LEN) * float('-inf'), diagonal=1)
        self.register_buffer('mask', mask)

    def forward(self, x, temperature=1.0):
        ## V2.1 FIX: Pass the registered causal mask to the attention call.
        attn_output, _ = self.attention(x, x, x, attn_mask=self.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__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

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

```

```

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

    def forward(self, x, temperature=1.0):
        x = self.embedding(x) * math.sqrt(self.embedding.embedding_dim)
        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)

```

Part 4: Training, Evaluation, and Metrics

```

# =====
scaler = torch.amp.GradScaler("cuda")

```

```

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):
    model.train()
    total_loss = 0
    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.amp.autocast("cuda"):
            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()

            ## V2.2 FIX: Correct LR Scheduler Order
            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):
    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.amp.autocast("cuda"):
                logits, _, _ = model(inputs, temperature=0.5)
                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, epochs):
    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')
    plt.title('Training and Validation Loss (V2 Baseline)')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.xticks(range(1, epochs + 1))
    plt.savefig('loss_curve_v2_baseline.png')
    print("\nLoss curve plot saved to loss_curve_v2_baseline.png")

# Part 5: Main Execution Block (V2 Baseline Config)
# =====
D_MODEL = 512
NUM_HEADS = 8
NUM_LAYERS = 8
some_config = {
    "num_experts": 256,
    "d_ffn": 1536,
    "top_k": 8,
    "alpha": 0.01,
    "beta": 0.001,
    "delta": 0.001,
    "theta_percentile": 0.05,
    "warmup_steps": 2000,
    "ema_decay": 0.99
}

NUM_EPOCHS = 10
LEARNING_RATE = 6e-4
TRAINING_TEMP = 0.8

# --- Initialization ---
model = SOMETransformer(
    vocab_size=tokenizer.vocab_size, d_model=D_MODEL, num_heads=NUM_HEADS,
    num_layers=NUM_LAYERS, some_config=some_config
).to(device)

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

```

```

optimizer = torch.optim.AdamW([p for p in model.parameters() if p.requires_grad],
lr=LEARNING_RATE, betas=(0.9, 0.95), weight_decay=0.1)
criterion = nn.CrossEntropyLoss(ignore_index=-100)
total_steps = len(train_loader) * NUM_EPOCHS
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=total_steps)

print(f"\nTotal parameters: {sum(p.numel() for p in model.parameters())/1e6:.2f}M")
print(f"Trainable parameters: {sum(p.numel() for p in model.parameters() if
p.requires_grad)/1e6:.2f}M")
print(f"Total training steps: {total_steps}")

# --- Run Training and Metric Tracking ---
train_losses, val_losses = [], []
best_val_loss = float('inf')

for epoch in range(NUM_EPOCHS):
    print(f"\n--- Epoch {epoch+1}/{NUM_EPOCHS} ---")
    train_loss = train_epoch(model, train_loader, optimizer, criterion, scheduler,
current_temp=TRAINING_TEMP)
    val_loss = evaluate_epoch(model, validation_loader, criterion)
    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[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(), "best_some_transformer_v2_baseline.pth")
        print("Model saved as best_some_transformer_v2_baseline.pth")

print("\n--- V2 Baseline Training Complete ---")
plot_losses(train_losses, val_losses, NUM_EPOCHS)

```

```
Using device: cuda
A100 GPU detected. Enabling TF32.
--- Loading/Training Custom Tokenizer ---
Tokenizer already exists. Loading from file.
Custom tokenizer loaded with vocab size: 8192
```

```
--- Tokenizing Dataset ---
Train dataset size (subset): 40000
Using 6 workers for DataLoader.
```

Compiling the model for faster training...

Total parameters: 3244.34M
Trainable parameters: 18.92M
Total training steps: 6250

--- Epoch 1/10 ---

Training: 0%| | 0/625 [00:00<?,
?it/s]/usr/local/lib/python3.12/dist-packages/torch/optim/lr_scheduler.py:192: UserWarning:
Detected call of `lr_scheduler.step()` before `optimizer.step()`. In PyTorch 1.1.0 and later, you
should call them in the opposite order: `optimizer.step()` before `lr_scheduler.step()`. Failure to
do this will result in PyTorch skipping the first value of the learning rate schedule. See more
details at <https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate>
warnings.warn(

Epoch 1: Train Loss = 1.4702, Val Loss = 1.0995, Val Perplexity = 3.00
 └ Middle Layer Expert Metrics: Gini = 0.740, Entropy = 5.780
 Model saved as best_some_transformer_v2_baseline.pth

--- Epoch 2/10 ---

Epoch 2: Train Loss = 1.0929, Val Loss = 0.9834, Val Perplexity = 2.67
 └ Middle Layer Expert Metrics: Gini = 0.714, Entropy = 5.841
 Model saved as best_some_transformer_v2_baseline.pth

--- Epoch 3/10 ---

Epoch 3: Train Loss = 0.9882, Val Loss = 0.9299, Val Perplexity = 2.53
 └ Middle Layer Expert Metrics: Gini = 0.706, Entropy = 5.868
 Model saved as best_some_transformer_v2_baseline.pth

--- Epoch 4/10 ---

Epoch 4: Train Loss = 0.9241, Val Loss = 0.8964, Val Perplexity = 2.45
└ Middle Layer Expert Metrics: Gini = 0.703, Entropy = 5.889
Model saved as best_some_transformer_v2_baseline.pth

--- Epoch 5/10 ---

Epoch 5: Train Loss = 0.8757, Val Loss = 0.8763, Val Perplexity = 2.40
└ Middle Layer Expert Metrics: Gini = 0.699, Entropy = 5.880
Model saved as best_some_transformer_v2_baseline.pth

--- Epoch 6/10 ---

Epoch 6: Train Loss = 0.8360, Val Loss = 0.8622, Val Perplexity = 2.37
└ Middle Layer Expert Metrics: Gini = 0.698, Entropy = 5.887
Model saved as best_some_transformer_v2_baseline.pth

--- Epoch 7/10 ---

Epoch 7: Train Loss = 0.8027, Val Loss = 0.8524, Val Perplexity = 2.35
└ Middle Layer Expert Metrics: Gini = 0.697, Entropy = 5.878
Model saved as best_some_transformer_v2_baseline.pth

--- Epoch 8/10 ---

Epoch 8: Train Loss = 0.7762, Val Loss = 0.8482, Val Perplexity = 2.34
└ Middle Layer Expert Metrics: Gini = 0.689, Entropy = 5.907
Model saved as best_some_transformer_v2_baseline.pth

--- Epoch 9/10 ---

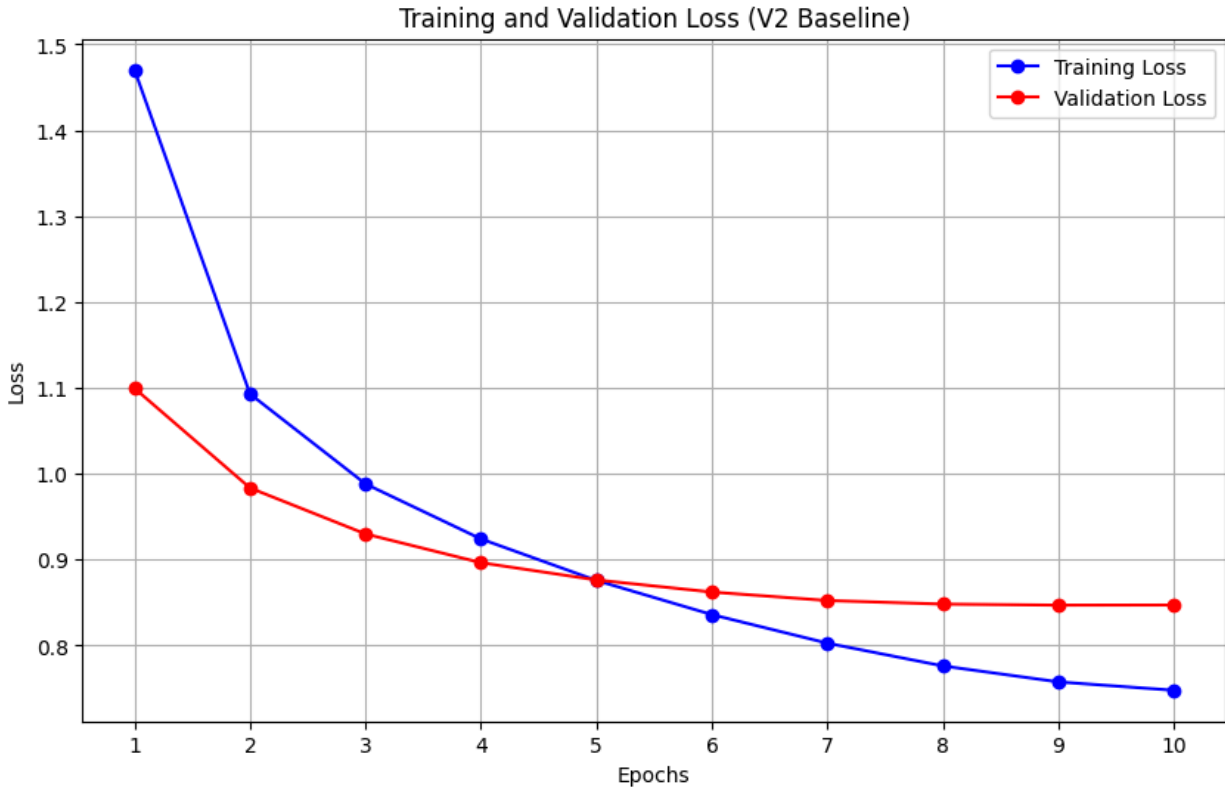
Epoch 9: Train Loss = 0.7576, Val Loss = 0.8470, Val Perplexity = 2.33
└ Middle Layer Expert Metrics: Gini = 0.690, Entropy = 5.898
Model saved as best_some_transformer_v2_baseline.pth

--- Epoch 10/10 ---

Epoch 10: Train Loss = 0.7479, Val Loss = 0.8471, Val Perplexity = 2.33
└ Middle Layer Expert Metrics: Gini = 0.691, Entropy = 5.893

--- V2 Baseline Training Complete ---

Loss curve plot saved to loss_curve_v2_baseline.png



```
#
```

```
=====
```

```
=====
```

```
# Cell 2: Advanced Visualization of Expert "Knowledge Galaxies"
```

```
#
```

```
=====
```

```
=====
```

```
from sklearn.manifold import TSNE
```

```
import matplotlib.pyplot as plt
```

```
import numpy as np
```

```
# Ensure the model class definitions from the previous cell are available
```

```
# (SOMTransformer, SOMTransformerBlock, SOMELayer, Expert, PositionalEncoding)
```

```
# --- Step 1: Re-instantiate the model with the exact trained architecture ---
```

```
# This uses the configuration variables from the previous cell to prevent errors.
```

```
print("--- Re-instantiating model for visualization ---")
```

```
model_viz = SOMTransformer(
```

```
    vocab_size=tokenizer.vocab_size,
```

```
    d_model=D_MODEL,
```

```
    num_heads=NUM_HEADS,
```

```

    num_layers=NUM_LAYERS,
    some_config=some_config
)

# --- Step 2: Load the saved weights ---
## V2 CHANGE: Updated model path to load the new V2 baseline weights.
model_path = "best_some_transformer_v2_baseline.pth"
print(f"--- Loading weights from {model_path} ---")
try:
    # We load onto the CPU for analysis to free up GPU VRAM
    model_viz.load_state_dict(torch.load(model_path, map_location="cpu"))
    print("Successfully loaded trained model weights.")
except Exception as e:
    print(f"Error loading model weights: {e}")
    print("Please ensure the model file exists and the architecture matches the saved weights.")

model_viz.eval()

# --- Step 3: Define a reusable plotting function ---
def plot_galaxy(layer_index, perplexity=30, n_iter=1000):
    """
    Extracts keys and usage from a specific layer, runs t-SNE, and plots the result.
    """
    print(f"\n--- Visualizing Layer {layer_index} ---")

    # Extract data from the specified layer
    try:
        layer_to_inspect = model_viz.layers[layer_index].some_layer
        keys = layer_to_inspect.key_store.detach().cpu().numpy()
        usage = layer_to_inspect.usage_count.detach().cpu().numpy()
    except IndexError:
        print(f"Error: Layer index {layer_index} is out of bounds for a model with {NUM_LAYERS} layers.")
    return

    print(f"Extracted {keys.shape[0]} keys. Total usage (EMA-scaled): {usage.sum():.2f}")

    # Handle the case where a layer might have zero usage if training was very short
    if usage.sum() == 0:
        print(f"Warning: Layer {layer_index} has zero expert usage. Cannot normalize size or color. Plotting uniformly.")
        usage_normalized = np.zeros_like(usage)
    else:
        # We can use the raw EMA value for color and a normalized version for size

```

```

usage_normalized_size = usage / usage.sum()

# Perform t-SNE dimensionality reduction
print(f"Running t-SNE with perplexity={perplexity}... (this may take a moment)")
tsne = TSNE(n_components=2, perplexity=perplexity, random_state=42, n_iter=n_iter,
init='pca', learning_rate='auto')
keys_2d = tsne.fit_transform(keys)
print("t-SNE complete.")

# Create the plot
fig, ax = plt.subplots(figsize=(16, 12))
scatter = ax.scatter(
    keys_2d[:, 0],
    keys_2d[:, 1],
    c=usage,
    s=20 + usage_normalized_size * 20000, # Increased multiplier for better size variation
    cmap='viridis',
    alpha=0.8,
    edgecolor='k',
    linewidth=0.5
)

cbar = fig.colorbar(scatter, ax=ax, pad=0.01)
cbar.set_label('Expert Activation Frequency (EMA Usage)', fontsize=14)

ax.set_title(f't-SNE Visualization of SoME Expert Key Space for Layer {layer_index}',
fontsize=18, pad=20)
ax.set_xlabel('t-SNE Dimension 1', fontsize=14)
ax.set_ylabel('t-SNE Dimension 2', fontsize=14)

info_text = (
    "How to Read This Plot:\n"
    "• Each circle represents one of the {num_experts} experts.\n"
    "• Proximity suggests conceptual similarity learned by the router.\n"
    "• Bright, large circles are high-usage 'generalist' experts.\n"
    "• Dark, small circles are low-usage 'specialist' or unused experts.\n"
    "Clusters of points are emerging 'Knowledge Galaxies'."
).format(num_experts=keys.shape[0])

ax.text(0.98, 0.02, info_text, transform=ax.transAxes, fontsize=12,
    verticalalignment='bottom', horizontalalignment='right',
    bbox=dict(boxstyle='round,pad=0.5', fc='aliceblue', alpha=0.8))

plt.tight_layout()

```

```
plt.savefig(f"expert_galaxy_layer_{layer_index}_v2.png", dpi=300)
print(f"Visualization for layer {layer_index} saved as
'expert_galaxy_layer_{layer_index}_v2.png")
plt.show()
```

```
# --- Step 4: Generate plots for key layers ---
```

```
# We visualize the first, middle, and last layers to see how organization evolves.
```

```
layers_to_plot = [0, NUM_LAYERS // 2, NUM_LAYERS - 1]
```

```
for layer_idx in layers_to_plot:
```

```
    plot_galaxy(layer_index=layer_idx)
```

```
--- Re-instantiating model for visualization ---
```

```
--- Loading weights from best_some_transformer_v2_baseline.pth ---
```

```
Successfully loaded trained model weights.
```

```
--- Visualizing Layer 0 ---
```

```
Extracted 256 keys. Total usage (EMA-scaled): 262144.25
```

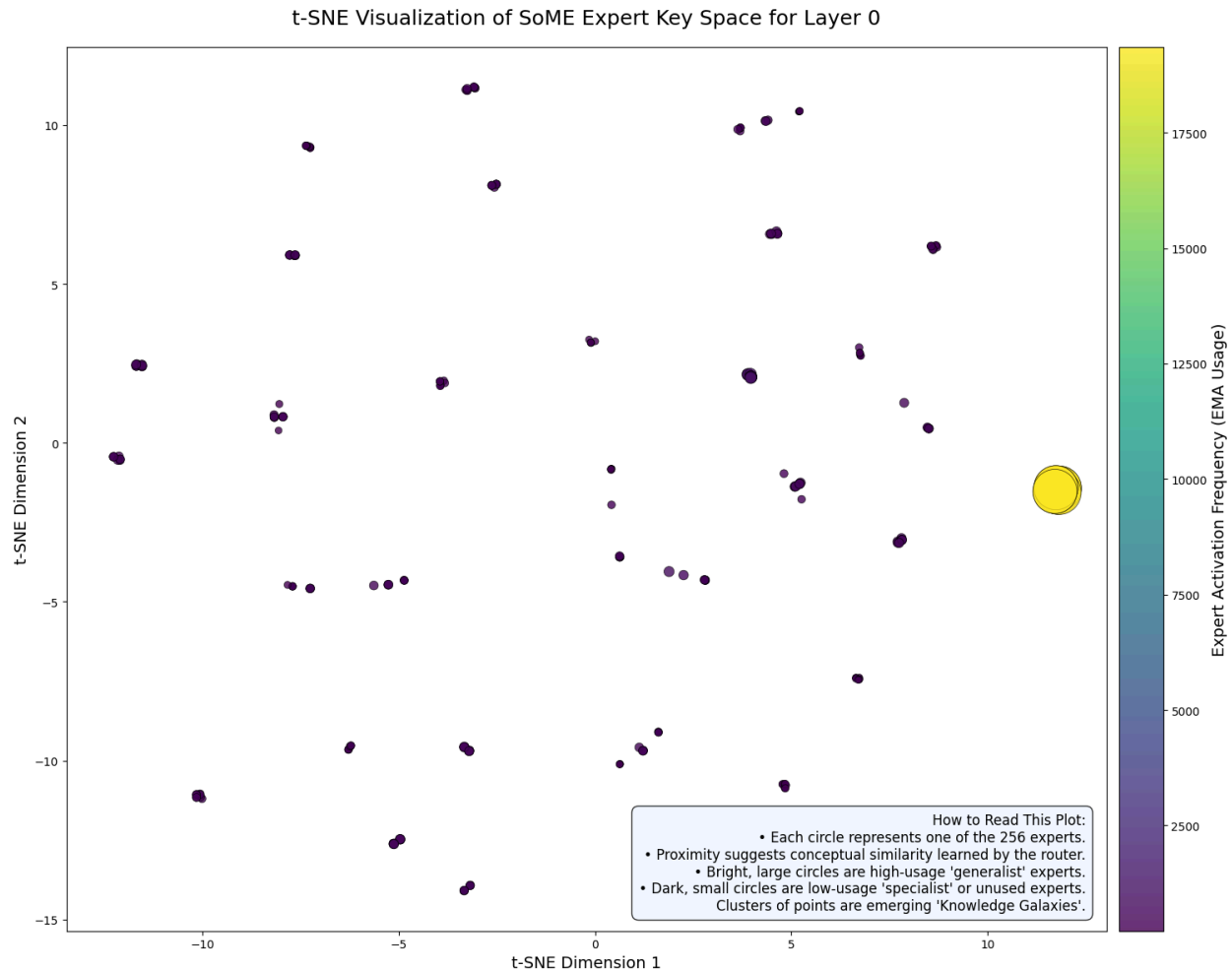
```
Running t-SNE with perplexity=30... (this may take a moment)
```

```
/usr/local/lib/python3.12/dist-packages/sklearn/manifold/_t_sne.py:1164: FutureWarning: 'n_iter'
was renamed to 'max_iter' in version 1.5 and will be removed in 1.7.
```

```
warnings.warn(
```

```
t-SNE complete.
```

```
Visualization for layer 0 saved as 'expert_galaxy_layer_0_v2.png'
```

--- Visualizing Layer 4 ---

Extracted 256 keys. Total usage (EMA-scaled): 262144.25

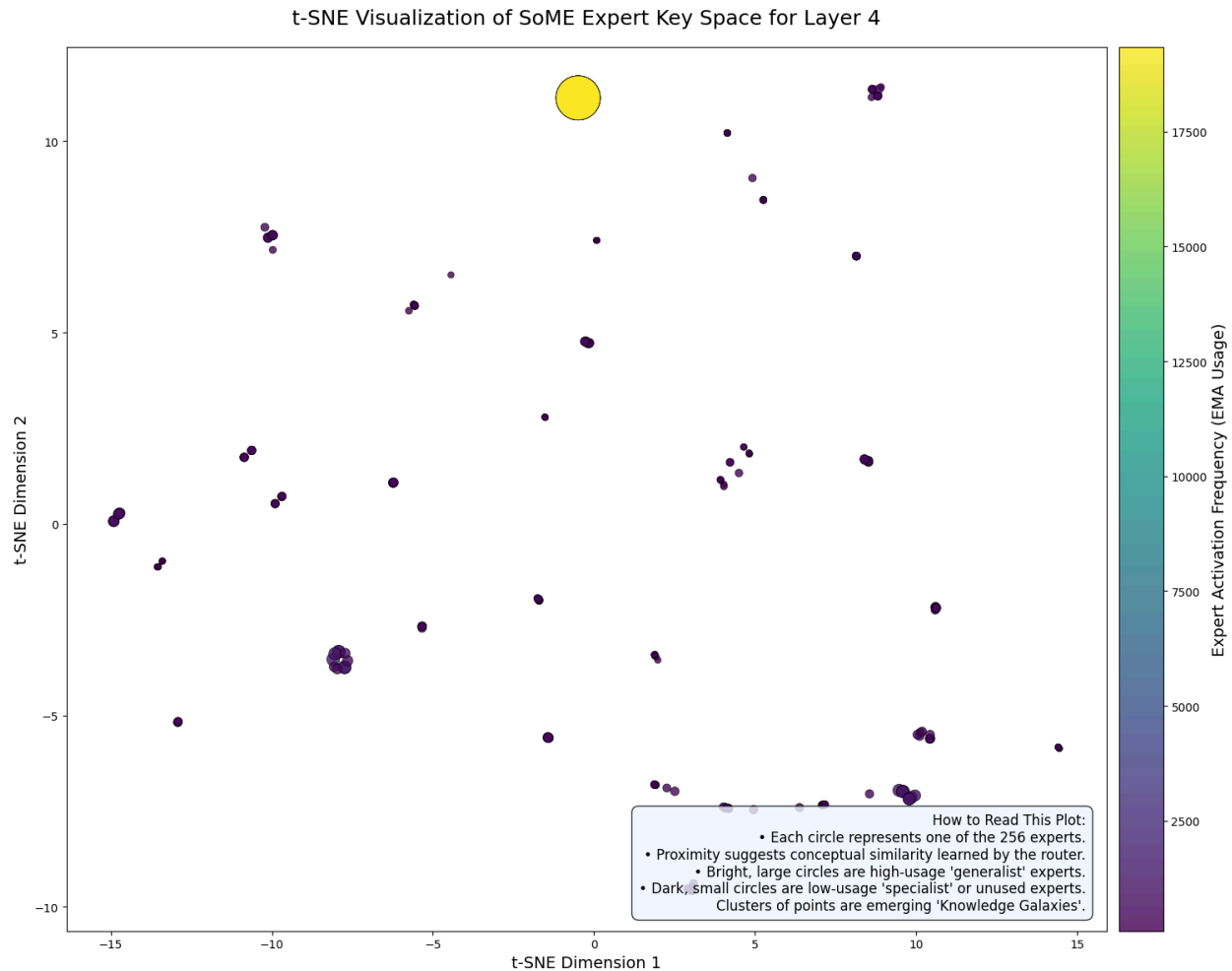
Running t-SNE with perplexity=30... (this may take a moment)

/usr/local/lib/python3.12/dist-packages/sklearn/manifold/_t_sne.py:1164: FutureWarning: 'n_iter' was renamed to 'max_iter' in version 1.5 and will be removed in 1.7.

warnings.warn(

t-SNE complete.

Visualization for layer 4 saved as 'expert_galaxy_layer_4_v2.png'



--- Visualizing Layer 7 ---

Extracted 256 keys. Total usage (EMA-scaled): 262144.28

Running t-SNE with perplexity=30... (this may take a moment)

/usr/local/lib/python3.12/dist-packages/sklearn/manifold/_t_sne.py:1164: FutureWarning: 'n_iter' was renamed to 'max_iter' in version 1.5 and will be removed in 1.7.

warnings.warn(

t-SNE complete.

Visualization for layer 7 saved as 'expert_galaxy_layer_7_v2.png'

t-SNE Visualization of SoME Expert Key Space for Layer 7

