

SoME v7 (The Clamp Fix) code

```
# Based on v6, correcting the "Immortal Expert" bug by removing forced normalization.
```

```
# Cell 1: Setup and Dependencies
```

```
print("--- Part 1: Setup and Dependencies ---")
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 (Crucial for 40GB optimization)
```

```
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 2: Core SoME Framework (v7 Clamp Fix)
```

```
# -----
```

```
# --- 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.numExperts = 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']

    # v6 Feature: Respawn Threshold (The "Phoenix" Trigger)
    self.respawn_threshold = some_config.get('respawn_threshold', 0.1)

    # Key management parameters
    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}
    )

```

```

print(f"SoME Layer Ablation Flags: {self.ablation_flags}")

# Router / query network (trainable)
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}!")

# Init query network
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)

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

# Usage tracking
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)

# Dead expert tracking for logging
self.register_buffer("dead_expert_count", 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)

    # 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)

    # Dot product scores.
    # Low magnitude (decayed) keys will naturally have low scores.
    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
    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]

    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

    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)

    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 = 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 (v7)
# Instead of F.normalize() which snaps norm to 1.0, we only clamp.
# This allows keys to have norm < 1.0 (preserving decay).
# We only shrink keys that have accidentally grown > 1.0 due to updates.
current_norms = self.key_store.norm(p=2, dim=-1, keepdim=True)
# Avoid division by zero, though unlikely with > 1.0 check
clamp_mask = (current_norms > 1.0)
# Only modify keys that are too large
if clamp_mask.any():
    self.key_store = torch.where(
        clamp_mask,
        self.key_store / current_norms,
        self.key_store
    )

# 4. Apply 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
        # Permanent shrinkage!
        self.key_store[low_usage_mask] *= (1.0 - self.delta)

# 5. The Phoenix Mechanism (Respawn)
# Recalculate norms after decay
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) # Track for logging

```

```

if num_dead > 0:
    # Pick random queries from the current batch to seed the new experts
    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()
        # Normalize new keys to 1.0 (Full Brightness)
        new_keys = F.normalize(new_keys + torch.randn_like(new_keys) * 0.01, p=2, dim=-1)

        # Overwrite dead keys
        self.key_store[dead_mask] = new_keys

        # Reset usage stats for respawned experts (fresh start)
        self.usage_count[dead_mask] = 0.0
        self.usage_mass[dead_mask] = 0.0

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()

        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):
        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)

# -----
# Cell 3: Data Preparation & Training (Standard)
# -----


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"

    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}")

    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))

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, targets, attention_mask = input_ids.to(device, non_blocking=True),
        targets.to(device, non_blocking=True), 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()

        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}'})

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, targets, attention_mask = input_ids.to(device), targets.to(device),
            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(config):
    print(f"\n--- Starting Experiment: {config['run_name']} ---")
    train_loader, val_loader, tokenizer = prepare_data(config)

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

    optimizer = torch.optim.AdamW([p for p in model.parameters() if p.requires_grad],
                                lr=config['training']['learning_rate'], weight_decay=0.1)
    criterion = nn.CrossEntropyLoss(ignore_index=-100)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=len(train_loader)
* config['training']['num_epochs'])

    print("\n--- Part 4: Training ---")
    for epoch in range(config['training']['num_epochs']):
        train_loss = train_epoch(model, train_loader, optimizer, criterion, scheduler,
                                config['training']['training_temp'], config['model']['vocab_size'])
        val_loss = evaluate_epoch(model, val_loader, criterion, config['model']['vocab_size'],
                                config['training']['eval_temp'])

        model_state = model._orig_mod if hasattr(model, '_orig_mod') else model
        mid_layer = model_state.layers[config['model']['num_layers'] // 2].some_layer
        dead_count = mid_layer.dead_expert_count.item()


```

```

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}, Phoenix Respawns (Last Step)={dead_count}")

torch.save(model_state.state_dict(), f"best_model_{config['run_name']}.pth")

# -----
# Cell 4: Config & Run
# -----
config = {
    "run_name": "v7_ClampFix_Phoenix",
    "data": { "train_subset_size": 20000, "val_subset_size": 2000, "batch_size": 32 },
    "model": { "vocab_size": 8192, "d_model": 512, "num_heads": 8, "num_layers": 8, "seq_len": 768 },
    "some_layer": {
        "num_experts": 256,
        "d_ffn": 1024,
        "top_k": 4,
        "init_method": "sparse",
        "alpha": 0.015,
        "beta": 0.001,
        "delta": 0.005, # Decay
        "respawn_threshold": 0.1, # Phoenix Threshold
        "theta_percentile": 0.05,
        "warmup_steps": 200,
        "ema_decay": 0.995,
        "router_type": "mlp", "router_mlp_mult": 2.0,
        "ablation_flags": {"use_alpha": True, "use_beta": True, "use_delta": True}
    },
    "training": { "num_epochs": 4, "learning_rate": 6e-4, "training_temp": 1.0, "eval_temp": 1.0 }
}

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

```