SoME v2.1.2 "Darwinian" Curation Code

```
#
================================================================================
======
# 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 2: "Darwinian" Curation, Model Definition, Training, and Evaluation
#
================================================================================
======

# Set a seed for reproducibility of the expert curation process
torch.manual_seed(42)
np.random.seed(42)

# Part 2: Data Preparation & Configuration
# ------------------------------------------------------------------------
print("\n--- Part 2: Data Preparation & Configuration ---")
SEQ_LEN = 512
BATCH_SIZE = 256
VOCAB_SIZE = 8192
train_subset_size = 20000
val_subset_size = 4000

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=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(train_subset_size))
val_subset = full_dataset['validation'].select(range(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=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'])
        targets = inputs.clone()
        targets[:-1] = inputs[1:]
        targets[-1] = -100
        return inputs, targets

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=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 (With Curation)
# ------------------------------------------------------------------------------
print("\n--- Part 3: Model Definition ---")

class Expert(nn.Module):
    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)))

def create_curated_expert_pool(num_total_candidates, num_to_select, d_model, d_ffn,
init_method, fitness_method='diversity', device='cuda'):
    print(f"\n--- Starting Darwinian Curation of Experts ---")
    print(f"Generating {num_total_candidates} candidates to select the top {num_to_select} using
'{fitness_method}' fitness...")
    candidate_pool = [Expert(d_model, d_ffn, init_method=init_method).to(device) for _ in
range(num_total_candidates)]
    sample_data = torch.randn(128, SEQ_LEN, d_model, device=device)

    if fitness_method == 'diversity':
        with torch.no_grad():
            outputs = [expert(sample_data).mean(dim=[0, 1]) for expert in tqdm(candidate_pool,
desc="Evaluating candidates")]
            outputs_tensor = F.normalize(torch.stack(outputs), p=2, dim=1)
        sim_matrix = torch.matmul(outputs_tensor, outputs_tensor.T)
        selected_indices = [0]
        for _ in tqdm(range(num_to_select - 1), desc="Selecting diverse experts"):
            min_avg_sim, best_candidate_idx = float('inf'), -1
            for i in range(num_total_candidates):
                if i in selected_indices: continue
                avg_sim = sim_matrix[i, selected_indices].mean().item()
                if avg_sim < min_avg_sim:
                    min_avg_sim, best_candidate_idx = avg_sim, i
            selected_indices.append(best_candidate_idx)
        curated_experts = [candidate_pool[i] for i in selected_indices]
    elif fitness_method == 'sparsity':
        scores = []
        with torch.no_grad():
            for expert in tqdm(candidate_pool, desc="Evaluating candidates"):
                hidden_activations = expert.activation(expert.w_down(sample_data))
                scores.append((hidden_activations.abs() < 1e-3).float().mean().item())
        top_indices = np.argsort(scores)[::-1][:num_to_select]
        curated_experts = [candidate_pool[i] for i in top_indices]
    else: raise ValueError(f"Unknown fitness_method: {fitness_method}")
    print(f"--- Curation Complete. Selected {len(curated_experts)} experts. ---")
```

```
    return nn.ModuleList(curated_experts)

class SOMELayer(nn.Module):
    def __init__(self, d_model, experts_module_list, top_k, alpha, beta, delta, theta_percentile,
warmup_steps, ema_decay):
        super().__init__()
        self.d_model, self.top_k = d_model, top_k
        self.alpha, self.beta, self.delta = alpha, beta, delta
        self.theta_percentile, self.warmup_steps, self.ema_decay = theta_percentile,
warmup_steps, ema_decay
        self.experts = experts_module_list
        self.num_experts = len(self.experts)
        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))
        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, t=1.0):
        bs, sl, _ = x.shape
        xf = x.view(-1, self.d_model)
        q = F.normalize(self.query_network(xf), p=2, dim=-1)
        s = torch.matmul(q, self.key_store.t())
        st, it = torch.topk(s, self.top_k, dim=-1)
        gw = F.softmax(st / t, dim=-1)
        fit = it.view(-1)
        si, pm = torch.sort(fit)
        ue, c = torch.unique_consecutive(si, return_counts=True)
        fi = xf.repeat_interleave(self.top_k, dim=0)
        pi = fi[pm]
        spi = torch.split(pi, c.tolist(), dim=0)
        oc = [self.experts[eid](spi[i]) for i, eid in enumerate(ue)]
        co = torch.cat(oc, dim=0)
        ipm = torch.argsort(pm)
        eo = co[ipm]
        wo = (eo.view(-1, self.top_k, self.d_model) * gw.unsqueeze(-1)).sum(dim=1)
        return x + wo.view(bs, sl, self.d_model), q, it
    @torch.no_grad()
    def update_keys(self, q, it):
        self.steps += 1
        ui, c = torch.unique(it, return_counts=True)
```

```
        self.usage_count.mul_(self.ema_decay).index_add_(0, ui, (1.0-self.ema_decay)*c.float())
        for i in range(self.top_k):
            ind = it[:, i]
            ine = 1.0 + self.usage_count[ind]
            ae = self.alpha / ine.unsqueeze(-1)
            uv = q - self.key_store[ind]
            self.key_store.index_add_(0, ind, ae * uv)
        if self.top_k > 1:
            ii = it[:, self.peer_pull_indices[:,0]].reshape(-1)
            ij = it[:, self.peer_pull_indices[:,1]].reshape(-1)
            ki, kj = self.key_store[ii], self.key_store[ij]
            inei, inej = (1.0+self.usage_count[ii]).unsqueeze(-1),
(1.0+self.usage_count[ij]).unsqueeze(-1)
            be = self.beta / torch.min(inei, inej)
            uvi, uvj = be * (kj - ki), be * (ki - kj)
            self.key_store.index_add_(0, ii, uvi).index_add_(0, ij, uvj)
        self.key_store.data = F.normalize(self.key_store.data, p=2, dim=-1)
        if self.steps > self.warmup_steps:
            auc = self.usage_count[self.usage_count > 0]
            if auc.numel() > 0:
                dt = torch.quantile(auc.float(), self.theta_percentile)
                lum = self.usage_count < dt
                self.key_store[lum] *= (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, self.norm2 = nn.LayerNorm(d_model), nn.LayerNorm(d_model)
        self.some_layer = some_layer
    def forward(self, x, temperature=1.0):
        mask = torch.triu(torch.ones(x.size(1), x.size(1), device=x.device)*float('-inf'), diagonal=1)
        ax, _ = self.attention(x, x, x, attn_mask=mask)
        x = self.norm1(x + ax)
        sx, q, it = self.some_layer(x, t=temperature)
        return self.norm2(sx), q, it

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pos = torch.arange(max_len).unsqueeze(1)
        div = 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(pos * div)
```

```
        pe[0, :, 1::2] = torch.cos(pos * div)
        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,
curated_experts_list):
        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,
experts_module_list=curated_experts_list, **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)
        aq, ai = [], []
        for layer in self.layers:
            x, q, it = layer(x, temperature=temperature)
            aq.append(q); ai.append(it)
        return self.fc_out(x), aq, ai
    @torch.no_grad()
    def update_all_keys(self, aq, ai):
        for i, lb in enumerate(self.layers):
            q = aq[i].view(-1, lb.some_layer.d_model)
            it = ai[i].view(-1, lb.some_layer.top_k)
            lb.some_layer.update_keys(q, it)

# Part 4: Training, Evaluation, and Metrics (Functions)
# ... (These utility functions are defined exactly as before) ...
print("\n--- Part 4: Training, Evaluation, and Metrics ---")
scaler=torch.amp.GradScaler("cuda")
def calculate_gini(uc):
    c=uc.cpu().to(torch.float32).numpy()
    if np.sum(c)==0:return 0.0
    c=np.sort(c);n=len(c);i=np.arange(1,n+1)
    return (np.sum((2*i-n-1)*c))/(n*np.sum(c))
def calculate_entropy(uc):
    tu=uc.sum()
    if tu==0:return 0.0
    p=uc/tu;p=p[p>0]
    return -torch.sum(p*torch.log2(p)).item()
```

```
def train_epoch(m,dl,o,c,s,ct):
    m.train();tl=0;pb=tqdm(dl,desc="Training",leave=False)
    for i,t in pb:
        i,t=i.to(device,non_blocking=True),t.to(device,non_blocking=True)
        with
torch.amp.autocast("cuda"):l,q,ind=m(i,temperature=ct);loss=c(l.view(-1,VOCAB_SIZE),t.view(-1
))

o.zero_grad(set_to_none=True);scaler.scale(loss).backward();scaler.unscale_(o);torch.nn.utils.c
lip_grad_norm_(m.parameters(),1.0);scaler.step(o);scaler.update();s.step();m.update_all_keys(q
,ind);tl+=loss.item();pb.set_postfix({'loss':f'{loss.item():.4f}','lr':f'{s.get_last_lr()[0]:.1e}'})
    return tl/len(dl)
def evaluate_epoch(m,dl,c):
    m.eval();tl=0;pb=tqdm(dl,desc="Evaluating",leave=False)
    with torch.no_grad():
        for i,t in pb:
            i,t=i.to(device,non_blocking=True),t.to(device,non_blocking=True)
            with
torch.amp.autocast("cuda"):l,_,_=m(i,temperature=0.5);loss=c(l.view(-1,VOCAB_SIZE),t.view(-1
))
            tl+=loss.item();pb.set_postfix({'loss':f'{loss.item():.4f}'})
    return tl/len(dl)
def plot_losses(tls,vls,e,fm):
    plt.figure(figsize=(10,6));plt.plot(range(1,e+1),tls,'b-o',label='Training
Loss');plt.plot(range(1,e+1),vls,'r-o',label='Validation Loss');plt.title(f'Training Loss (Curated -
{fm.capitalize()})');plt.xlabel('Epochs');plt.ylabel('Loss');plt.legend();plt.grid(True);plt.xticks(range(
1,e+1));fn=f'loss_curve_v2.1_curated_{fm}.png';plt.savefig(fn);print(f"\nLoss curve saved to
{fn}");plt.show()

# Part 5: Main Execution Block
# ------------------------------------------------------------------------------
print("\n--- Part 5: Main Execution Block ---")
D_MODEL,NUM_HEADS,NUM_LAYERS=384,6,6
some_config={"top_k":4,"alpha":0.015,"beta":0.001,"delta":0.001,"theta_percentile":0.05,"warmu
p_steps":400,"ema_decay":0.995}
NUM_EXPERTS_TO_SELECT,D_FFN=128,1024
INIT_METHOD,FITNESS_METHOD='orthogonal','diversity'
NUM_CANDIDATES=2048
curated_expert_pool=create_curated_expert_pool(NUM_CANDIDATES,NUM_EXPERTS_TO_
SELECT,D_MODEL,D_FFN,INIT_METHOD,FITNESS_METHOD,device)
NUM_EPOCHS,LEARNING_RATE,TRAINING_TEMP=6,8e-4,1.0
model_save_path=f"best_some_transformer_v2.1_curated_{FITNESS_METHOD}.pth"
model=SOMETransformer(VOCAB_SIZE,D_MODEL,NUM_HEADS,NUM_LAYERS,some_confi
g,curated_expert_pool).to(device)
```

```
if hasattr(torch,'compile'):print("\nCompiling model...");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 params: {sum(p.numel() for p in model.parameters())/1e6:.2f}M, Trainable:
{sum(p.numel() for p in model.parameters() if p.requires_grad)/1e6:.2f}M")
print(f"Total training steps: {total_steps}, using fitness: {FITNESS_METHOD}")
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,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
    uc=model_to_inspect.layers[NUM_LAYERS//2].some_layer.usage_count
    gini,entropy=calculate_gini(uc),calculate_entropy(uc)
    print(f"Epoch {epoch+1}: Train Loss={train_loss:.4f}, Val Loss={val_loss:.4f}, Val
PPL={perplexity:.2f}")
    print(f"  Mid Layer Metrics: Gini={gini:.3f}, Entropy={entropy:.3f}")
    if val_loss<best_val_loss:

best_val_loss=val_loss;torch.save(model_to_inspect.state_dict(),model_save_path);print(f"Mod
el saved to {model_save_path}")
print(f"\n--- Training Complete ---")
plot_losses(train_losses,val_losses,NUM_EPOCHS,FITNESS_METHOD)

#
================================================================================
======
# Cell 3: Interactive Inference
#
================================================================================
======

print("\n--- Part 6: Interactive Inference ---")

# Set the same seed to ensure the exact same expert pool is generated for model loading
torch.manual_seed(42)
np.random.seed(42)
```

```
def generate(model, tokenizer, prompt, max_new_tokens=100, temperature=0.7, top_k=50,
device="cuda"):
    """Generates text from a trained SoME model using temperature and top-k sampling."""
    model.eval()
    input_ids = tokenizer.encode(prompt, return_tensors="pt").to(device)
    generated_ids = input_ids
    print(f"\n--- Prompt ---\n{prompt}", end="")
    with torch.no_grad():
        for _ in range(max_new_tokens):
            with torch.amp.autocast("cuda"):
                outputs, _, _ = model(generated_ids)
            next_token_logits = outputs[:, -1, :]
            if temperature > 0: next_token_logits = next_token_logits / temperature
            top_k_logits, top_k_indices = torch.topk(next_token_logits, top_k)
            probs = F.softmax(top_k_logits, dim=-1)
            next_token_relative_id = torch.multinomial(probs, num_samples=1)
            next_token_id = torch.gather(top_k_indices, -1, next_token_relative_id)
            if next_token_id.item() == tokenizer.eos_token_id: break
            generated_ids = torch.cat((generated_ids, next_token_id), dim=1)
            print(tokenizer.decode(next_token_id[0]), end="", flush=True)
    print("\n--- End of Generation ---")

# --- Setup Inference Model ---
print("\nSetting up model for inference...")

# Re-create the exact configuration from the training cell
D_MODEL_INF, NUM_HEADS_INF, NUM_LAYERS_INF = 384, 6, 6
some_config_inf = {"top_k":4, "alpha":0.015, "beta":0.001, "delta":0.001, "theta_percentile":0.05,
"warmup_steps":400, "ema_decay":0.995}
NUM_EXPERTS_INF, D_FFN_INF = 128, 1024
INIT_METHOD_INF, FITNESS_METHOD_INF = 'orthogonal', 'diversity'
NUM_CANDIDATES_INF = 2048
model_path_to_load = f"best_some_transformer_v2.1_curated_{FITNESS_METHOD_INF}.pth"

if os.path.exists(model_path_to_load):
    # Re-run the curation process. Due to the fixed seed, this will select the *exact same*
experts.
    curated_experts_for_inf = create_curated_expert_pool(
        num_total_candidates=NUM_CANDIDATES_INF,
        num_to_select=NUM_EXPERTS_INF,
        d_model=D_MODEL_INF,
        d_ffn=D_FFN_INF,
        init_method=INIT_METHOD_INF,
        fitness_method=FITNESS_METHOD_INF,
```

```
        device=device
    )

    inference_model = SOMETransformer(
        vocab_size=VOCAB_SIZE, d_model=D_MODEL_INF, num_heads=NUM_HEADS_INF,
        num_layers=NUM_LAYERS_INF, some_config=some_config_inf,
curated_experts_list=curated_experts_for_inf
    ).to(device)

    inference_model.load_state_dict(torch.load(model_path_to_load))
    print(f"Successfully loaded weights from {model_path_to_load}")

    # --- Talk to the Model! ---
    prompts = [
        "Once upon a time, there was a brave knight who",
        "The secret to making the best pizza is",
        "A lonely robot sat on a hill watching the stars. It wondered,"
    ]
    for p in prompts:
        generate(inference_model, tokenizer, p, max_new_tokens=80, temperature=0.8, top_k=50)
        print("-" * 30)
else:
    print(f"Could not find a saved model at '{model_path_to_load}'. Please run the training cell
first.")
```