SoME Colab Code v2.1

```
#
================================================================================
======
# 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: Model Definition, Training, and Evaluation
#
================================================================================
======

# Part 2: Data Preparation & Configuration
# --------------------------------------------------------------------------
print("\n--- Part 2: Data Preparation & Configuration ---")

# -- SoME V2.1 "Honest Speedrun" Configuration --
SEQ_LEN = 512
BATCH_SIZE = 256  # Increased for A100 throughput
VOCAB_SIZE = 8192
train_subset_size = 20000  # Reduced for speed
val_subset_size = 4000  # Reduced for speed

# -- Loading/Training Custom Tokenizer ---
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}")

# --- Tokenizing Dataset ---
print("\nTokenizing dataset...")
full_dataset = load_dataset("roneneldan/TinyStories", streaming=False) # Use non-streaming
for select
```

```
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
        targets = inputs.clone()
        targets[:-1] = inputs[1:]
        targets[-1] = -100  # Ignore 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)
# ------------------------------------------------------------------------------
print("\n--- Part 3: Model Definition ---")

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)

        # === NEW: STRUCTURED INITIALIZATION LOGIC ===
        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, num_experts, d_ffn, top_k, alpha, beta, delta, theta_percentile,
warmup_steps, ema_decay, init_method='default'):
        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.ema_decay = ema_decay
        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))

        # === MODIFIED: Pass init_method to each Expert ===
        self.experts = nn.ModuleList([Expert(d_model, d_ffn, init_method=init_method) 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))

    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, 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)

        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

    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(max_len, 1, 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.transpose(0, 1))
    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
# ------------------------------------------------------------------------------
```

```
print("\n--- 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()
        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) # 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, epochs, init_method):
    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(f'Training and Validation Loss (V2.1 Speedrun - {init_method.capitalize()} Init)')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.xticks(range(1, epochs + 1))
    filename = f'loss_curve_v2.1_speedrun_{init_method}.png'
    plt.savefig(filename)
    print(f"\nLoss curve plot saved to {filename}")
    plt.show()


# Part 5: Main Execution Block
# ------------------------------------------------------------------------------
print("\n--- Part 5: Main Execution Block ---")

# -- Model Dimensions --
D_MODEL = 384
NUM_HEADS = 6
NUM_LAYERS = 6

# -- SoME V2.1 Hyperparameters --
some_config = {
    "num_experts": 128,
    "d_ffn": 1024,
    "top_k": 4,
    "alpha": 0.015,
    "beta": 0.001,
    "delta": 0.001,
```

```
    "theta_percentile": 0.05,
    "warmup_steps": 400,
    "ema_decay": 0.995,
    # === CONTROL YOUR EXPERIMENT HERE ===
    # Options: 'default', 'orthogonal', 'sparse'
    "init_method": "orthogonal"
}

# -- Training Schedule --
NUM_EPOCHS = 6
LEARNING_RATE = 8e-4
TRAINING_TEMP = 1.0

# -- Initialization --
model_save_path = f"best_some_transformer_v2.1_{some_config['init_method']}.pth"
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}")
print(f"Using expert initialization method: {some_config['init_method']}")


# -- 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(), model_save_path)
        print(f"Model saved as {model_save_path}")

print(f"\n--- V2.1 'Honest Speedrun' Training Complete ---")
plot_losses(train_losses, val_losses, NUM_EPOCHS, some_config['init_method'])
```

```
--- Part 2: Data Preparation & Configuration ---
Training custom tokenizer...

README.md:
  1.06k/?  [00:00<00:00,  114kB/s]
data/train-00000-of-00004-2d5a1467fff108(…):  100%
  249M/249M  [00:02<00:00,  251MB/s]
data/train-00001-of-00004-5852b56a2bd28f(…):  100%
  248M/248M  [00:01<00:00,  310MB/s]
data/train-00002-of-00004-a26307300439e9(…):  100%
  246M/246M  [00:01<00:00,  132MB/s]
data/train-00003-of-00004-d243063613e5a0(…):  100%
  248M/248M  [00:01<00:00,  122MB/s]
data/validation-00000-of-00001-869c898b5(…):  100%
  9.99M/9.99M  [00:00<00:00,  20.8MB/s]
Generating  train  split:  100%
  2119719/2119719  [00:06<00:00,  339467.91  examples/s]
Generating  validation  split:  100%
  21990/21990  [00:00<00:00,  316045.23  examples/s]
```

Custom tokenizer loaded with vocab size: 8192

Tokenizing dataset...

Map (num_proc=12): 100%
20000/20000 [00:03<00:00, 6167.38 examples/s]
Map (num_proc=12): 100%
4000/4000 [00:00<00:00, 434.57 examples/s]
Train dataset size (subset): 20000
Using 6 workers for DataLoader.

--- Part 3: Model Definition ---

--- Part 4: Training, Evaluation, and Metrics ---

--- Part 5: Main Execution Block ---

Compiling the model for faster training...

Total parameters: 615.81M
Trainable parameters: 10.74M
Total training steps: 468
Using expert initialization method: orthogonal

--- Epoch 1/6 ---

Training: 0%| | 0/78 [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 = 2.3959, Val Loss = 1.5460, Val Perplexity = 4.69
Middle Layer Expert Metrics: Gini = 0.755, Entropy = 4.770
Model saved as best_some_transformer_v2.1_orthogonal.pth

--- Epoch 2/6 ---

Epoch 2: Train Loss = 1.5637, Val Loss = 1.3744, Val Perplexity = 3.95
Middle Layer Expert Metrics: Gini = 0.750, Entropy = 4.748
Model saved as best_some_transformer_v2.1_orthogonal.pth

--- Epoch 3/6 ---

Epoch 3: Train Loss = 1.4259, Val Loss = 1.2912, Val Perplexity = 3.64
  Middle Layer Expert Metrics: Gini = 0.750, Entropy = 4.736
Model saved as best_some_transformer_v2.1_orthogonal.pth

--- Epoch 4/6 ---

Epoch 4: Train Loss = 1.3462, Val Loss = 1.2458, Val Perplexity = 3.48
  Middle Layer Expert Metrics: Gini = 0.755, Entropy = 4.716
Model saved as best_some_transformer_v2.1_orthogonal.pth

--- Epoch 5/6 ---

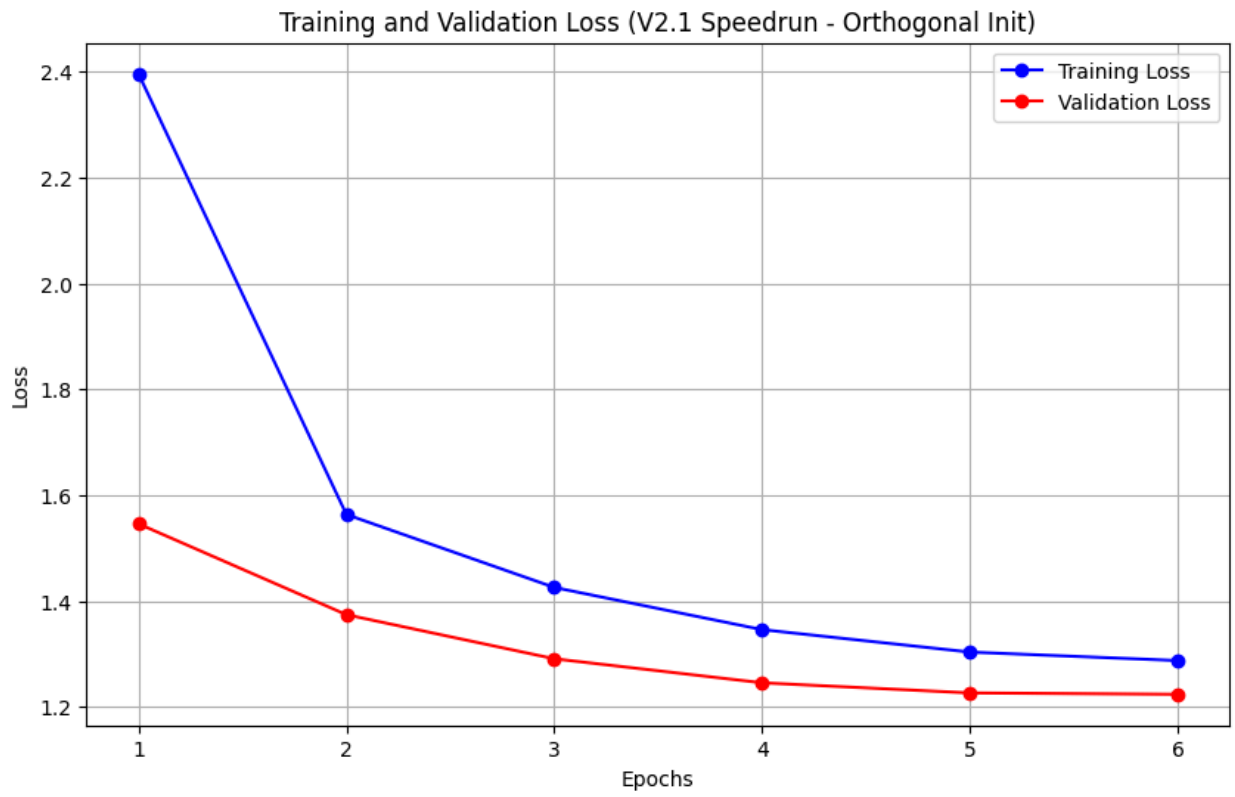
Epoch 5: Train Loss = 1.3038, Val Loss = 1.2266, Val Perplexity = 3.41
  Middle Layer Expert Metrics: Gini = 0.760, Entropy = 4.700
Model saved as best_some_transformer_v2.1_orthogonal.pth

--- Epoch 6/6 ---

Epoch 6: Train Loss = 1.2878, Val Loss = 1.2241, Val Perplexity = 3.40
  Middle Layer Expert Metrics: Gini = 0.763, Entropy = 4.690
Model saved as best_some_transformer_v2.1_orthogonal.pth

--- V2.1 'Honest Speedrun' Training Complete ---

Loss curve plot saved to loss_curve_v2.1_speedrun_orthogonal.png



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

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

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, :]

            # Apply temperature
            if temperature > 0:
                next_token_logits = next_token_logits / temperature

            # Apply top-k filtering
            top_k_logits, top_k_indices = torch.topk(next_token_logits, top_k)
            probs = F.softmax(top_k_logits, dim=-1)

            # Sample the next token
            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)

            new_token = tokenizer.decode(next_token_id[0])
            print(new_token, end="", flush=True)

    print("\n--- End of Generation ---")


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

# Use the same configuration as the training cell
inference_config = some_config
model_path_to_load = model_save_path

if os.path.exists(model_path_to_load):
    inference_model = SOMETransformer(
        vocab_size=tokenizer.vocab_size, d_model=D_MODEL, num_heads=NUM_HEADS,
        num_layers=NUM_LAYERS, some_config=inference_config
    ).to(device)

    inference_model.load_state_dict(torch.load(model_path_to_load))
    print(f"Loaded weights from {model_path_to_load}")
```

```
    # Note: torch.compile can sometimes have issues with dynamic shapes in generation
    # It's often safer to run inference without it unless tested thoroughly.

    # --- 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.")
```

--- Part 6: Interactive Inference ---

Setting up model for inference...
Loaded weights from best_some_transformer_v2.1_orthogonal.pth

--- Prompt ---
Once upon a time, there was a brave knight
whowasverynervous.Hewantedtogototheworldaroundthemarket.Helookedforabig,it,buthecouldn't
findtodo.Sohesawthatit'snotlookingforthebest.Hedidn'tknowwhattodo.Hetriedtohelpbeforeitupand
makeitveryspecial.Suddenly,aholewasakindofotherchildren.Hewas
--- End of Generation ---

--- Prompt ---
The secret to making the best pizza
isthat."Iwanttobuyitcan."So,theyheardabigcar.Thetruckwasscared.Hetoldittohismum.Thecarwast
hecar.Hewassohewantedtogetitahat.Itwasabigandbrightnewfriend.Thecarwashappyandhappy.Th
eyworkedhardandplayed.Theyallthecarandtheanimals.Itwasthe
--- End of Generation ---

--- Prompt ---
A lonely robot sat on a hill watching the stars. It
wondered,andalittlegirlnamed"Thatboy!"Theboywasverycurious."Whatcanweplayinthegarden?""
CanIhaveaplan?"Hismomlookedforabig,butshewantedtoseetheboy."Idon'tworry,sweetheart.Iwant
topickourthingsinthewoods.Maybeoneday,alittlegirlnamedSue.Shehadalotof

--- End of Generation ---
