Multi-Task SoME

```
#
================================================================================
======
# Cell 1: Setup and Dependencies (Unchanged)
#
================================================================================
======

print("--- Multi-Task SoME: Part 1: Setup and Dependencies ---")
!pip install torch datasets transformers huggingface_hub tokenizers matplotlib sentencepiece -q

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset, ConcatDataset
from transformers import AutoModelForCausalLM, 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
from tqdm import tqdm
import math
import os
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive

# Mount Google Drive for persistent storage
try:
    drive.mount('/content/drive')
    DRIVE_MOUNTED = True
    print("Google Drive mounted successfully.")
except Exception as e:
    DRIVE_MOUNTED = False
    print(f"Could not mount Google Drive: {e}. Results will be ephemeral.")

# Hardware setup
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
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

print("\nSetup complete.")


#
================================================================================
======
# Cell 2 (Corrected): Core SoME Framework (with Professional Hybridization)
#
================================================================================
======

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 == 'sparse':
            nn.init.sparse_(self.w_down.weight, sparsity=0.5)
            nn.init.sparse_(self.w_up.weight, sparsity=0.5)
        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):
    """SoME Layer upgraded with multiple, isolated key stores for multi-task learning."""
    def __init__(self, d_model, some_config, num_tasks, pretrained_ffn_state_dict=None):
        super().__init__()
        self.d_model, self.num_experts, self.d_ffn, self.top_k = d_model,
some_config['num_experts'], some_config['d_ffn'], some_config['top_k']
        self.alpha, self.beta, self.delta = some_config['alpha'], some_config['beta'],
some_config['delta']
        self.theta_percentile, self.warmup_steps, self.ema_decay =
some_config['theta_percentile'], some_config['warmup_steps'], some_config['ema_decay']

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

        self.num_tasks = num_tasks
        key_stores = torch.stack([F.normalize(torch.randn(self.num_experts, d_model), p=2,
dim=-1) for _ in range(num_tasks)])
```

```
        usage_counts = torch.stack([torch.zeros(self.num_experts) for _ in range(num_tasks)])
        self.register_buffer("key_stores", key_stores)
        self.register_buffer("usage_counts", usage_counts)
        self.register_buffer("steps", torch.zeros(num_tasks, dtype=torch.long))

        self.experts = nn.ModuleList()
        num_pretrained = self.num_experts // 2 if pretrained_ffn_state_dict else 0
        num_random = self.num_experts - num_pretrained
        if pretrained_ffn_state_dict:
            print(f"  -> Initializing {num_pretrained} experts from pre-trained Pro FFN.")
            for _ in range(num_pretrained):
                expert = Expert(d_model, self.d_ffn, 'default')
                expert.w_down.load_state_dict(pretrained_ffn_state_dict['down'])
                expert.w_up.load_state_dict(pretrained_ffn_state_dict['up'])
                self.experts.append(expert)

        # --- CORRECTED PRINT STATEMENT ---
        print(f"  -> Initializing {num_random} experts with '{some_config['init_method']}' method.")
        for _ in range(num_random):
            self.experts.append(Expert(d_model, self.d_ffn,
init_method=some_config['init_method']))
        for p in self.experts.parameters(): p.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, task_id, temperature=1.0):
        key_store = self.key_stores[task_id]
        batch_size, seq_len, _ = x.shape
        x_flat = x.view(-1, self.d_model)
        queries = F.normalize(self.query_network(x_flat), p=2, dim=-1)
        scores = torch.matmul(queries, 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)
        sorted_indices, permutation_map = torch.sort(top_k_indices.view(-1))
        unique_expert_ids, counts = torch.unique_consecutive(sorted_indices,
return_counts=True)
        flat_inputs = x_flat.repeat_interleave(self.top_k, dim=0)
        split_inputs = torch.split(flat_inputs[permutation_map], counts.tolist(), dim=0)
        output_chunks = [self.experts[expert_id](split_inputs[i]) for i, expert_id in
enumerate(unique_expert_ids)]
        expert_outputs = torch.cat(output_chunks, dim=0)[torch.argsort(permutation_map)]
        weighted_outputs = (expert_outputs.view(-1, self.top_k, self.d_model) *
gating_weights.unsqueeze(-1)).sum(dim=1)
```

```
        return x + weighted_outputs.view(batch_size, seq_len, self.d_model), queries,
top_k_indices

    @torch.no_grad()
    def update_keys(self, queries, top_k_indices, task_id):
        key_store, usage_count, steps = self.key_stores[task_id], self.usage_counts[task_id],
self.steps[task_id]
        steps.add_(1)
        unique_indices, counts = torch.unique(top_k_indices, return_counts=True)
        usage_count.mul_(self.ema_decay).index_add_(0, unique_indices, (1.0 - self.ema_decay)
* counts.float())
        for i in range(self.top_k):
            indices = top_k_indices[:, i]
            alpha_eff = self.alpha / (1.0 + usage_count[indices]).unsqueeze(-1)
            key_store.index_add_(0, indices, alpha_eff * (queries - key_store[indices]))
        if self.top_k > 1:
            i, j = top_k_indices[:, self.peer_pull_indices[:, 0]].reshape(-1), top_k_indices[:,
self.peer_pull_indices[:, 1]].reshape(-1)
            beta_eff = self.beta / torch.min((1.0 + usage_count[i]).unsqueeze(-1), (1.0 +
usage_count[j]).unsqueeze(-1))
            key_store.index_add_(0, i, beta_eff * (key_store[j] - key_store[i]))
            key_store.index_add_(0, j, beta_eff * (key_store[i] - key_store[j]))
        if steps > self.warmup_steps and usage_count.sum() > 0:
            active_counts = usage_count[usage_count > 0]
            if active_counts.numel() > 0:
                key_store[usage_count < torch.quantile(active_counts, self.theta_percentile)] *= (1.0 -
self.delta)
        key_store.data = F.normalize(key_store.data, p=2, dim=-1)

class TransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, some_config, num_tasks, pretrained_ffn_state_dict):
        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.ff_layer = SOMELayer(d_model, some_config, num_tasks, pretrained_ffn_state_dict)

    def forward(self, x, task_id, temperature=1.0):
        mask = torch.triu(torch.ones(x.size(1), x.size(1), device=x.device) * float('-inf'), diagonal=1)
        attn_output, _ = self.attention(x, x, x, attn_mask=mask, need_weights=False)
        x = self.norm1(x + attn_output)
        ff_output, queries, top_k_indices = self.ff_layer(x, task_id, temperature=temperature)
        x = self.norm2(ff_output)
        return x, queries, top_k_indices
```

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pos = 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(pos * div_term); pe[0, :,
1::2] = torch.cos(pos * div_term)
        self.register_buffer('pe', pe)
    def forward(self, x): return x + self.pe[:, :x.size(1)]

class SoMEHybridTransformer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        model_config, some_config = config['model'], config['some_layer']
        self.num_tasks = len(config['tasks'])
        self.embedding = nn.Embedding(model_config['vocab_size'], model_config['d_model'])
        self.pos_encoder = PositionalEncoding(model_config['d_model'], model_config['seq_len'])

        print("--- Initializing Multi-Task SoME-Hybrid Model ---")
        # --- CORRECTED PRINT STATEMENT ---
        print(f"Loading professional model '{config['pro_model_name']}' to source experts.")
        pro_model = AutoModelForCausalLM.from_pretrained(config['pro_model_name'])

        pretrained_state_dicts = []
        for layer in pro_model.gpt_neox.layers:
            layer_weights = {'down': {'weight': layer.mlp.dense_h_to_4h.weight, 'bias':
layer.mlp.dense_h_to_4h.bias},
                             'up': {'weight': layer.mlp.dense_4h_to_h.weight, 'bias':
layer.mlp.dense_4h_to_h.bias}}
            pretrained_state_dicts.append(layer_weights)

        num_extracted_layers = len(pretrained_state_dicts)
        if num_extracted_layers != model_config['num_layers']:
            # --- CORRECTED ERROR MESSAGE ---
            raise ValueError(
                f"CRITICAL ERROR: Mismatch in number of layers. "
                f"Your config expects {model_config['num_layers']} layers, "
                f"but the pre-trained model '{config['pro_model_name']}' only has
{num_extracted_layers} layers. "
                f"Please update 'num_layers' in your config to match."
            )
```

```
        self.layers = nn.ModuleList([
            TransformerBlock(model_config['d_model'], model_config['num_heads'], some_config,
self.num_tasks, pretrained_state_dicts[i])
            for i 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, task_id, temperature=1.0):
        x = self.embedding(x) * math.sqrt(self.d_model)
        x = self.pos_encoder(x)
        all_queries, all_indices = [], []
        for layer in self.layers:
            x, queries, top_k_indices = layer(x, task_id, 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, task_id):
        for i, layer_block in enumerate(self.layers):
            layer_block.ff_layer.update_keys(all_queries[i].view(-1, self.d_model),
all_indices[i].view(-1, self.config['some_layer']['top_k']), task_id)

print("Multi-Task SoME model components defined and corrected.")

#
==========================================================================
======
# Cell 3: Data, Training, and Evaluation (Upgraded for Multi-Task)
#
==========================================================================
======

class MultiTaskDataset(Dataset):
    """A dataset that prepends a task-specific token to each example."""
    def __init__(self, tokenized_data, task_token_id, task_id):
        self.data = tokenized_data
        self.task_token_id = task_token_id
        self.task_id = task_id

    def __len__(self):
        return len(self.data)
```

```
    def __getitem__(self, idx):
        item = self.data[idx]
        original_ids = torch.tensor(item['input_ids'])

        # Prepend task token, remove last token to keep sequence length constant
        inputs = torch.cat([torch.tensor([self.task_token_id]), original_ids[:-1]])

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

def prepare_data(config):
    print("\n--- Part 2: Data Preparation & Configuration ---")
    tokenizer_path = "multitask_bpe_tokenizer.json"

    tasks = config['tasks']
    task_tokens = [task['token'] for task in tasks]

    if not os.path.exists(tokenizer_path):
        print("Training universal BPE tokenizer with special task tokens...")
        base_dataset = load_dataset("roneneldan/TinyStories", split="train", streaming=True)
        def get_training_corpus():
            for i, item in enumerate(base_dataset):
                if i >= 20000: break
                yield item['text']

        # Add task tokens to the special tokens list
        special_tokens = ["[UNK]", "[PAD]", "[EOS]"] + task_tokens
        tokenizer_model = Tokenizer(BPE(unk_token="[UNK]"))
        tokenizer_model.pre_tokenizer = Whitespace()
        trainer = BpeTrainer(special_tokens=special_tokens,
vocab_size=config['model']['vocab_size'])
        tokenizer_model.train_from_iterator(get_training_corpus(), trainer=trainer)
        tokenizer_model.save(tokenizer_path)
    else:
        print("Universal tokenizer already exists. Loading from file.")

    tokenizer = PreTrainedTokenizerFast(tokenizer_file=tokenizer_path)
    tokenizer.add_special_tokens({'pad_token': '[PAD]', 'eos_token': '[EOS]'})
    tokenizer.add_tokens(task_tokens, special_tokens=True)
    print(f"Tokenizer loaded with vocab size: {tokenizer.vocab_size} (including task tokens)")
```

```
    task_token_ids = {task['name']: tokenizer.convert_tokens_to_ids(task['token']) for task in
tasks}

    all_train_datasets = []
    all_val_datasets = []

    for i, task in enumerate(tasks):
        print(f"\nProcessing task '{task['name']}'...")
        dataset = load_dataset(task['dataset_name'])
        train_split = 'train' if 'train' in dataset else list(dataset.keys())[0]
        val_split = 'validation' if 'validation' in dataset else 'test' if 'test' in dataset else train_split

        train_subset = dataset[train_split].select(range(min(len(dataset[train_split]),
task['train_size'])))
        val_subset = dataset[val_split].select(range(min(len(dataset[val_split]), task['val_size'])))

        content_column = task['content_column']
        def tokenize_function(examples):
            # No EOS token here, as we need space for the task token
            return tokenizer(examples[content_column], truncation=True, padding="max_length",
max_length=config['model']['seq_len'])

        tokenized_train = train_subset.map(tokenize_function, batched=True,
remove_columns=train_subset.column_names, num_proc=os.cpu_count())
        tokenized_val = val_subset.map(tokenize_function, batched=True,
remove_columns=val_subset.column_names, num_proc=os.cpu_count())

        tokenized_train.set_format(type='torch', columns=['input_ids'])
        tokenized_val.set_format(type='torch', columns=['input_ids'])

        all_train_datasets.append(MultiTaskDataset(tokenized_train, task_token_ids[task['name']],
i))
        all_val_datasets.append(MultiTaskDataset(tokenized_val, task_token_ids[task['name']], i))

    train_dataset = ConcatDataset(all_train_datasets)
    validation_dataset = ConcatDataset(all_val_datasets)

    num_workers = max(2, os.cpu_count() // 2 if os.cpu_count() else 2)
    train_loader = DataLoader(train_dataset, batch_size=config['training']['batch_size'],
shuffle=True, drop_last=True, num_workers=num_workers, pin_memory=True)
    validation_loader = DataLoader(validation_dataset, batch_size=config['training']['batch_size'],
drop_last=True, num_workers=num_workers, pin_memory=True)

    print(f"\nCombined train dataset size: {len(train_dataset)}")
```

```
    print(f"Combined validation dataset size: {len(validation_dataset)}")
    return train_loader, validation_loader, tokenizer

# --- The rest of the functions are updated to handle the task_id ---
def train_epoch(model, dataloader, optimizer, criterion, scheduler, current_temp,
tokenizer_vocab_size):
    model.train()
    total_loss = 0
    scaler = torch.cuda.amp.GradScaler(enabled=torch.cuda.is_available())
    progress_bar = tqdm(dataloader, desc="Training", leave=False)

    for inputs, targets, task_ids in progress_bar:
        # Batches now contain task_ids
        inputs, targets, task_ids = inputs.to(device), targets.to(device), task_ids.to(device)

        # We assume all items in a batch are from the same task for simplicity
        # This is guaranteed by how ConcatDataset and DataLoader work without a custom
sampler
        task_id = task_ids[0].item()

        optimizer.zero_grad(set_to_none=True)
        with torch.cuda.amp.autocast(enabled=torch.cuda.is_available()):
            logits, queries, indices = model(inputs, task_id=task_id, temperature=current_temp)
            loss = criterion(logits.view(-1, tokenizer_vocab_size), targets.view(-1))

        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, task_id=task_id)

        total_loss += loss.item()
        progress_bar.set_postfix({'loss': f'{loss.item():.4f}', 'lr': f'{scheduler.get_last_lr()[0]:.1e}'})

    return total_loss / len(dataloader)

def evaluate_epoch(model, dataloader, criterion, tokenizer_vocab_size):
    model.eval()
    total_loss = 0
    progress_bar = tqdm(dataloader, desc="Evaluating", leave=False)

    with torch.no_grad():
        for inputs, targets, task_ids in progress_bar:
            inputs, targets, task_ids = inputs.to(device), targets.to(device), task_ids.to(device)
```

```
            task_id = task_ids[0].item()
            with torch.cuda.amp.autocast(enabled=torch.cuda.is_available()):
                logits, _, _ = model(inputs, task_id=task_id, 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)

# plot_and_save_metrics and other helpers can remain largely the same, but the Gini/Entropy
# would need to be logged per-task for a truly deep analysis. We omit that for now for simplicity.

print("Multi-Task data, training, and evaluation functions defined.")

#
==========================================================================
======
# Cell 4: Main Execution Function
#
==========================================================================
======
# The previous main() function can be used, but we'll simplify the logging for this run.

def main(config):
    print(f"\n--- Starting Multi-Task Experiment: {config['run_name']} ---")

    base_save_dir = '/content/drive/MyDrive/SoME_Experiments' if DRIVE_MOUNTED else
'/content/SoME_Experiments'
    run_save_dir = os.path.join(base_save_dir, config['run_name'])
    os.makedirs(run_save_dir, exist_ok=True)

    train_loader, val_loader, tokenizer = prepare_data(config)
    model = SoMEHybridTransformer(config).to(device)

    # ... (Standard training setup: optimizer, criterion, scheduler)
    optimizer = torch.optim.AdamW(model.parameters(), lr=config['training']['learning_rate'])
    criterion = nn.CrossEntropyLoss(ignore_index=-100)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=len(train_loader)
* config['training']['num_epochs'])

    best_val_loss = float('inf')
    model_save_path = os.path.join(run_save_dir, "final_multitask_model.pth")

    for epoch in range(config['training']['num_epochs']):
```

```
        print(f"\n--- Epoch {epoch+1}/{config['training']['num_epochs']} ---")
        train_loss = train_epoch(model, train_loader, optimizer, criterion, scheduler, 1.0,
tokenizer.vocab_size)
        val_loss = evaluate_epoch(model, val_loader, criterion, tokenizer.vocab_size)
        print(f"Epoch {epoch+1}: Train Loss={train_loss:.4f}, Val Loss={val_loss:.4f}, Val
PPL={math.exp(val_loss):.2f}")
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            torch.save(model.state_dict(), model_save_path)
            print(f"  -> New best model saved to {model_save_path}")

    print(f"\n--- Multi-Task Training Complete ---")
    print(f"Final Validation Perplexity: {math.exp(best_val_loss):.2f}")

#
==========================================================================
======
# Cell 5: The Multi-Task Grand Challenge Control Panel
#
==========================================================================
======

def get_multitask_config():
    """Generates the single configuration for our multi-task experiment."""
    config = {
        "run_name": "final_multitask_model",
        "pro_model_name": "EleutherAI/pythia-70m",
        "model": {
            "vocab_size": 8192 + 3, # 8192 + 3 special task tokens
            "d_model": 512, "num_layers": 6, "num_heads": 8, "seq_len": 768,
        },
        "tasks": [
            {
                "name": "tinystories", "token": "<tinystories>", "dataset_name":
"roneneldan/TinyStories",
                "content_column": "text", "train_size": 30000, "val_size": 3000
            },
            {
                "name": "tinytextbooks", "token": "<tinytextbooks>", "dataset_name":
"nampdn-ai/tiny-textbooks",
                "content_column": "text", "train_size": 30000, "val_size": 3000
            },
            {
                "name": "tinycodes", "token": "<tinycodes>", "dataset_name": "nampdn-ai/tiny-codes",
```

```
            "content_column": "response", "train_size": 30000, "val_size": 3000
        },
    ],
    "training": {
        "num_epochs": 4, # Train for longer on the combined dataset
        "learning_rate": 5e-4, "batch_size": 24,
    },
    "some_layer": {
        "num_experts": 128, "d_ffn": 2048, "top_k": 8, "init_method": "sparse",
        "alpha": 0.015, "beta": 0.001, "delta": 0.001,
        "theta_percentile": 0.05, "warmup_steps": 400, "ema_decay": 0.995,
    }
}
return config

# --- Execute the Multi-Task Experiment ---
config = get_multitask_config()
main(config)

#
================================================================================
======
# Cell 6: Interactive Playground (Upgraded for Multi-Task)
#
================================================================================
======

# ... (This cell is largely the same, but the generate function and UI are updated)
# You would need to update the loading path and the generate function to handle the task token.
# Here is the complete, correct cell.

import ipywidgets as widgets
from IPython.display import display, clear_output
import warnings

warnings.filterwarnings('ignore', category=UserWarning)

print("--- Preparing Multi-Task Interactive Playground ---")

final_model_path =
"/content/drive/MyDrive/SoME_Experiments/final_multitask_model/final_multitask_model.pth"
tokenizer_path = "multitask_bpe_tokenizer.json"
analysis_model = None
```

```
if os.path.exists(final_model_path) and os.path.exists(tokenizer_path):
    print("Loading final multi-task model and tokenizer...")
    try:
        final_config = get_multitask_config()
        analysis_model = SoMEHybridTransformer(final_config).to(device)
        analysis_model.load_state_dict(torch.load(final_model_path))
        analysis_model.eval()
        tokenizer = PreTrainedTokenizerFast(tokenizer_file=tokenizer_path)
        tokenizer.add_special_tokens({'pad_token': '[PAD]'}) # EOS is part of vocab
        print("\n✅ Model and tokenizer loaded successfully!")
    except Exception as e:
        print(f"\n❌ An error occurred during model loading: {e}")
else:
    print(f"\n❌ ERROR: Could not find the final trained model or tokenizer.")

@torch.no_grad()
def generate_text_multitask(model, tokenizer, prompt, max_new_tokens=100,
temperature=0.7):
    model.eval()

    # Identify task from prompt
    task_id = -1
    for i, task in enumerate(model.config['tasks']):
        if prompt.strip().startswith(task['token']):
            task_id = i
            break

    if task_id == -1:
        print("ERROR: Prompt must start with a valid task token, e.g., '<tinystories>'.")
        return

    input_ids = tokenizer.encode(prompt, return_tensors="pt").to(device)
    generated_ids = input_ids

    print(f"\nTask: '{model.config['tasks'][task_id]['name']}'")
    print(f"Prompt: '{prompt}'")
    print("Model Output: ", end='', flush=True)

    for _ in range(max_new_tokens):
        with torch.cuda.amp.autocast(enabled=torch.cuda.is_available()):
            logits, _, _ = model(generated_ids, task_id=task_id)
            next_token_logits = logits[:, -1, :] / temperature

        probs = F.softmax(next_token_logits, dim=-1)
```

```
        next_token_id = torch.multinomial(probs, num_samples=1)

        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 ---")

if analysis_model:
    prompt_area = widgets.Textarea(
        value='<tinycodes>def factorial(n):\n    """Calculates the factorial of a number.""",
        placeholder="Start with a task token, e.g., '<tinystories> Once upon a time...'",
        description='Prompt:', layout=widgets.Layout(width='90%', height='100px')
    )
    generate_button = widgets.Button(description='Generate Text', button_style='success')
    output_area = widgets.Output()

    def on_button_clicked(b):
        output_area.clear_output()
        with output_area:
            generate_text_multitask(analysis_model, tokenizer, prompt_area.value)

    generate_button.on_click(on_button_clicked)
    print("\n\n==================== Multi-Task SoME Playground
====================")
    display(prompt_area, generate_button, output_area)
```