

SoME v4.6 Residual Grounding + Dropout Code & Results

```
#  
=====  
=====  
# SoME v5: Utilization Dropout Edition  
# Authors: Focus Labs & Norm  
# Environment: Google Colab (A100 Single GPU Optimized)  
#  
=====  
=====  
  
# --- Part 1: Dependencies & Setup ---  
import subprocess  
import sys  
  
def install_dependencies():  
    packages = ["torch", "datasets", "transformers", "huggingface_hub", "tokenizers", "matplotlib",  
"scikit-learn"]  
    subprocess.check_call([sys.executable, "-m", "pip", "install"] + packages + ["-q"])  
  
try:  
    import torch  
except ImportError:  
    install_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  
from sklearn.manifold import TSNE  
import copy  
from tqdm import tqdm  
import math  
import os  
import numpy as np  
import matplotlib.pyplot as plt  
import time
```

```

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

# A100 Optimization: Enable TF32
if torch.cuda.is_available() and torch.cuda.get_device_capability(0)[0] >= 8:
    print("A100/H100 GPU detected. Enabling TF32.")
    torch.set_float32_matmul_precision('high')

torch.backends.cudnn.benchmark = True

#
=====
=====

# --- Part 2: The SoME v5 Architecture (Grounded + Utilization Dropout) ---
#
=====

=====

class Expert(nn.Module):
    """
    An independent, specialized sub-network.
    In SoME, these are initialized randomly and frozen (reservoir computing).
    """

    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)

    # Scientific Rigor: Explicit Initialization Control
    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':
        pass # Use PyTorch Kaiming/Xavier defaults
    else:
        raise ValueError(f"Unknown initialization method: {init_method}")

    # Bias initialization for stability

```

```

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):
    """
    Self-Organizing Mixture of Experts Layer (v5).
    Features:
    1. Knowledge Gravity Updates (Alpha, Beta, Delta)
    2. MLP Router
    3. Router Grounding (from v4)
    4. Utilization Dropout (NEW in v5)
    """

    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']

        # Grounding Hyperparameters (v4 Intervention)
        self.grounding_coeff = some_config.get('grounding_coeff', 0.0)

        # Utilization Dropout Hyperparameters (v5 Intervention)
        # Rate: Probability of applying dropout in a forward pass (e.g., 0.2)
        self.util_dropout_rate = some_config.get('util_dropout_rate', 0.0)
        # Percentile: Usage threshold to define "High Usage" experts (e.g., 0.9 = top 10%)
        self.util_dropout_percentile = some_config.get('util_dropout_percentile', 0.9)

        # Heuristic Update Parameters
        self.alpha = some_config['alpha']
        self.beta = some_config['beta']
        self.delta = some_config['delta']

        # Dynamics 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})

    # --- The MLP Router ---

```

```

hidden_dim = d_model * 2
self.query_network = nn.Sequential(
    nn.Linear(d_model, hidden_dim),
    nn.GELU(),
    nn.Linear(hidden_dim, d_model)
)

# --- The Dynamic Key Store ---
keys = torch.randn(self.num_experts, d_model)
self.register_buffer("key_store", F.normalize(keys, p=2, dim=-1))

# State Tracking
self.register_buffer("usage_count", torch.zeros(self.num_experts))
self.register_buffer("steps", torch.tensor([0], dtype=torch.long))

# --- The Frozen Expert Pool ---
self.experts = nn.ModuleList([
    Expert(d_model, self.d_ffn, init_method=some_config['init_method'])
    for _ in range(self.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, grounding_input=None):
    batch_size, seq_len, _ = x.shape
    x_flat = x.view(-1, self.d_model)

# --- v4 Intervention: Router Grounding ---
if grounding_input is not None and self.grounding_coeff > 0:
    grounding_flat = grounding_input.view(-1, self.d_model)
    router_input = x_flat + (self.grounding_coeff * grounding_flat)
else:
    router_input = x_flat

# Generate Queries
queries_raw = self.query_network(router_input)
queries = F.normalize(queries_raw, p=2, dim=-1)

```

```

# Cosine Similarity Search
scores = torch.matmul(queries, self.key_store.t())

# --- v5 Intervention: Utilization Dropout ---
# Only apply during training to force exploration
if self.training and self.util_dropout_rate > 0:
    # Roll the dice
    if torch.rand(1).item() < self.util_dropout_rate:
        # Calculate the usage threshold (e.g., usage of the 90th percentile expert)
        limit = torch.quantile(self.usage_count, self.util_dropout_percentile)

        # Identify experts exceeding this limit ("Entrenched Generalists")
        mask_indices = torch.nonzero(self.usage_count > limit).squeeze()

        # Safety check: Don't mask everyone if everyone is equal
        if mask_indices.numel() > 0 and mask_indices.numel() < self.num_experts:
            # Set their scores to -inf so they cannot be selected
            scores[:, mask_indices] = float('-inf')

# Top-K Selection
top_k_scores, top_k_indices = torch.topk(scores, self.top_k, dim=-1)

# Softmax Gating
gating_weights = F.softmax(top_k_scores / temperature, dim=-1)

# --- Efficient Dispatch & Combiner ---
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):
    """
    The 'Knowledge Gravity' Update Mechanism.
    """

    self.steps += 1

    # 1. Update Usage Statistics (EMA)
    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())

    # 2. Alpha: Attraction (Query Pull)
    if self.ablation_flags.get('use_alpha', True):
        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)

    # 3. Beta: Clustering (Peer Pull)
    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 = self.key_store[indices_i]
        keys_j = self.key_store[indices_j]

        inertia_i = (1.0 + self.usage_count[indices_i]).unsqueeze(-1)
        inertia_j = (1.0 + self.usage_count[indices_j]).unsqueeze(-1)

        beta_effective = self.beta / torch.min(inertia_i, inertia_j)

        update_vec_i = beta_effective * (keys_j - keys_i)
        update_vec_j = beta_effective * (keys_i - keys_j)

        self.key_store.index_add_(0, indices_i, update_vec_i)

```

```

        self.key_store.index_add_(0, indices_j, update_vec_j)

    self.key_store.data = F.normalize(self.key_store.data, p=2, dim=-1)

# 4. Delta: 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
        self.key_store[low_usage_mask] *= (1.0 - self.delta)

class SOMETwitterBlock(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, temperature=1.0, causal_mask=None, grounding_input=None):
        attn_output, _ = self.attention(x, x, x, attn_mask=causal_mask, is_causal=True)
        x = self.norm1(x + attn_output)

        some_output, queries, top_k_indices = self.some_layer(
            x, temperature=temperature, grounding_input=grounding_input
        )
        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(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.model_config = model_config
        self.d_model = model_config['d_model']

        self.embedding = nn.Embedding(model_config['vocab_size'], self.d_model)
        self.pos_encoder = PositionalEncoding(self.d_model, model_config['seq_len'])

        self.layers = nn.ModuleList([
            SOMETransformerBlock(self.d_model, model_config['num_heads'], some_config)
            for _ in range(model_config['num_layers'])
        ])

        self.fc_out = nn.Linear(self.d_model, model_config['vocab_size'])

        mask = torch.triu(torch.ones(model_config['seq_len'], model_config['seq_len']) * float('-inf'),
                          diagonal=1)
        self.register_buffer('causal_mask', mask)

        self.grounding_start_layer = some_config.get('grounding_start_layer', 0)

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

        grounding_signal = x.clone()

        all_queries, all_indices = [], []

        for i, layer in enumerate(self.layers):
            current_grounding = grounding_signal if i >= self.grounding_start_layer else None

            x, queries, top_k_indices = layer(
                x,
                temperature=temperature,
                causal_mask=self.causal_mask[:x.size(1), :x.size(1)],
                grounding_input=current_grounding
            )
            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 3: Data & Utilities ---
# =====#
=====#
=====

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 = item['input_ids'].clone().detach()
        targets = inputs.clone()
        targets[:-1] = inputs[1:]
        targets[-1] = -100
        return inputs, targets

def prepare_data(config):
    print("\n--- Data Preparation ---")
    tokenizer_path = "tinystories-tokenizer-v5.json"

    if not os.path.exists(tokenizer_path):
        print("Training custom tokenizer...")
    dataset = load_dataset("roneneldan/TinyStories", split="train", streaming=False)
    def batch_iterator(batch_size=1000):
        for i in range(0, len(dataset), batch_size):
            yield dataset[i : i + batch_size]["text"]

    tokenizer = Tokenizer(BPE(unk_token="[UNK]"))
    tokenizer.pre_tokenizer = Whitespace()

```

```

trainer = BpeTrainer(special_tokens=["[UNK]", "[PAD]", "[EOS]"],
vocab_size=config['model']['vocab_size'])
    tokenizer.train_from_iterator(batch_iterator(), trainer=trainer)
    tokenizer.save(tokenizer_path)
else:
    print("Loading existing tokenizer.")
    tokenizer = PreTrainedTokenizerFast(tokenizer_file=tokenizer_path)
    tokenizer.add_special_tokens({'pad_token': '[PAD]', 'eos_token': '[EOS]'})

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")

print(f"Tokenizing {len(train_subset)} training examples...")
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())

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

train_loader = DataLoader(LanguageModelDataset(tokenized_train),
batch_size=config['data']['batch_size'], shuffle=True,
drop_last=True, num_workers=2, pin_memory=True)
val_loader = DataLoader(LanguageModelDataset(tokenized_val),
batch_size=config['data']['batch_size'], drop_last=True,
num_workers=2, pin_memory=True)

return train_loader, val_loader, tokenizer

#
=====
=====
# --- Part 4: Training Engine ---
#
=====
```

```

def calculate_metrics(usage_counts):
    counts = usage_counts.cpu().to(torch.float32).numpy()
    if np.sum(counts) == 0: return 0.0, 0.0

    sorted_counts = np.sort(counts)
    n = len(counts)
    index = np.arange(1, n + 1)
    gini = (np.sum((2 * index - n - 1) * sorted_counts)) / (n * np.sum(sorted_counts))

    probs = counts / np.sum(counts)
    probs = probs[probs > 0]
    entropy = -np.sum(probs * np.log2(probs))

    return gini, entropy

def train_epoch(model, dataloader, optimizer, criterion, scheduler, current_temp, vocab_size):
    model.train()
    total_loss = 0
    scaler = torch.amp.GradScaler("cuda")

    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, 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, vocab_size):
    model.eval()
    total_loss = 0
    with torch.no_grad():
        for inputs, targets in dataloader:
            inputs, targets = inputs.to(device, non_blocking=True), targets.to(device,
non_blocking=True)
            with torch.amp.autocast("cuda"):
                logits, _, _ = model(inputs, temperature=0.5)
                loss = criterion(logits.view(-1, vocab_size), targets.view(-1))
                total_loss += loss.item()
    return total_loss / len(dataloader)

#
=====
=====

# --- Part 5: Visualization & Analysis ---
#
=====

=====

def visualize_galaxy(model, layer_idx, run_name):
    """Generates t-SNE plot for a specific layer's expert keys."""
    try:
        layer = model.layers[layer_idx].some_layer
        keys = layer.key_store.detach().cpu().numpy()
        usage = layer.usage_count.detach().cpu().numpy()
    except IndexError:
        print(f"Layer {layer_idx} out of bounds.")
        return

    print(f"Visualizing Layer {layer_idx} Galaxy...")

    tsne = TSNE(n_components=2, perplexity=30, init='pca', learning_rate='auto',
random_state=42)
    keys_2d = tsne.fit_transform(keys)

    if usage.sum() > 0:
        norm_usage = usage / usage.sum()
    else:
        norm_usage = np.zeros_like(usage)

    plt.figure(figsize=(12, 10))
    sc = plt.scatter(keys_2d[:, 0], keys_2d[:, 1], c=usage,

```

```

        s=30 + norm_usage * 5000, cmap='viridis', alpha=0.8, edgecolors='k')
plt.colorbar(sc, label='Expert Usage (EMA)')
plt.title(f"Knowledge Galaxy: Layer {layer_idx} (Grounded + Dropout)\n{run_name}")
plt.xlabel("t-SNE Dim 1")
plt.ylabel("t-SNE Dim 2")

plt.savefig(f"galaxy_layer_{layer_idx}_{run_name}.png", dpi=300)
plt.close()
print(f"Saved galaxy_layer_{layer_idx}_{run_name}.png")

def plot_training_curves(train_losses, val_losses, run_name):
    plt.figure(figsize=(10, 6))
    epochs = range(1, len(train_losses) + 1)
    plt.plot(epochs, train_losses, 'b-o', label='Training Loss')
    plt.plot(epochs, val_losses, 'r-o', label='Validation Loss')
    plt.title(f"Training Dynamics: {run_name}")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.grid(True)
    plt.savefig(f"loss_curve_{run_name}.png")
    plt.close()

#
=====
=====
# --- Part 6: Main Execution Block ---
#
=====

def main():
    # --- Experiment Configuration ---
    config = {
        "run_name": "v5_Utilization_Dropout_Experiment",
        "data": {
            "train_subset_size": 20000,
            "val_subset_size": 2000,
            "batch_size": 32
        },
        "model": {
            "vocab_size": 8192,
            "d_model": 512,
            "num_heads": 8,

```

```

        "num_layers": 12,
        "seq_len": 768
    },
    "some_layer": {
        "num_experts": 64,
        "d_ffn": 1024,
        "top_k": 4,
        "init_method": "default",

        # --- v5 INTERVENTION: UTILIZATION DROPOUT ---
        "util_dropout_rate": 0.2,      # 20% of batches will trigger dropout
        "util_dropout_percentile": 0.9, # Mask the top 10% most used experts

        # --- v4 INTERVENTION: ROUTER GROUNDING ---
        "grounding_coeff": 0.1,
        "grounding_start_layer": 6,

        # Dynamics
        "alpha": 0.01,
        "beta": 0.005,
        "delta": 0.001,
        "theta_percentile": 0.05,
        "warmup_steps": 400,
        "ema_decay": 0.99,
    },
    "training": {
        "num_epochs": 5,
        "learning_rate": 6e-4,
        "training_temp": 0.8
    }
}

print(f"Starting Experiment: {config['run_name']}")

train_loader, val_loader, tokenizer = prepare_data(config)

model = SOMETransformer(config['model'], config['some_layer']).to(device)

if hasattr(torch, 'compile'):
    print("Compiling model...")
    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)
total_steps = len(train_loader) * config['training']['num_epochs']
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=total_steps)
criterion = nn.CrossEntropyLoss(ignore_index=-100)

train_losses, val_losses = [], []
best_val_loss = float('inf')

print(f"\n--- Training for {config['training']['num_epochs']} Epochs ---")
for epoch in range(config['training']['num_epochs']):
    print(f"\nEpoch {epoch+1}")

    t_loss = train_epoch(model, train_loader, optimizer, criterion, scheduler,
                         config['training']['training_temp'], tokenizer.vocab_size)

    v_loss = evaluate_epoch(model, val_loader, criterion, tokenizer.vocab_size)

    train_losses.append(t_loss)
    val_losses.append(v_loss)

    orig_model = model._orig_mod if hasattr(model, '_orig_mod') else model
    deep_layer_idx = config['model']['num_layers'] - 2
    gini, entropy =
    calculate_metrics(orig_model.layers[deep_layer_idx].some_layer.usage_count)

    print(f" Train Loss: {t_loss:.4f} | Val Loss: {v_loss:.4f} | Val PPL: {math.exp(v_loss):.2f}")
    print(f" Layer {deep_layer_idx} Metrics -> Gini: {gini:.3f} | Entropy: {entropy:.3f}")

    if v_loss < best_val_loss:
        best_val_loss = v_loss
        torch.save(orig_model.state_dict(), f"best_{config['run_name']}.pth")

plot_training_curves(train_losses, val_losses, config['run_name'])

viz_layers = [0, config['model']['num_layers']//2, config['model']['num_layers']-1]
for idx in viz_layers:
    visualize_galaxy(orig_model, idx, config['run_name'])

if __name__ == "__main__":
    main()

```

Using device: cuda
A100/H100 GPU detected. Enabling TF32.
Starting Experiment: v5_Utilization_Dropout_Experiment

--- Data Preparation ---

Loading existing tokenizer.

Tokenizing 20000 training examples...

Map (num_proc=12): 100%

20000/20000 [00:03<00:00, 6666.79 examples/s]

Map (num_proc=12): 100%

2000/2000 [00:00<00:00, 3746.22 examples/s]

Compiling model...

--- Training for 5 Epochs ---

Epoch 1

Training: 0% | 0/625 [00:00<?, ?it/s] W0210 19:27:17.414000 5188

torch/_dynamo/variables/tensor.py:1048] [0/0] Graph break from `Tensor.item()`, consider setting:

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0]

torch._dynamo.config.capture_scalar_outputs = True

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0] or:

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0] env

TORCHDYNAMO_CAPTURE_SCALAR_OUTPUTS=1

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0] to include these operations in the captured graph.

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0]

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0] Graph break: from user code at:

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0] File

"/tmp/ipython-input-1605295806.py", line 328, in forward

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0] x, queries, top_k_indices = layer(

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0] File

"/tmp/ipython-input-1605295806.py", line 276, in forward

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0] some_output, queries, top_k_indices = self.some_layer(

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0] File

"/tmp/ipython-input-1605295806.py", line 172, in forward

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0] if torch.rand(1).item() < self.util_dropout_rate:

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0]

W0210 19:27:17.414000 5188 torch/_dynamo/variables/tensor.py:1048] [0/0]

/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(
```

Train Loss: 1.4824 | Val Loss: 1.1015 | Val PPL: 3.01
Layer 10 Metrics -> Gini: 0.749 | Entropy: 4.318

Epoch 2

Train Loss: 1.1621 | Val Loss: 0.9663 | Val PPL: 2.63
Layer 10 Metrics -> Gini: 0.729 | Entropy: 4.354

Epoch 3

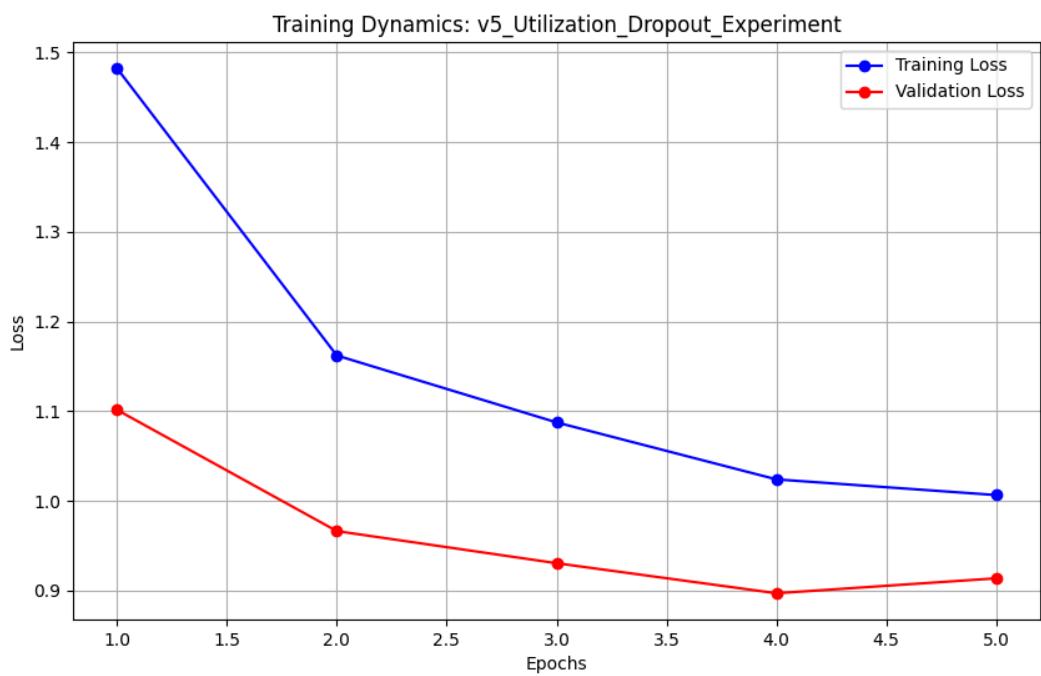
Train Loss: 1.0875 | Val Loss: 0.9305 | Val PPL: 2.54
Layer 10 Metrics -> Gini: 0.731 | Entropy: 4.313

Epoch 4

Train Loss: 1.0239 | Val Loss: 0.8970 | Val PPL: 2.45
Layer 10 Metrics -> Gini: 0.714 | Entropy: 4.526

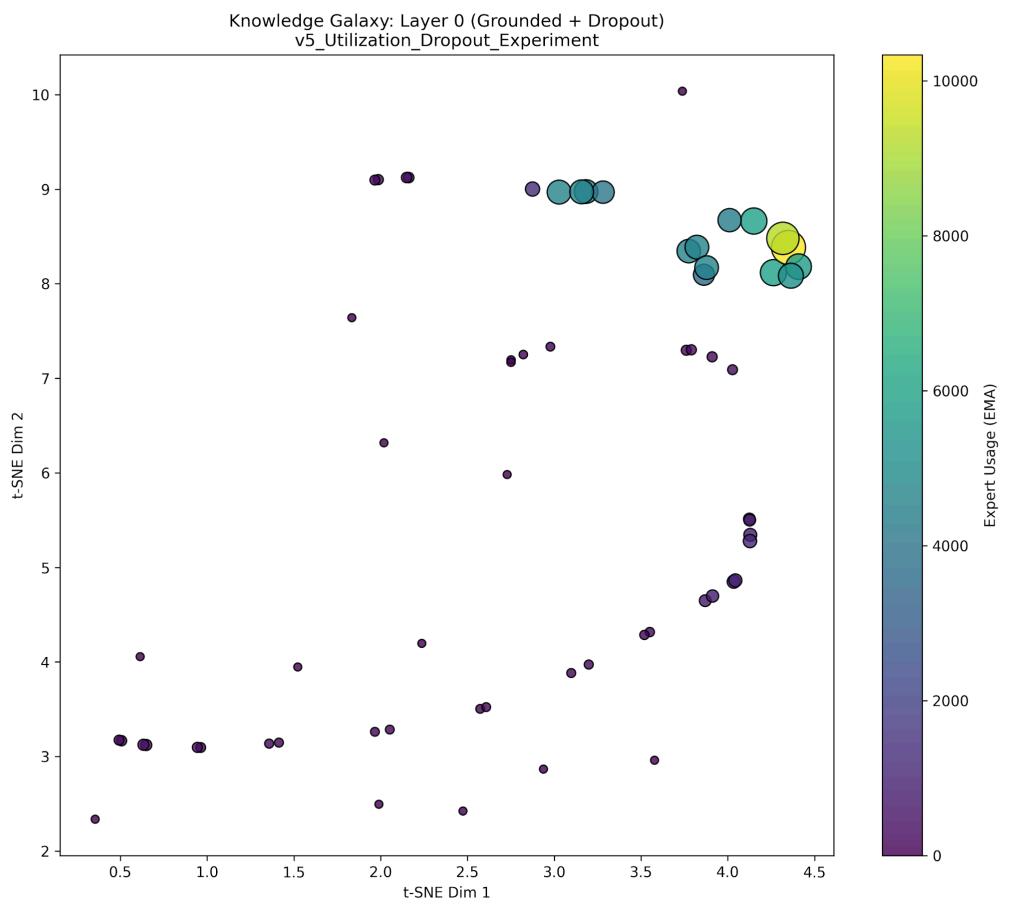
Epoch 5

Train Loss: 1.0064 | Val Loss: 0.9136 | Val PPL: 2.49
Layer 10 Metrics -> Gini: 0.698 | Entropy: 4.642



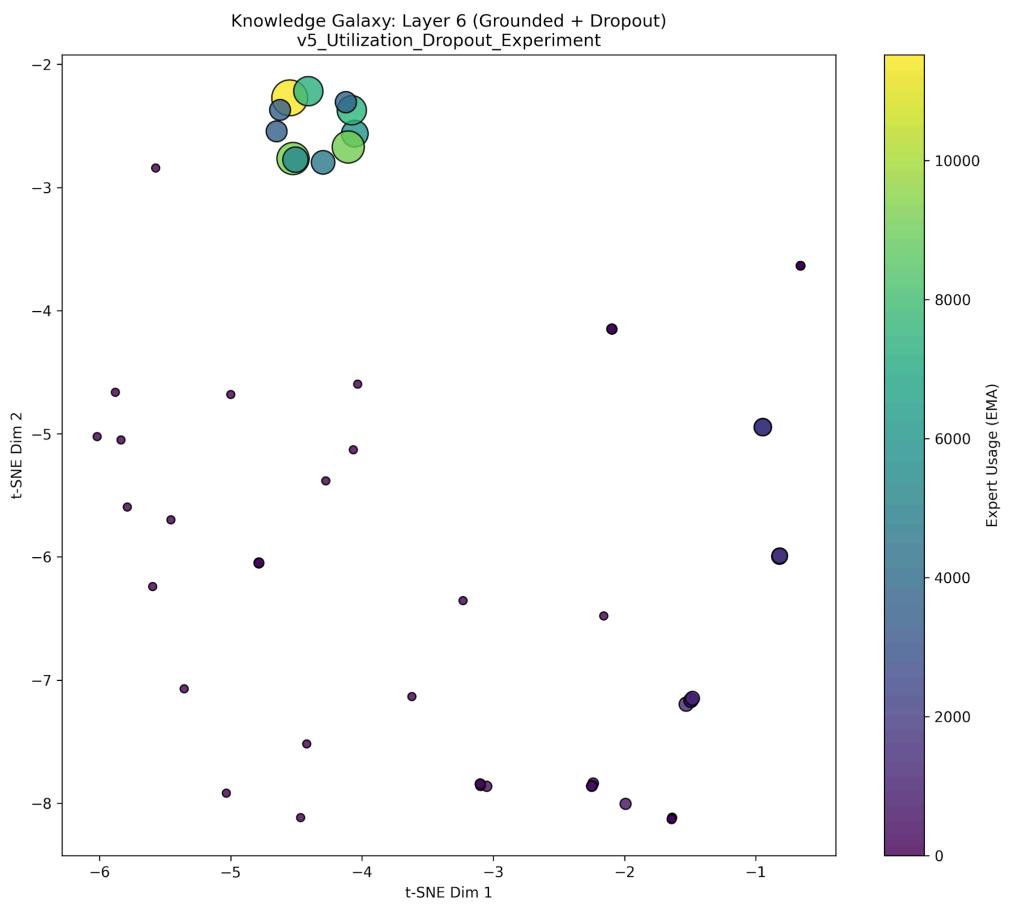
Visualizing Layer 0 Galaxy...

Saved galaxy_layer_0_v5_Utilization_Dropout_Experiment.png



Visualizing Layer 6 Galaxy...

Saved galaxy_layer_6_v5_Utilization_Dropout_Experiment.png



Visualizing Layer 11 Galaxy...

Saved galaxy_layer_11_v5_Utilization_Dropout_Experiment.png

