SoME Testing

v2 (SoME MVP v2)
- Core: d_model=256, num_layers=4, num_heads=4
- MoE: num_experts=64, top_k=4, d_ffn=1024, alpha=0.01, beta=0.001, delta=0.001, theta=200
- Data: seq_len=256, batch_size=64, vocab_size=8192
- Train: epochs=3, lr=8e-4, AdamW (wd=0.1), AMP + torch.compile
- Size: Total ~72.80M, Trainable ~5.52M
- Result (last epoch): val loss 0.3454, ppl 1.41

v3 (SoME MVP v3)
- Core: d_model=384, num_layers=8, num_heads=6
- MoE: num_experts=32, top_k=4, d_ffn=1024, alpha=0.01, beta=0.001, delta=0.001, theta=200
- Data: seq_len=256, batch_size=64, vocab_size=8192
- Train: epochs=2, lr=8e-4, AdamW (wd=0.1), AMP + torch.compile
- Size: Total ~213.91M, Trainable ~12.23M
- Result (epoch 2): val loss 0.1017, ppl 1.11

v4 (SoME MVP v4)
- Core: d_model=384, num_layers=8, num_heads=6
- MoE: num_experts=64, top_k=2, d_ffn=1536, alpha=0.01, beta=0.001, delta=0.001, theta=200
- Data: seq_len=256, batch_size=512, vocab_size=8192
- Train: epochs=1, lr=8e-4, AdamW (wd=0.1), AMP + torch.compile
- Size: Total ~617.19M, Trainable ~12.23M
- Result (epoch 1): Val Loss = 1.9435, ppl 6.98

V5
- Core: D_MODEL = 384, NUM_HEADS= 6, NUM_LAYERS = 8
- MoE: num_experts: 32, d_ffn: 1024, top_k: 4, theta: 200
- Data: SEQ_LEN = 256, BATCH_SIZE = 64, VOCAB_SIZE = 8192
- Train: Epoch 2: lr=8e-4, AdamW (wd=0.1), AMP + torch.compile
- Size: Total parameters: 213.91M, Trainable parameters: 12.23M
- Results: (Epoch 2) Train Loss = 0.2041, Val Loss = 0.1017, Val Perplexity = 1.11

V6 (upcoming)
- Core: d_model=384, num_layers=10, num_heads=8
- MoE: num_experts=128, top_k=4, d_ffn=1536, alpha=0.01, beta=0.001, delta=0.001, theta=200
- Data: seq_len=256, batch_size=256, vocab_size=8192
- Train: epochs=3, lr=8e-4, AdamW (wd=0.1), AMP + torch.compile

- Size: Total parameters: 1526.11M, Trainable parameters: 13.71M
- Results: Train Loss = 2.3146, Val Loss = 2.2723, Val Perplexity = 9.70

What's Next:

1. Use the Full Dataset: Train the model on the entire TinyStories dataset for a more extended period.
2. Analyze Expert Specialization and Dynamics: The core hypothesis of SoME is that experts will self-organize into "knowledge galaxies." Now is the time to verify this.
   a. Visualize the Key Store: After training, extract the final key_store tensor from one of the SOMELayers. Use dimensionality reduction techniques like t-SNE or PCA to plot the 32 expert keys in a 2D space. Are there distinct clusters?
   b. Track Key Movement: Log the positions of the keys at different stages of training (e.g., after epoch 1 and epoch 2). Visualizing this can show how the "knowledge galaxies" form over time.
   c. Analyze Expert Usage: Plot a histogram of the usage_count buffer.
      i. Is the load balanced? A relatively even distribution is a good sign.
      ii. Are there "generalist" experts? A few experts with very high usage counts could be acting as stable "galactic centers," as your theory predicts.
      iii. Are any experts being pruned? Check if any experts consistently fall below the theta threshold and are being decayed.

Ablation Studies

A. Fix backbone (use v3 core), vary MoE

- v3 core + 32e, k=4 (current) → baseline
- v3 core + 64e, k=4 → tests if v3 still wins when experts increase (watch for renewed imbalance).
- v3 core + 32e, k=8 → holds experts constant and increases mixture breadth per token; often improves ppl if capacity allows.
- Readouts: per-layer gate entropy, load Gini, % dropped tokens, validation ppl.

B. Fix experts (use 32e, k=4), vary backbone

- Shrink to v2 core (4×256) to confirm the bulk of the improvement was core capacity.
- Middle ground (6×320) to find a sweet spot for compute vs. ppl.
- Readouts: ppl vs. total params curve; plot loss vs. trained tokens to see sample-efficiency.

C. Isolate SOME’s in-inference adaptation
Run two modes on the same trained checkpoint:

- Frozen keys (disable Query/Peer/Repulsive updates) vs. SOME active (current α/β/δ/θ).

- Evaluate under a short domain shift (TinyStories → out-of-domain subset) and report ppl over time and post-shift validation. This directly shows the benefit of address-only plasticity.

D. Stress-test routing stability
- Key norm control: L2-normalize keys after each update; add a learnable temperature τ in routing softmax and target a gate-entropy band (e.g., 1.5–2.5 nats).
- Clip Δk (e.g., global norm or per-step cap) and consider a small EMA over keys to smooth updates.
- Usage-aware α floor: don't let α decay below ~0.005 so cold experts can still adapt.

What each knob does

1) Backbone capacity (dense transformer): These change the shared capacity (applies to all tokens).
1. d_model (hidden size)
    - Quality: ↑↑ (usually the strongest single dense knob).
    - Compute per token: ↑ (attention, MLP scale with ~d^2).
    - Memory: ↑ (activations, params/opt states).
    - Range: 320–448 on A100-80G for fast runs.
    - Tip: Increase d before layers if your model is shallow (<8).
2. num_layers (depth)
    - Quality: ↑↑ (great bang-per-FLOP at this scale).
    - Compute per token: ↑ linearly with layers.
    - Memory: ↑ (activations).
    - Range: 6–10 is a good sweet spot for sub-hour runs.
3. num_heads
    - Quality: ↑ (helps mixing, but smaller effect than d/layers).
    - Compute/Memory: mild ↑ (within same d_model).
    - Range: keep divisors of d_model; 6–8 for d=384–448.

2) MoE knobs (capacity vs compute): These control sparse capacity and routing compute.
1. num_experts
    - Quality: ↑ (more niches); diminishing returns if router underpowered.
    - Compute per token: ~neutral if top_k fixed; memory ↑ for expert weights (but frozen experts don't add optimizer state).
    - Range: 24–64. For speed, 32 is a sweet spot.
2. top_k
    - Quality: ↑ with larger k (more expert collaboration).
    - Compute per token: ↑ linearly (k extra FFNs).
    - Memory: small ↑.
    - Rule of thumb: If you halve k (4→2), increase d_ffn a bit (e.g., 1024→1536) or add LoRA to recover quality at much lower cost.
3. d_ffn (expert FFN width)

    - Quality: ↑ with larger FFN.
    - Compute per token: ↑ (MLP dominates).
    - Tandem with top_k: tune as a product: effective capacity ≈ k×d_ffn.
4. Experts trainability (frozen vs LoRA vs full)
    - Frozen: fastest, smallest optimizer state; rely on routing to specialize.
    - LoRA on experts (e.g., r=4–8): big ROI on quality, tiny overhead.
    - Full train: best quality but heavy optimizer/memory/time.

3) Router & self-organization (your special sauce): Affects specialization, stability, and some overhead.

1. Query network (depth/width; e.g., 1-layer linear vs 2-layer MLP d→h→d)
    - Quality: ↑ with 2-layer (GELU), esp. with more experts.
    - Compute: slight ↑. Worth it.
2. Self-org hyperparams: alpha (attraction), beta (repulsion), delta (decay), theta (low-usage threshold)
    - Quality/Stability:
        i. Higher β early spreads experts (prevents collapse).
        ii. Moderate α consolidates niches.
        iii. δ/θ push underused experts to explore.
    - Best practice: β-anneal (e.g., 3e-3 → 5e-4 over training).
3. Key update cadence
    - Every step vs every other step / token subsampling (e.g., 50%):
        i. Speed: improves 10–20% with minimal quality hit.
        ii. Use subsampling to buy throughput if time-bound.
4. Eval behavior
    - Never update keys during eval. (Keeps validation stationary and honest.) However the system is supposed to adapt in real-time, so hypothetically it is fair to allow update keys during eval.
5. Top-k warmup
    - Start with k=1 for 10–20% steps → ramp to final k.
    - Stability: reduces early routing chaos; often improves final loss.
6. Usage/balance logging
    - Track per-layer entropy and Gini of expert usage.
    - Symptoms: low entropy + high Gini → dominance/collapse; increase β or extend k=1 warmup.

4) Sequence length, batch, and accumulation: The main speed vs memory levers.

1. seq_len (train)
    - Quality: mild ↑ if task needs long context.
    - Compute: $O(L^2)$ for attention — expensive.
    - Strategy: Keep L modest (320–384) to go much faster; extrapolate at inference with RoPE/ALiBi.
2. Micro-batch size

- Speed: ↑↑—this is how you use your 80GB.
- Memory: ↑ (activations).
- Target: raise until ~60–70 GB used; then set grad_accum to hit your global_batch_tokens target (e.g., 120k–180k tokens/step).

3. Global batch tokens (micro_batch × seq_len × grad_accum)
   - Throughput: scales well until you saturate data pipeline.
   - Stability: don't overshoot LR/BS scaling; keep LR in check.

5) Optimization & schedule:

1. LR (peak) and schedule (warmup + cosine)
   - Quality/Stability: warmup 3–5% of steps is usually sweet; cosine is safe.
   - Note: If you surge batch a lot, LR may need a slight downscale.
2. Weight decay
   - 0.05–0.1 typical for small LM/TinyStories.
3. Aux balance losses
   - I'm using self-org (α/β/δ/θ). If you add a classic MoE load-balancing loss, use a tiny coeff to avoid fighting the self-org mechanism.

6) System throughput knobs (make the GPU sweat):

1. Precision: BF16 autocast.
2. FlashAttention / SDPA: must be on.
3. Fused AdamW (fused=True if available) or Apex FusedAdam.
4. torch.compile(mode="reduce-overhead") after ~50 warmup steps.
5. Disable activation checkpointing unless you truly need longer seq_len.
6. Data loader: pre-tokenize; num_workers=8–16, pin_memory=True, prefetch_factor=2–4, non_blocking=True.
7. Optional: CUDA Graphs if shapes are static (removes launch overhead).

7) Data & tokenizer:

1. Tokenizer/vocab (e.g., 8k)
   - Quality: can help; smaller vocab reduces softmax cost.
2. Packing/bucketing
   - Speed: ↑ by cutting padding; more real tokens per step.

8) Interactions that matter (rules of thumb):

1. top_k × d_ffn: keep their product roughly constant to hold MoE compute steady.
   - Example: k=4, FFN=1024 ≈ k=2, FFN=1536 (the latter is faster).
2. seq_len × micro_batch × grad_accum: fit VRAM via micro-batch first; avoid growing L unless needed.
3. Experts trainability: Frozen + LoRA often matches most of the gain of full finetune at a fraction of cost.
4. Router depth: A small 2-layer MLP router often pays for itself at E≥32.