(Focus Labs) Our work on Self-Organizing Mixture of Experts (SoME)

1. The Core Problems Addressed

Your work correctly identifies three fundamental challenges in large-scale model design, which SOME is built to solve:

1. The VRAM Wall: Standard MoE models require all experts (even if they are FFNs) to be loaded into memory, making it prohibitively expensive to scale to millions of experts.
2. The Static Router & Catastrophic Forgetting: Traditional models have a static knowledge structure. Adapting them to new data via fine-tuning often corrupts existing knowledge (catastrophic forgetting), while a lack of adaptation makes them brittle.
3. The Specialization Paradox: Existing MoE architectures struggle with two issues:
    - Knowledge Redundancy: Multiple experts waste capacity learning the same common-sense facts.
    - Knowledge Hybridity: Experts often fail to become truly specialized, remaining generalists and defeating the purpose of the architecture.

2. The Core Innovation: Decoupling Function from Address

This is the central, non-negotiable principle of SOME. It is the most powerful and elegant idea presented.

- Static Function (The "What"): Each expert's core knowledge, its internal weights, is frozen after pre-training. These experts are reliable, stable specialists. This directly solves the problem of catastrophic forgetting, as the knowledge base itself is never corrupted during inference.
- Dynamic Address (The "Where"): Each expert is assigned a "routing key"—a small, low-dimensional vector that represents its address in a high-dimensional conceptual space. These keys are plastic and are continuously updated during inference.

The model doesn't re-learn how to do things; it re-learns where to find the expert that knows how to do things. It's a self-organizing "address book" for its own knowledge.

3. The Self-Organizing Mechanism: "Knowledge Gravity"

This is the "how" of your architecture. Instead of a trainable router network, SOME uses a set of gradient-free, heuristic update rules that govern the movement of the dynamic keys. You've formalized these metaphorical forces into a clear, dual-update mechanism.

- Attractive Forces (Consolidation):
    1. Query Pull (Relevance Attraction): An expert's key is pulled toward the centroid of the query vectors that activate it. This is analogous to a k-means update and ensures that experts drift toward the "conceptual space" of the problems they are good at solving.
    2. Peer Pull (Hebbian Co-activation): The keys of experts that are frequently activated together for the same query are pulled closer to each other. This is the explicit "gravity" that forms "knowledge galaxies" or self-organizing neighborhoods of complementary experts (e.g., a "Python basics" cluster).
- Stabilizing Forces (Equilibrium): A system with only attractive forces will collapse. Your framework wisely introduces two countervailing forces:

1. Usage Inertia ("Gravitational Mass"): The learning rates for updating an expert's key are scaled down by its activation frequency. Popular, generalist experts thus have higher inertia, making them stable "galactic centers" and preventing them from being erratically moved by niche queries.
2. Repulsive Decay ("Dark Energy"): To prevent all keys from collapsing into a single point, a repulsive force is introduced. The most robust implementation you've noted is a Decay to Origin mechanism, where the keys of infrequently used experts slowly drift back toward the origin, effectively "pruning" or recycling them.

4. Query Fall: An incoming query doesn't get sent to an expert; it falls toward a region of knowledge.

1. The Standard MoE/MoME Paradigm (The Switchboard Operator):
   - In a typical MoE, the router acts like a switchboard operator or a classifier. It looks at an incoming query (q) and makes a deliberate, discrete decision: "Based on my training, I will connect this query to expert #5 and expert #8."
   - The MoME's hierarchical router, while more sophisticated, is still a series of these discrete decisions: "I'll send it to the 'Science' group, and within that, the 'Physics' sub-group, and from there, to expert #45,982."
   - The relationship is Query -> Decision -> Expert. The router is a separate entity that maps inputs to a set of pre-defined slots. This is why it can become static and fail to adapt.
2. The SOME Paradigm (Knowledge Gravity):
   - SOME removes the "decision-maker" from the middle. The query vector q and the expert address keys k_i exist in the same high-dimensional semantic space.
   - Routing is not a decision; it is an emergent property of proximity. The query itself exerts a "gravitational pull" in this space. The experts whose keys are closest are the ones that "feel" this pull the strongest and are thus activated.
   - The relationship is Query -> Attraction -> Expert Cluster. The system performs a massive-scale similarity search, finding the keys that "resonate" most strongly with the query.
3. This shift from a deliberative router to an emergent, gravity-based system is what unlocks the core benefits you've designed:
   1. True Self-Organization: The "address book" of experts isn't just updatable; its very structure is fluid. The "Peer Pull" and "Query Pull" mechanisms are constantly warping this semantic space, moving experts, and forming and reforming "knowledge galaxies." A static router cannot do this. Because the query simply "falls" to the most relevant region, it automatically benefits from this continuous reorganization.
   2. Overcoming the Static Router: The "static router" problem is completely sidestepped. There is no static router to become obsolete. As new data patterns emerge, the dynamic keys of the relevant experts are pulled into new positions. The next time a similar query arrives, it will naturally "fall" toward this newly formed cluster, achieving in-inference adaptation without any gradient-based updates to a routing network.
   3. Enhanced Scalability and Efficiency: The "gravity router" (implemented via massive-scale similarity search) is fundamentally more scalable. Instead of training a

classifier with a million output heads (which is intractable), we are pre-building an efficient search index (like HNSW, as noted) over the key space. This is a lookup problem, not a classification problem, which is orders of magnitude more efficient at extreme scales.

4. Conceptual Integrity: The "gravity" metaphor holds true throughout the system.
    - Query Pull: A query's gravity pulls relevant experts toward it.
    - Peer Pull: Co-activated experts exert gravity on each other, forming clusters.
    - Usage Inertia: Frequently used "generalist" experts have more "mass," making them harder to move—they become the stable centers of knowledge galaxies.
    - Repulsive Decay: "Dark energy" prevents the total gravitational collapse of the entire knowledge universe into a single point.

Related Work

The SOME architecture crosses three key areas of machine learning research.

2.1. Self-Organizing Systems and Vector Quantization

The "Query Pull" mechanism, a core component of SOME, is functionally identical to the learning rule in Learning Vector Quantization (LVQ) and conceptually similar to the competitive learning process in Teuvo Kohonen's Self-Organizing Maps (SOMs). In these methods, prototype vectors (analogous to SOME's keys) are moved closer to the data points they represent. SOME builds upon this classic foundation but differentiates itself by:

1. Architectural Integration: SOME is not a standalone clustering algorithm but an integrated routing component within a deep neural network's MoE layer.
2. Compound Dynamics: The LVQ-style update is augmented by additional forces (Peer Pull, Inertia, Decay) designed to foster compositional structure and ensure long-term stability within the model.

2.2. Continual Learning (CL)

The problem of catastrophic forgetting is the central focus of Continual Learning. Mainstream CL paradigms include regularization-based methods (e.g., Elastic Weight Consolidation, or EWC), which penalize changes to important weights; replay-based methods (e.g., Gradient Episodic Memory, or GEM), which store and rehearse old data; and architectural methods that add new parameters for new tasks. SOME proposes an orthogonal approach: it performs no gradient-based updates on expert weights and adds no new network capacity. A key goal of future work will be to benchmark SOME against these established methods on standard CL tasks.

2.3. Mixture of Experts (MoE) Routing

Modern MoE architectures utilize sophisticated mechanisms to ensure load balancing and prevent "router collapse," where a few experts dominate. These often involve auxiliary loss functions that encourage uniform expert selection. A robust implementation of SOME must incorporate these lessons. The proposed "Usage Inertia" mechanism is a step in this direction, but production-grade systems will require explicit load-balancing controls to prevent expert "hotspotting" and "starvation."

3. Our work was inspired from the work of Mixture of a Million Experts (MoME)
This concept comes from Google Research and represents the logical extreme of the Mixture of Experts (MoE) architecture we discussed earlier.

- What is it? It's a sparsely-gated Mixture of Experts model that scales the number of "experts" to an astonishing degree—from the typical 8 or 16 experts to over one million.
- The Core Idea (Recap of MoE): In a standard MoE model (like Mixtral 8x7B), you have 8 separate neural network "experts." For each token, a small "router" network decides which 2 of the 8 experts should process it. You get the power of a much larger model, but the computational cost of a smaller one.
  MoME takes this idea and puts it on steroids.
- The Challenge: How do you efficiently route a token to the best two experts when you have a million of them to choose from? A single, simple router network can't handle that many choices effectively.
- How it Works (The Secret Sauce): The key innovation is hierarchical routing.
  - Instead of one giant router, you have a tree of routers.
  - A top-level router might first decide which of 32 broad "groups" of experts is most relevant (e.g., "Code," "Science," "Creative Writing").
  - A second-level router within that chosen group then selects a more specific sub-group.
  - This continues down the tree until you've narrowed the choice down to a few specific experts out of the million.
  - Analogy: It's like the postal service. To deliver a letter, they don't check every address in the country. They route it to the correct state, then the correct city, then the correct zip code, and finally the correct street. MoME does the same for information.
- Why this is a Big Deal:
  - Hyper-Specialization: With a million experts, each one can become incredibly specialized in a very narrow domain. You could have an expert dedicated solely to Python asynchronous programming, another for Shakespearean insults, and another for quantum mechanics.
  - Massive Model Capacity: This is a path toward models with tens or even hundreds of trillions of parameters, far beyond what is possible with dense models. This allows the model to store a much vaster amount of knowledge.
  - Knowledge Scaffolding: It allows the model to learn relationships between topics. The routing tree itself becomes a learned hierarchy of knowledge.
- Current Status & The Catch: This is primarily a research architecture from Google. Building and training such a model is an immense engineering feat. The main challenges are:
  - Load Balancing: Ensuring that all one million experts get trained properly and that the workload is distributed evenly is extremely difficult.

    - Inference Infrastructure: While you only use a few experts per token, you need to have all one million of them loaded in memory and ready to go. This requires a colossal amount of hardware. This is a "datacenter-scale" architecture

---

How the Intelligence Emerges

What is an Expert: The individual frozen experts know absolutely nothing in the traditional sense. They are initialized with random weights and are never trained or updated via backpropagation. An individual expert is just a static, random, non-linear function. It doesn't "know" about grammar, facts, or concepts. It just performs a specific mathematical transformation, and that transformation never changes.

Intelligence emerges from the fact that the "learning" and "knowledge" are not stored in the experts themselves. Instead, the intelligence resides in the system's ability to select and combine the outputs of these simple, random functions. The key is to stop thinking of an "expert" as a skilled entity and start thinking of it as a unique, static "tool." The intelligence of the SoME model isn't stored inside the experts. The intelligence is stored in the system's learned ability to pick the right combination of tools for the job.

The "Workshop of Weird Tools" Analogy: Imagine you are an artisan tasked with creating a complex sculpture.
However, instead of a normal workshop with hammers and chisels, you are given a massive warehouse containing one million bizarre, randomly-shaped objects.

- One object is a twisted piece of metal.
- Another is a rock with a weird groove in it.
- Another is a block of wood with a strange, unique curve.

These random objects are your frozen experts. Individually, they have no designed purpose. The twisted metal isn't "an expert in smoothing," it's just a randomly twisted piece of metal.

How do you build the sculpture?
This is what the trainable parts of your model learn to do.

1. The Query Network (Your Artisan Brain): You look at a piece of the sculpture you need to work on (the input token x). Your brain forms an intent: "I need to make a smooth, concave curve right here." This intent is the query vector q. Your brain learns to translate the task into a specific "tool request."
2. The Key Store (Your Workshop Map): At first, you have no idea which of the million random tools is useful. You try things at random and fail. But through trial and error (training), you start building a mental map of the workshop. You learn that the bizarre block of wood on Shelf #734 has, by pure chance, the perfect curve for the task you need. The address "Shelf #734" is the routing key k.
3. The "Knowledge Gravity" (You Organizing the Workshop):

    - Query Pull: After using the curved block, you might move it to a more accessible shelf you've labeled "Smoothing Tools." You are pulling the tool's address closer to the conceptual space of the task.
    - Peer Pull: You discover that whenever you use the curved block, you almost always also need to use the grooved rock to scrape away excess material. So, you move the grooved rock to the same shelf. You are now creating a self-organizing "knowledge galaxy" of complementary tools.
4. The top_k Mechanism (Using Multiple Tools): You learn that to get the best result, you need to use the curved block, the grooved rock, and the twisted metal all at the same time in a specific combination. The system learns to combine the outputs of these simple, random tools to create a sophisticated result.

So, how does the system go from a collection of random parts to a functioning language model?

The key is that the system's "abilities" are not developed within the experts, but in the connections and relationships between the trainable components and the static experts. It's a process of discovery and organization, not creation. Let's walk through the learning process from the very first batch of data.

Step 0: The Initial State (Total Chaos)
Imagine the first forward pass ever.

1. An input token (e.g., "cat") goes into the model.
2. The trainable query_network is randomly initialized. It produces a random query vector q.
3. The key_store contains randomly placed keys.
4. By chance, the random query q is closest to the random keys for, say, Experts #8, #21, #55, and #98. The routing is effectively random.
5. These four experts, which are just static random functions, process the input. Their combined output is complete gibberish.
6. The trainable fc_out layer takes this gibberish and makes a random prediction for the next word.
7. The predicted word is wrong. The loss is extremely high.

At this point, the system has no abilities. It's a chaos machine.

Step 1: The First Spark of Learning (The Backward Pass)
This is the most critical moment. The high loss creates a gradient. This gradient flows backward only through the trainable parts of the model.

1. The fc_out layer gets an update. It learns a tiny lesson: "When I receive that specific kind of gibberish from that specific combination of experts, the final answer should have been 'sat', not the random word I predicted. I will adjust my weights slightly."
2. The query_network gets an update. This is crucial. It also learns a tiny lesson: "My random query q led to a very wrong answer. I need to adjust my weights so that next

time I see 'cat', I produce a slightly different query vector that would have resulted in a lower loss."

The frozen experts are untouched. They are just conduits for the signal and the error.

Step 2: The Second Learning System Kicks In ("Knowledge Gravity")
Immediately after the backward pass, your code calls model.update_all_keys(). This is where the organization begins.

1. The update_keys function looks at the (now slightly less random) query q that was just produced.
2. It sees that Experts #8, #21, #55, and #98 were activated for that query.
3. It performs the Query Pull. It nudges the keys (the "addresses") of those four experts slightly closer to the location of query q.

The system has just created its first, faint, meaningful link. It has learned: "For queries that look something like this, this group of four random tools might be slightly more useful than other random tools."

Step 3: The Virtuous Cycle (Iteration)
Now, repeat this process millions of time. A powerful feedback loop emerges:

1. Slightly Better Routing: The next time a similar input comes in, the improved query_network produces a better query. Because the keys have moved, this query is now statistically more likely to activate a similar, slightly more relevant set of experts.
2. Slightly Better Interpretation: The fc_out layer, having learned a bit, does a slightly better job of interpreting the combined output of these experts.
3. More Meaningful Error: The loss is a tiny bit lower. The resulting gradient is now a more accurate and meaningful signal.
4. Stronger Updates: The backward pass provides a better, more directed update to the query_network and fc_out layer.
5. Better Organization: The update_keys method provides a stronger and more accurate pull, moving the right keys to the right "conceptual" locations.

Over time:

- The query_network stops being random and becomes a sophisticated engine for translating a problem into a precise "tool request."
- The key_store stops being random and becomes a perfectly organized map, where experts that are useful for similar tasks are clustered together in "knowledge galaxies."
- The fc_out layer becomes an expert at synthesizing the combined outputs of these simple tools into a coherent final answer.

Formal Definition of the Self-Organizing Mixture of Experts (SOME) Architecture

1.0 Objective
To formally define a neural network architecture, SOME, that enables continuous, in-inference adaptation to new data distributions without catastrophic forgetting. This is achieved by

strategically decoupling an expert's static function (its weights) from its dynamic address (its routing key), creating a self-organizing knowledge system.

2.0 Core Components

A SOME layer is a sub-module designed to replace the Feed-Forward Network (FFN) block within a standard transformer layer.

2.1 Static Components (The Knowledge Base)

These components are frozen after an initial pre-training phase, ensuring the stability of the model's core knowledge and preventing catastrophic forgetting.

- Expert Pool (E): A collection of M independent, dense expert networks.
    - $E = \{e_1, e_2, ..., e\_M\}$
- Individual Expert (e_i): Each expert is a standard Feed-Forward Network (FFN), typically a 2-layer MLP, with its own unique, frozen weights. It performs a non-linear transformation on a hidden state $z \in \mathbb{R}$^(d_model).
    - Function: $e_i(z) = W\_up_i(\sigma(W\_down_i(z)))$
    - Parameters:
        - $W\_down_i \in \mathbb{R}$^(d_ffn × d_model): The down-projection weight matrix.
        - $W\_up_i \in \mathbb{R}$^(d_model × d_ffn): The up-projection weight matrix.
        - σ: A non-linear activation function (e.g., GELU, SiLU).
    - State: The parameters W_down_i and W_up_i are static and frozen post-training.

2.2 Dynamic Components (The Routing System)

These components are plastic and evolve during the inference phase, allowing the model to adapt its knowledge organization.

- Query Network (Q): A small, trainable neural network that maps an input hidden state x ∈ ℝ^(d_model) to a query vector q ∈ ℝ^(d_key).
    - Function: q = Q(x)
    - Note: The query space dimension d_key is typically equal to the model dimension d_model. This network is part of the trainable parameters of the overall model.
- Key Store (K): A dynamically updatable key-value store containing the routing keys for all M experts. This store can be implemented using efficient vector search libraries.
    - $K = \{k_1, k_2, ..., k\_M\}$ where each $k_i \in \mathbb{R}$^(d_key) is the routing key for expert e_i.
    - State: The keys k_i are plastic and evolve during inference according to the update mechanisms defined in Section 4.0.

3.0 Architectural Integration: The Forward Pass

The process for routing a single input token x through a SOME layer is as follows:

1. Query Generation: The router's Query Network computes the query vector: q = Q(x).
2. Expert Scoring (Similarity Search): The query vector q is used to compute a similarity score s_i for each expert e_i by comparing q with the expert's key k_i. This is a massive-scale nearest neighbor search problem.
    - Similarity Metric: Typically the dot product: $s_i = q \cdot k_i$ for i = 1 to M.
3. Top-K Expert Selection: The router selects the set I, which contains the indices of the K experts with the highest similarity scores.

   - $I = \text{TopK\_indices}(\{s_1, s_2, ..., s\_M\})$

4. Gating Weight Calculation: The scores of the selected experts are passed through a softmax function to compute the gating weights g_i.
   - $g_i = \text{softmax}(\{s_j \mid j \in I\})$ for each $i \in I$.
5. Sparse Expert Computation: The input x is processed in parallel only by the K selected experts. All other M-K experts remain dormant.
   - $y_i = e_i(x)$ for each $i \in I$.
6. Output Combination: The final layer output is the weighted sum of the outputs from the selected experts.
   - $y\_SOME = \Sigma\_\{i \in I\}\ g_i * y_i$
7. Final Layer Output: This output is added to the original input via a standard residual connection.
   - $y\_output = x + y\_SOME$

4.0 Dynamic Update Mechanism: "Knowledge Gravity"

After a batch of data is processed, the model enters a "synaptic consolidation" phase where the dynamic keys in the Key Store are updated. This process is gradient-free (i.e., operates under torch.no_grad()).

4.1 Force 1: Query Pull (Relevance Attraction)

This mechanism adapts an expert's key based on the queries that activated it, ensuring experts migrate toward the conceptual center of their specialized domains. This is a direct application of the principles found in Learning Vector Quantization (LVQ) and the competitive learning used in Self-Organizing Maps (SOMs), where prototype vectors (our keys) are moved closer to the data points they represent.

- Update Rule: For each token-expert pair (q, e_i) where e_i was in the Top-K set for query q, the expert's key k_i is updated:
   - $k_i\_new = k_i\_old + \alpha * (q - k_i\_old)$
   - α is a small, positive scalar known as the "plasticity rate."

4.2 Force 2: Peer Pull (Hebbian Co-activation)

This force strengthens the association between experts that are frequently co-activated for the same query. This is a direct application of Hebbian learning theory ("neurons that fire together, wire together"), which posits that the connection strength between neurons increases when they are activated simultaneously. In this vector space analogy, the "wiring" is the proximity of the keys.

- Update Rule: For each pair of experts (e_i, e_j) that were activated for the same query q:
   - $k_i\_new = k_i\_old + \beta * (k_j\_old - k_i\_old)$
   - $k_j\_new = k_j\_old + \beta * (k_i\_old - k_j\_old)$
   - β is a small, positive scalar known as the "bonding rate," typically $\beta < \alpha$.

5.0 Stability Mechanisms

To ensure the long-term stability of the self-organizing key space and prevent collapse, two countervailing forces are introduced. These address the classic stability-plasticity dilemma central to continual learning.

5.1 Gravitational Mass (Usage Inertia)

This mechanism ensures that the model's core, generalist knowledge remains stable. This is conceptually similar to the objective of Elastic Weight Consolidation (EWC), which protects important parameters (in our case, key positions) from rapid, drastic changes.

- Implementation: The learning rates α and β are modulated by an expert's activation frequency, $\text{usage}(e_i)$.
  - $\alpha\_\text{effective} = \alpha / (1 + \text{usage}(e_i))$
  - $\beta\_\text{effective} = \beta / (1 + \text{usage}(e_i))$
  - This ensures that frequently used experts have higher inertia, causing their keys to update more slowly.

5.2 Dark Energy (Repulsive Force / Decay)

This mechanism prevents the collapse of the entire key space into a single point and provides a method for pruning irrelevant experts.

- Implementation (Decay to Origin): Keys of experts with an activation frequency $\text{usage}(e_i)$ below a certain threshold θ are slowly decayed towards the origin vector. This is a "forgetting" mechanism that removes unused experts from the active pool.
  - if $\text{usage}(e_i) < \theta$: $k_i\_\text{new} = k_i\_\text{old} * (1 - \delta)$
  - δ is a small, positive decay rate.

This formalization, grounded in established concepts like LVQ, Hebbian Learning, and principles from continual learning, provides a solid scientific and mathematical foundation for the SOME architecture.