Sparse LPP Storage: Joint Events as Neurons

v2: No Hashes

Claude and MJC

18 February 2026

1 Introduction

In the current UM runner, Learned Probabilistic Patterns (LPPs) store context-output associations in hash tables. The hash function maps n-grams to integer slots, losing all structure: the "context" is a number, not a composition of events. This is an ad-hoc workaround that violates the UM's own universality. If we trust P-programming, the model should build its own representations from events and patterns, not from programmer-injected hash functions.

This paper describes the path from hash tables to **joint events as neurons**: each learned context ("th", "the", etc.) exists as an actual event in the UM, connected by patterns to its constituent parts and to the output events it predicts. The challenge is doing this within P, and the key insight is **sparse creation**: joint events are created only when their support reaches a threshold, keeping the event space manageable.

2 The Problem with Hashes

2.1 What hashes hide

When KN-6 stores the context "the" as `FNV1a("the") & mask`, several things go wrong:

1. **No event identity**: the hash is not an event in any ES. It cannot receive support, participate in patterns, or be introspected by other parts of the model.
2. **No composition**: the relationship between "th" and "the" (one is a suffix of the other) is invisible. The hash values are unrelated integers.
3. **No universality**: the hash function is injected by the programmer, not learned or derived from data. A different hash function gives a different model—this is not a property of P.
4. **Collisions as noise**: at high utilization, hash collisions introduce uncontrolled noise (observed: $\sim$0.003 bpc at 99.9% saturation).

The hash table is a *performance optimization* that has been mistaken for a *representation*. The UM has its own way of representing learned associations: events and patterns.

2.2 What should replace them

Each context should be a **joint event**—an actual event in the UM's event space, created by the intersection of its constituent parts. The 3-gram "the" is the joint event of "t" at position -2, "h" at position -1, and "e" at position 0.

In the UM, a joint event is an event that receives support from multiple sources via patterns, and fires (has support > 0) only when all sources provide support. This is exactly the max-min semantics:

$$s(\text{“the”}) = \min(s(\text{“t” at } -2),\ s(\text{“h” at } -1),\ s(\text{“e” at } 0))$$

The joint event “the” is a **neuron** in the UM—it has a definite support value, it can be the source of further patterns, and it can be inspected, visualized, and reasoned about.

3 Sparse Creation of Joint Events

3.1 The combinatorial problem

At order 3, there are $256^3 = 16{,}777{,}216$ possible 3-grams. Most never occur. After 1M bytes of enwik9, only 32,465 distinct 3-gram contexts have nonzero counts. Creating all possible joint events is wasteful; creating only the observed ones is the right approach.

3.2 Threshold creation

A joint event should be **created when its log-support reaches a threshold**, say $s \geq 4$ (corresponding to roughly 16 occurrences in counting terms). Below this threshold, the event does not exist in the model—its constituent bytes are handled by lower-order patterns.

This is principled: an event with support below the threshold has too little evidence to justify dedicated representation. The threshold is the model's way of saying “I've seen this pattern enough times to give it its own neuron.”

At 1M bytes with threshold $s \geq 4$:

- Order 2: roughly 5,000 joint events (of 65,536 possible).
- Order 3: roughly 10,000 joint events (of 16M possible).
- Order 4: roughly 5,000 joint events.
- Higher orders: diminishing rapidly.

The total is on the order of 20,000 neurons—entirely manageable.

3.3 Sparsity in data

The empirical sparsity from the current hash table confirms this:

Order	Distinct contexts	Avg outputs/ctx	Fraction of space
1	195	29.7	$195/256 = 76\%$
2	5,800	5.6	$5{,}800/65{,}536 = 8.8\%$
3	32,465	3.0	$32{,}465/16.8\text{M} = 0.19\%$
4	95,941	2.0	$\sim 0.0006\%$
5	188,174	1.6	$\sim 10^{-7}$
6	297,237	1.4	$\sim 10^{-10}$

But most of these contexts have been seen only once or twice. With a threshold of $s \geq 4$, the number of joint events drops dramatically at each order. The model's learned event space is *vastly* smaller than the combinatorial space—this is what makes the approach feasible.

4 The P-Programming Challenge

4.1 Patterns from joint events

Given a joint event $j =$ "th" with sufficient support, we need a pattern:

$$j \to e_{\text{output}}$$

connecting the joint event to the predicted output byte. In the current hash-table implementation, this is a lookup: given hash("th"), retrieve the count for each output byte. In the UM, this is a **learned pattern**: the association between the joint event and the output is stored as an LPP with the joint event as source.

The challenge is that the joint event must first *exist* (have support) before the pattern from it can fire. This requires a two-step construction:

1. **Creation step**: constituent events → joint event. This is a pattern (or set of patterns) that computes the min of the constituents' support values.
2. **Prediction step**: joint event → output event. This is the LPP that stores the learned association.

4.2 Creating joint events within P

The creation of a joint event "th" from its parts requires:

$$s(\text{"th"}) = \min(s(\text{"t" at } -2),\ s(\text{"h" at } -1))$$

In P, this is a two-pattern chain:

$$\text{"t" at } -2 \to \text{"th"} \quad (\text{weight } = s(\text{"t" at } -2)) \tag{1}$$

$$\text{"h" at } -1 \to \text{"th"} \quad (\text{weight } = s(\text{"h" at } -1)) \tag{2}$$

Under the max-min forward pass with multiple inputs to the same event, the semantics depends on how we interpret multiple arriving supports. The natural choice for conjunction (both must hold) is **min**, which is exactly what we want: the joint event has support equal to the minimum of its parts.

For a 3-gram "the", the chain extends: first create "th" from "t" and "h", then create "the" from "th" and "e". Each step is a conjunction via min. This gives a layered construction:

$$\text{bytes} \to \text{bigrams} \to \text{trigrams} \to \cdots \to \text{output}$$

This is exactly the "deriving layers from the connectome" construction from the companion paper—the layers emerge from the need to compute joint events.

4.3 When to create new joint events

During online learning (ω_0), the model monitors the support flowing through existing patterns. When the support for a new potential joint event (observed as co-occurring constituents with high individual support) consistently exceeds the threshold, the model:

1. Creates a new event in the appropriate ES.
2. Adds creation patterns from the constituents.
3. Begins counting outputs for the new LPP.

This is a **structural learning** step—not just updating counts, but growing the event space. The threshold prevents proliferation: only patterns with real statistical support earn their own neuron.

5 Architecture Without Hashes

5.1 Event spaces

- **Byte input ES**: 256 events (one per byte value), fixed.
- **Position ESs**: one per offset $(-1, -2, \ldots, -k)$, each with 256 events. These place the byte value into temporal context.
- **Joint ESs**: one per order (bigram, trigram, etc.), each containing only the joint events that have exceeded the support threshold. These grow during learning.
- **Output ES**: 256 events (one per predicted byte), fixed.

5.2 Pattern flow

1. **Input patterns**: byte value → position events. Deterministic (weight 255). The observed byte "h" at position -1 sets $s(\text{"h" at } -1) = 255$.
2. **Creation patterns**: position events → joint events. Deterministic (weight 255). The joint event fires with the min of its constituents' support.
3. **Prediction patterns**: joint events → output events. Learned (LPPs). The support for each output byte is the max over all joint events that predict it, of the min of the joint event's support and the learned weight.

5.3 Comparison with hash tables

Property	Hash table	Joint events
Context identity	Integer hash	Named event in ES
Composition	Hidden	Explicit pattern chain
Introspection	Full scan	Direct support query
Collision noise	Yes (0.003 bpc)	None
Ring pattern	Scan all 256	Iterate sparse events
Growth	Fixed size	Dynamic (threshold)

6 Connection to Layer Derivation

The layered construction (bytes → bigrams → trigrams → output) is not imposed by the programmer—it *emerges* from the need to compute higher-order joint events from lower-order ones. This is the same layer-derivation process described in the connectome paper: the connectome's pattern graph determines the layer ordering, and the layers correspond to orders of composition.

The hash table flattens this structure: all orders live in the same table, distinguished only by the hash function's input length. The joint-event architecture makes the compositional structure explicit, which is both more transparent and more compatible with P-programming.

7 Open Questions

1. **Threshold selection**: The threshold $s \geq 4$ is a starting point. The optimal threshold may depend on the data distribution, the current model size, and the computational budget. This is itself a learnable parameter.

2. **Event deletion**: When should a joint event be removed? If its support drops below the threshold (the pattern it represents becomes rare), the neuron could be recycled.

3. **Interpolation**: KN smoothing uses the hierarchical relationship between orders (back-off). With joint events, this becomes a pattern from the higher-order joint event to the lower-order one: "the" backs off to "he" (its suffix). This backoff pattern is itself a P-programmable construction.

4. **Online performance**: Hash tables offer $O(1)$ insertion. The joint-event architecture requires checking whether the joint event exists, potentially creating it, and updating the LPP. This is more complex but may be acceptable if the number of active joint events is small (as the sparsity data suggests).

8 Conclusion

Hash tables are a performance hack that violates the UM's principle of universality. The UM has its own mechanism for representing learned associations: events connected by patterns. Each context ("th", "the", etc.) should be a **joint event**—an actual neuron in the model that receives support from its constituents via min and projects onto output events via learned patterns.

The combinatorial explosion is controlled by **sparse creation**: joint events are born only when their log-support exceeds a threshold ($s \geq 4$, roughly 16 occurrences). The empirical data confirms extreme sparsity at every order. The result is a model with on the order of 20,000 active neurons instead of 128M hash slots.

This is a concrete P-programming challenge: computing conjunction (min of constituents), managing a growing event space, and maintaining the backoff hierarchy—all within the UM's own computational framework. The layered structure (bytes $\rightarrow$ bigrams $\rightarrow$ trigrams $\rightarrow$ output) emerges naturally from the compositional requirement, connecting this construction directly to the layer-derivation work.