Training Spiking Neural Networks Using Lessons from State Space Models

Andrew Smith, Taylor Kergan, Assel Kembay, Kimia Gholami,
Vincent Harkins, Binh Nguyen, Ruhai Lin, Ridger Zhu, Jason K. Eshraghian
University of California, Santa Cruz
Santa Cruz, CA, USA
andsmi@ucsc.edu, tkergan@ucsc.edu, akembay@ucsc.edu, fgholami@ucsc.edu,
vharkins@ucsc.edu, bnguy177@ucsc.edu, rlin50@ucsc.edu, ridger@ucsc.edu, jsn@ucsc.edu

***Abstract*—Anyone who has trained a spiking neural network (SNN) knows they can be difficult to work with: discrete spike events, surrogate gradients, subtle mismatch with deployment hardware, and the list goes on. A more fundamental issue lurks underneath: most SNNs are still trained by unrolling their dynamics *sequentially* in time. As sequence length grows, wall-clock time grows with it. Despite the energy efficiency of inference, this gain is partially offset by prohibitively slow training runs.**

In this paper, we accelerate core SNN primitives by borrowing ideas from state space models (SSMs). Recent architectures like Mamba use SSMs to turn recurrent neurons into convolutions or prefix scans that run in parallel over sequence length. This parallelism is possible when the recurrence is linear through time, and standard spiking neurons *almost* fit this criteria. We show how parallelization strategies can accelerate SNNs and when each is most appropriate. New spiking neuron models are introduced into SNNTORCH that provide substantial speedups while improving long-range sequence performance. Each construction comes with concise PyTorch code, and companion tutorials will be released at https://snntorch.readthedocs.io/.

***Index Terms*—Spiking neural networks, State space models, Neuromorphic computing, Sequence modeling, Parallel training, Associative memory, snnTorch**

I. Introduction

If you're reading this, then there is a high chance you've heard the phrase *"Spiking neural networks are the third-generation of deep learning"* [1], or that *the brain is the perfect blueprint for building more efficient neural networks* [2]. Despite that, training SNNs using deep learning-based techniques is considerably less stable and less scalable than standard deep learning. But even if a GPU isn't made of the same physical goo as brains, there are still a variety of features that gives neuromorphic engineers confidence that the brain is a valuable source of inspiration for enhancing deep learning workloads. Sparse activations reduce the pressure on memory access; low-precision variables reduce memory utilization and simplify arithmetic; dynamic, time-varying encoding schemes offer highly compressed representations of the world around us.

The progress in overcoming obstacles to scaling SNNs is real. It used to be thought that the non-differentiability of spikes was the biggest challenge in training SNNs. But surrogate gradients and quantization-aware training schemes have shown the remarkable robustness of neural networks to approximating the optimization process. Alongside this, rapid maturation of modern SNN libraries and training frameworks have made it routine to build, train, and deploy spiking models using standard deep-learning tooling [2].

This past decade has seen SNNs become more powerful by drawing upon many of the advances in deep learning. But the advent of large language models has drastically widened the gap in capability between SNNs and modern deep learning, and the SNN world has struggled to keep up. At present, the primary bottlenecks appear to be:

- **Binary activations** are an extremely lossy mode of compression
- **Sequential, time-unrolled training** is prohibitively slow for optimizing large-scale models and large-scale datasets

The most common solution to the binary activation issue is by introducing *graded spikes*, where spikes are prescribed a magnitude. In hardware, if a spike is being transmitted across cores, there is very little additional cost in attaching several additional bits. Most modern neuromorphic processors spinning out from industry have followed suit, including Intel's Loihi 2, SynSense's accelerators, BrainChip's Akida, and Innatera's SNP T1. The takeaway from this is that the reduced communication cost from sparse activations is far more important than the reduced arithmetic effort from binary precision.

But what solves the second issue of slow and sequential training? Most training pipelines still pay an unavoidable systems tax: neuron dynamics are evaluated by stepping through time one tick at a time, and gradients are computed by back-propagating through that unrolled computation graph [2], [3]. This makes the wall-clock cost scale linearly with sequence length T, both in compute and in activation storage, and it turns "long context" into "good luck" once T hits the thousands. In contrast, dense sequence models have learned to train comfortably at even thousands to tens-of-thousands of steps, because modern architectures and kernels are built around parallel evaluation over the sequence dimension [4]–[6].

In the RNN world, this exact bottleneck got "fixed" by switching viewpoints. Instead of treating recurrence as an inherently step-by-step program, you model it as a (mostly)

linear dynamical system and then use parallel primitives to evaluate it across the whole sequence. That is the state-space model (SSM) line of work [7]. SSMs have become a mainstream sequence-modeling toolchain, with families like S4 and Mamba demonstrating that structured state updates can be trained at scale and perform competitively on long-context benchmarks [8]–[10]. Hybrid models are now popping up across modalities, which forwards the core point: recurrence does not have to be sequential [11]–[13].

Thus, there exists a natural synthesis. SNNs and SSMs sit on the same dynamical-systems foundation. State evolves over time under simple update rules, and the modeling question is how to parameterize and compute those updates efficiently [14]. The SSM literature has recently produced a mature playbook for making recurrence scale [8], [10], and SNNs provide a mature interface for sparse, event-driven computation. Putting these together means integrating the scalability of SSM-based recurrence into SNN training, bringing the systems advantages of spikes into scalable state-based models, and retaining the efficiency motives that made spikes interesting in the first place.

Several recent works target the same scalability bottleneck we emphasize (i.e. the $\mathcal{O}(T)$ cost of time-unrolled spiking dynamics and their gradients) by making spiking neuron updates more amenable to *temporal parallelism*, for example via convolutional or scan-style evaluation of membrane dynamics over the full sequence [15], [16]. Separately, spiking sequence models have also been explored that borrow structure from state-space models to improve long-range modeling while retaining spike-based nonlinearities [17].

Motivated by this structural compatibility between spiking neuron dynamics and SSMs, our goal is to make the SSM perspective concrete and usable inside SNNTORCH. In Sec. II, we distill a small taxonomy of spiking-amenable recurrences and their time-parallel evaluation strategies under practical PyTorch constraints. Then, we instantiate a convolutional form (Sec. III) and a prefix-scan form (Sec. V) that integrate with standard training workflows. For each neuron model, we evaluate these primitives both as systems components and as sequence models, elucidating when each strategy is most appropriate.

II. Mapping SSM Structure to "Generations"

SSMs cast temporal updates as linear recurrences:

$$x_t = Ax_{t-1} + Bu_t \tag{1}$$

where x_t is the hidden state, u_t is the input, and A, B are parameter matrices. When the update is linear through time, we can often replace an $O(T)$ rollout with time-parallel primitives. Table I summarizes the options we use in this paper; the choice among them depends on how much structure the recurrence retains.

State space models are often grouped into "generations" that trade off recurrent expressivity against the strength of time-parallel primitives required for evaluation [18]. Fig. 1 highlights the classes most relevant for spiking-style recurrences.

Generation 1 models use *time-invariant diagonal* updates. For a membrane potential v_t with input current I_t and fixed decay β:

$$v_t = \beta v_{t-1} + I_t. \tag{2}$$

Linearity through time yields a closed-form exponential convolution $v = h * I$ with kernel $h_k = \beta^k$, enabling full parallel evaluation via 1D convolution. Spikes are generated by thresholding on θ: $s_t = H(v_t - \theta)$.

Generation 2 models extend the state to a *matrix* $S_t \in \mathbb{R}^{d\times n}$ and add associative-memory structure via outer-product writes with input-dependent decay:

$$S_t = S_{t-1}\,\mathrm{diag}(\alpha_t(I_t)) + v_t k_t^\top, \tag{3}$$

where v_t and k_t are value and key vectors, and $\alpha_t(I_t) \in (0,1)^n$ gates each column independently. Because each column evolves as a multiply–add recurrence, the full update remains parallelizable via prefix products and sums. The spikes are generated elementwise: $z_t = H(S_t - \theta)$.

Generation 3 models couple decay and content-based writes through projectors like $(I - \beta_t k_t k_t^\top)$, requiring general associative-scan kernels not available in PyTorch eager mode. We therefore focus on Generation 1 and Generation 2 constructions, as shown in Fig. 1[1].

III. StateLeaky: a "Generation 1" spiking state-space neuron model

A. From LIF to a reset-free linear recurrence

We start from the discrete-time leaky integrate-and-fire (LIF) neuron. Using the notation from Section II, let v_t denote the membrane potential at timestep t, with input current $I_t = WX_t$ (where W is a weight matrix and X_t the raw input). The standard LIF dynamics are:

$$v_t = \beta v_{t-1} + I_t - s_{t-1}\theta, \tag{4}$$

where $s_{t-1} \in \{0,1\}$ is the spike at the previous timestep, θ is the threshold, and β is the decay factor. The reset term $s_{t-1}\theta$ makes the recurrence nonlinear through time. Removing the reset by setting $s_{t-1} = 0$ yields the reset-free membrane $\tilde{v}$:

$$\tilde{v}_t = \beta \tilde{v}_{t-1} + I_t, \tag{5}$$

which exactly matches the Generation 1 form in Equation (2). Unrolling reveals that $\tilde{v}_t$ is a discrete convolution of the input current with an exponential kernel $h_k = \beta^k$, enabling parallel evaluation over the full sequence.

During training, spikes are computed from this reset-free membrane:

$$s_t = H(\tilde{v}_t - \theta), \tag{6}$$

where H is the Heaviside function. At inference, we can retain parallel evaluation or revert to the original iterative dynamics for event-driven deployment.

[1]For Generation 2, note that $S_t = S_{t-1} \odot (\mathbf{1}\alpha_t^\top) + v_t k_t \iff S_t = S_{t-1}\,\mathrm{diag}(\alpha_t) + v_t k_t^\top$.

TABLE I
TOOLS FOR PARALLELIZING RECURRENT/STATE-SPACE UPDATES IN PYTORCH/SNNTORCH.

Tool	Concept	Best for	snnTorch support
Iterative rollout	Sequential recurrence: $x_{t+1} = f(x_t, u_t)$	Exact dynamics; neuromorphic deployment	✓ Baseline
Convolution	Time-invariant linear filter: $y = h * u$	Fast parallel training with fixed decay	✓ `StateLeaky`
Prefix scan	Associative multiply–add: $x_t = \mathrm{diag}(\alpha_t)x_{t-1} + u_t$	Input-dependent or time-varying decay	✓ `AssociativeLeaky`
Associative scan	General associative operator: $(x, u) \oplus (x', u')$	Nonlinear recurrences	× Needs JAX/Triton

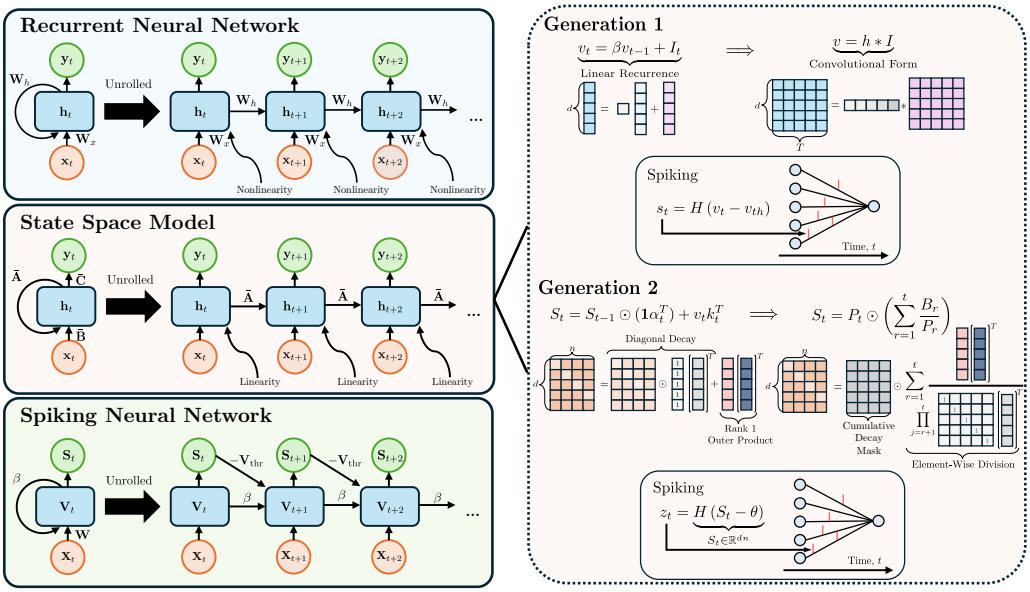

Fig. 1. Left: Unrolled dynamics of recurrent neural networks (RNNs), state space models (SSMs), and spiking neural networks (SNNs), highlighting their structural similarities and differences. Right: Two generations of spiking state-space formulations: Generation 1 (convolutional linear recurrence) and Generation 2 (prefix-scan associative recurrence), each producing spike outputs.

B. StateLeaky as a Generation 1 spiking SSM

Comparing Equation (5) with the canonical SSM form in Equation (1), we identify $x_t \leftrightarrow \tilde{v}_t$, $u_t \leftrightarrow I_t$, and $A \leftrightarrow \mathrm{diag}(\beta)$. This is a diagonal, time-invariant state update with an exponential impulse response.

In a batched setting with inputs $X \in \mathbb{R}^{T\times B\times C}$ (sequence length T, batch size B, channels C), which are permuted as needed for convolution, we form per-channel exponential kernels:

$$h^{(c)} = [1,\, \beta_c,\, \beta_c^2,\, \ldots,\, \beta_c^{T-1}],$$

collected into $h \in \mathbb{R}^{C\times 1\times T}$. The reset-free membrane is then computed by grouped 1D convolution:

$$\tilde{v} = \mathrm{Conv1d}_{\mathrm{groups}=C}(I, h),$$

where $I = WX$ is the input current tensor. The entire sequence is evaluated with one convolution call rather than a loop over t.

C. Implementation in SNNTORCH

The convolutional form is implemented in SNNTORCH as `StateLeaky`, which evaluates the reset-free membrane dynamics via grouped 1D convolution and returns membrane traces and spikes for full input sequences shaped as (T, B, C). A convenience wrapper, `LinearLeaky`, pairs a learned affine projection with the `StateLeaky` update, matching the common "linear layer + LIF dynamics" pattern. Both modules use the parallel convolutional evaluation during training; after training, the original iterative dynamics can be restored for neuromorphic hardware deployment.

IV. STATELEAKY: EMPIRICAL EVALUATIONS

We begin by treating the standard `Leaky` neuron model and the proposed `StateLeaky` model as drop-in recurrent layers and ask a simple question: how do these scale as we increase sequence length?

Two quantities are reported. Runtime is measured for a full forward–backward training step using synchronized GPU timers, so that we capture the actual wall-clock cost of one optimization update. Activation memory is measured as the difference between the peak (allocated) and baseline (idle) memory; which isolates the memory used by model states and saved activations from the fixed overhead of the framework and device. We also show how much activation-memory of the convolutional recurrence can be recovered in practice.

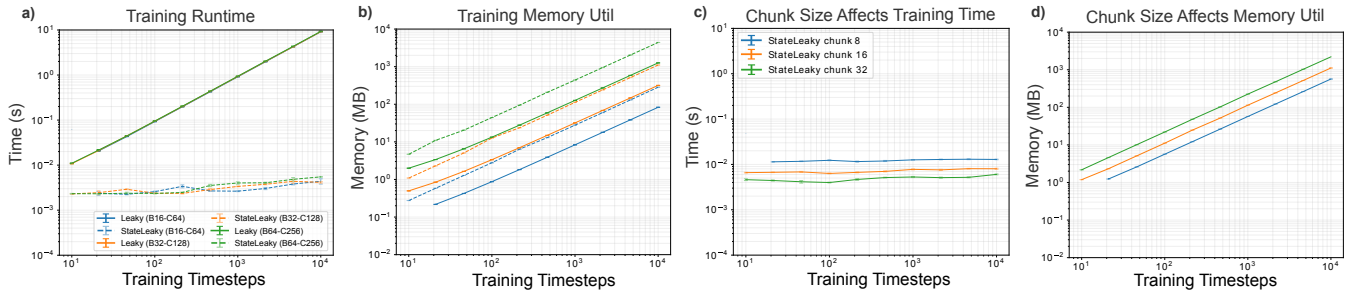

Fig. 2. Runtime and activation-memory scaling for `Leaky` (sequential recurrence) and `StateLeaky` (parallel convolutional recurrence). Panel (a) plots training-time over sequence length for `StateLeaky` vs `Leaky`. Panel (b) depicts the same conditions for (a) but plots memory utilization. Panel (c) measures the effect of gradient-accumulation chunk size on `StateLeaky` training runtime. Panel (d) reports the corresponding memory results for the training chunk-size sweep in (c). Experiments a) and b) sweep batch–channel configurations $(B, C) \in \{(16, 64), (32, 128), (64, 256)\}$. Experiments c) and d) operate on a $(B, C) \in (64, 256)$ Error bars denote standard deviation over ten runs.

A. *Scaling benefits of parallel `StateLeaky` evaluation*

Fig. 2(a) starts with the most basic question: how long does it take to run these neurons as we stretch the sequence length T? For the standard `Leaky` neuron, training runtime grows almost linearly in T, as expected from a loop that processes one timestep after another. `StateLeaky` replaces this loop with a single convolution over the sequence, and its curves in panel (a) rises much more slowly. As T grows, the gap between the two implementations widens. For example, at 1000 steps `StateLeaky` is 1000 times faster. Importantly, there is no visible penalty at short sequence lengths: `StateLeaky` stays competitive with `Leaky` even when T is small.

Fig. 2(b) makes the cost of this speedup visible in terms of memory, reporting peak activation memory during training. `Leaky` peak memory grows linearly with T, as intermediate tensors need to be retained as T is rolled out, and the computation graph is built. `StateLeaky`, by contrast, materializes the full sequence as a dense activation tensor and runs a 1D convolution over time. Its training-time memory therefore grows roughly linearly with T and sits noticeably above the `Leaky` baseline across all (B, C) configurations. Taken together, panels (a)-(b) show a simple trade-off: the convolutional recurrence buys much better runtime scaling, but it comes with a larger, more T-dependent memory footprint.

Once that trade-off is clear, the natural next step is to ask whether we can reduce the memory cost without changing the model itself. Fig. 2(c) and (d) explore one such knob: batch-wise gradient accumulation. Panel (d) varies the chunk size used in `StateLeaky` for gradient accumulation. Smaller chunks reduce the effective batch processed at once and cut peak activation memory by almost an order of magnitude in the largest configuration, bringing `StateLeaky` much closer to the `Leaky` baseline at long sequence lengths.

Fig. 2(c) then checks the runtime-cost side of this adjustment by plotting training runtime for the same chunk sizes. The curves move only slightly as chunk size changes, indicating that chunking recovers most of the memory overhead while introducing very little extra compute time. In practice, this means that `StateLeaky` can keep its parallel convolutional recurrence for speed, and gradient-accumulation chunking provides a straightforward way to make the memory footprint manageable on large, long-context workloads.

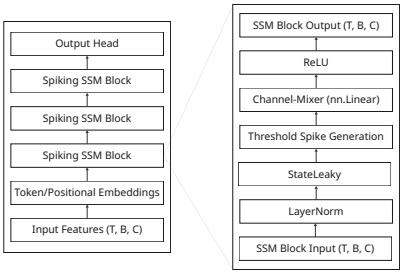

Fig. 3. Architecture used in the TinyStories benchmark. Each block applies layer normalization, a `StateLeaky` update, threshold spike generation, a channel mixer (`nn.Linear`), and a ReLU.

B. *TinyStories Language-Modeling Evaluation*

The synthetic benchmarks tell us how the two neuron models behave in isolation, but they leave out the question that usually matters most: what happens once we drop them into a real sequence model and train end to end? To probe this, we built a small spiking language model and trained it on the TinyStories dataset [19].

TinyStories is a convenient test bed: sequences are long enough to put pressure on the recurrence, but the model itself can stay modest. At a high level, the architecture follows a standard language–modeling pattern. Tokens are embedded, a positional embedding is added, and the resulting sequence of vectors is passed through a stack of identical blocks. Each block has two roles. Along the time axis it uses either a `Leaky` or `StateLeaky` update as an *information mixer*, deciding how past inputs influence the current state. Along the channel axis it applies a learned linear layer as a *channel mixer* to combine features at each timestep. Around these components we apply layer normalization, generate spikes by thresholding the membrane, and insert a ReLU nonlinearity. The full block is shown in Fig. 3. All runs use a fixed sequence length of

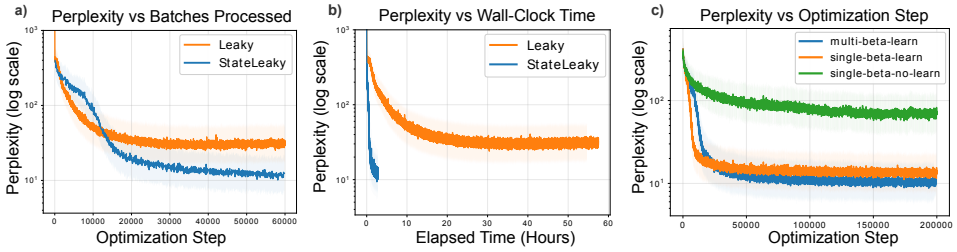

Fig. 4. TinyStories language–modeling benchmark. (a) Validation perplexity versus optimization step. Because the reset mechanism is removed during training, `StateLeaky` exhibits slower initial per-step improvement than the standard `Leaky` recurrence. (b) Perplexity versus wall-clock time. Parallel convolutional evaluation yields substantially faster updates, allowing `StateLeaky` to reach better perplexity sooner in real time despite its initially slower per-step trajectory. (c) Ablation of the decay parameter β. Learning β improves optimization dynamics, and learning a separate β per channel yields the strongest performance, indicating that richer temporal transition structure provides a measurable modeling benefit.

$T = 512$ tokens, Adam optimizer, a learning rate of 10^{-3}, and trained on a Nvidia A100 40GB instance.

Figures 4(a) and (b) show the same training runs, plotted against two different horizontal axes. Panel (a) tracks validation perplexity versus optimization step. Early in training, the standard `Leaky` neuron has a clear per-step advantage: its curve drops faster than that of `StateLeaky`. This is consistent with the extra temporal nonlinearity it carries. Because the reset term remains inside the recurrence, the membrane depends on both the linear leak–plus–input term and a history of discrete reset events, giving the model more ways to shape how information is stored and cleared over time. `StateLeaky`, by contrast, removes the reset during training so that the membrane follows a strictly linear state–space update; the only nonlinearity touching the temporal state is the spike threshold. In our runs this shows up as a slower initial drop in perplexity, with the `StateLeaky` curve trailing the `Leaky` curve at the beginning. As training continues, however, the situation reverses: `Leaky` saturates at a higher perplexity, while `StateLeaky` keeps improving and ultimately finishes with the better step-matched validation score.

Panel (b) replots the same trajectories against wall-clock time instead of step count. This view makes the systems effect of the convolutional recurrence explicit. `StateLeaky`'s faster training steps compound with the eventual advantage in step-matched perplexity, so the `StateLeaky` model reaches any given validation value earlier in real time. Over the full training window, it consistently achieves lower perplexity than `Leaky` when the two are compared on the axis of wall-clock times.

Only the temporal recurrence is linearized in `StateLeaky`. The rest of the block still contains nonlinearity through spike generation, channel mixing, and the final projection stack. In the TinyStories setting this combination appears sufficient: removing the reset from the recurrence changes the early learning dynamics, but the model still fits the task well, and over time the parallel evaluation lets it achieve both better step-matched perplexity and better wall-clock efficiency. For practitioners, the takeaway is that inside a standard language-model-style block, `StateLeaky` can stand in for `Leaky` while preserving modeling quality and delivering the system-level speedups seen in the synthetic benchmarks, visible in both per optimization step and in real time.

Finally, Fig. 4(c) examines how the choice of temporal decay parameterization affects performance within the `StateLeaky` model. Holding the architecture and training setup fixed, we compare a fixed global decay, a single learned decay shared across channels, and learned per-channel decays. Allowing the decay to be learned improves both optimization speed and final validation performance, and introducing per-channel decays yields a further, consistent gain.

These results motivate the next step in the design. If a modest increase in expressiveness—from a fixed β to a small vector of learned per-channel decays—already brings clear gains on TinyStories, then enriching the hidden state and its update rule should help further.

V. AssociativeLeaky: A Generation 2 Spiking State-Space Neuron Model

A. Motivation: From Diagonal StateLeaky to Matrix–State AssociativeLeaky

The `StateLeaky` neuron from Sec. III solves one problem very cleanly. It gives a fully parallel temporal pathway and, as the benchmarks show, removes the strict $O(T)$ runtime penalty of a step-by-step rollout.

At the same time, the hidden state in `StateLeaky` remains very simple. It is a *vector* of leaky traces, one per channel, each updated with its own scalar decay. Channels do not interact inside the recurrence, and every dimension of the state behaves like an independent fading memory of the past inputs. This is exactly what we want for many tasks, but it is a poor fit for settings where the model needs to store and later retrieve *associations* between pieces of information (for example, which key goes with which value, or which token should be recalled when a particular cue appears).

State-space models that sit one step beyond this diagonal case often add two ingredients: a *matrix-valued* state and input-dependent decays. The state now looks more like a small table than a flat vector, and inputs can write rank-1 updates into this table along directions determined by a key and value. With the right choice of update, this still yields a recurrence

that is linear through time, but it behaves much more like a simple associative memory: information can be written to and read from different "locations" in the state.

Our goal in the rest of this section is to build a spiking neuron that follows this pattern while staying close to the semantics of SNNTORCH. Instead of a single vector trace, the membrane state becomes a matrix S_t that is updated in two steps at each timestep: (i) an input-dependent decay that scales each column, and (ii) a rank-1 write given by an outer product of a value vector and a key vector derived from the current input. The nonlinearity from spikes (thresholding and surrogate gradients) remains outside this update, so the recurrence itself stays linear in time and can be evaluated using the prefix-scan tools from Table I.

We refer to this neuron as `AssociativeLeaky`. Conceptually, it extends `StateLeaky` from a bank of independent leaky traces to a small, trainable associative memory that still fits within the linear-through-time state-space view and remains parallelizable in PyTorch.

B. AssociativeLeaky: Matrix–State Spiking State–Space Recurrence

The hidden state is a matrix $S_t \in \mathbb{R}^{d\times n}$, interpreted as the membrane of $d \times n$ spiking neurons. Each timestep receives input $x_t \in \mathbb{R}^D$, projected into:

$$v_t = W_v x_t \in \mathbb{R}^d, \quad k_t = W_k x_t \in \mathbb{R}^n,$$
$$\alpha_t = \sigma(W_\alpha x_t) \in (0,1)^n.$$

Here v_t and k_t are value and key vectors, while α_t supplies input-conditioned decay applied columnwise.

Combining gated decay with a rank-1 outer-product write yields:

$$S_t = S_{t-1}\,\mathrm{diag}(\alpha_t) + v_t k_t^\top, \tag{7}$$

matching the Generation 2 form in Equation (3). Each column of S_{t-1} is scaled by its decay factor, and the outer product deposits v_t along the key direction $k_t^\top$. Because each column evolves as an independent multiply–add recurrence, the full matrix admits parallel evaluation via prefix products and sums.

Spikes are generated elementwise:

$$z_t = H(S_t - \theta). \tag{8}$$

Surrogate gradients are applied only to z_t and do not alter the dynamics of the recurrence itself.

Query-based readout. When `AssociativeLeaky` is used as an internal SSM block, the matrix state may be converted back into a vector representation through an optional query projection. Given x_t, a query matrix

$$Q_t = \mathrm{reshape}(W_q x_t) \in \mathbb{R}^{n\times d}$$

is computed, and the readout is

$$y_t = S_t Q_t \in \mathbb{R}^{d\times d}.$$

This corresponds to querying the associative memory stored in S_t along directions determined by Q_t, mirroring the use of query vectors in Generation 2 SSMs. If query projection is disabled, the flattened membrane (or spike map) $\mathrm{vec}(S_t)$ is returned instead.

VI. ASSOCIATIVELEAKY: EMPIRICAL EVALUATIONS

A. Systems-level scaling compared to StateLeaky and Leaky

The diagonal `StateLeaky` neuron already gives us a fast, parallel alternative to the sequential `Leaky` recurrence. The natural follow-up is to ask what happens when we move to the richer matrix-state `AssociativeLeaky`: do we keep the same long-sequence behavior, and what price do we pay compared to both `StateLeaky` and `Leaky` on a real GPU?

Panels (a) and (b) of Fig. 5 answer this at the training step level for a fixed $(B, C) = (64, 256)$ configuration. Panel (a) plots training runtime as a function of sequence length T. The sequential `Leaky` neuron again slows down almost linearly in T, as each timestep is processed in order. Both `StateLeaky` and `AssociativeLeaky` replace that loop with parallel operations over the whole sequence, and their runtime rises much more gently with T. Across the whole range of sequence lengths, the two parallel neurons follow a similar trend: `AssociativeLeaky` sits above `StateLeaky` by a stable margin, reflecting its extra projections (W_v, W_k, W_α, W_q), outer-product writes, and prefix-scan machinery, but it keeps the same "slow rise with T" behavior that makes `StateLeaky` attractive for long contexts.

Panel (b) shows the corresponding peak activation memory during backpropagation. All three neurons show clear growth with T, even `Leaky`, which is expected once intermediate activations must be stored for the backward pass. The curves for `StateLeaky` and `AssociativeLeaky` have similar slopes: both rely on parallel recurrences that carry information across all timesteps, and both pay for that by keeping more activations in memory as T increases. `AssociativeLeaky` lies above `StateLeaky` due to the aforementioned extra projections, but the shape of the curve is the same. Compared to the original `Leaky` neuron, both parallel variants use more memory at a given T, but they do so in a controlled, roughly linear way.

In summary, `AssociativeLeaky` mirrors `StateLeaky` system behavior: slightly heavier per step due to its richer state, but preserving favorable scaling with T and remaining far more practical than sequential rollout for long sequences.

B. TinyStories language–modeling evaluation

Once the systems picture is clear, the next question is whether the matrix–state recurrence actually buys anything in a real model, beyond being a more expensive version of `StateLeaky`. To test this, we return to the TinyStories language–modeling setup and keep everything the same except for the choice of spiking neuron.

The architecture is identical to the one used in the `StateLeaky` experiments (Fig. 3). The only change is that the internal recurrence is now implemented with `AssociativeLeaky` instead of `StateLeaky`.

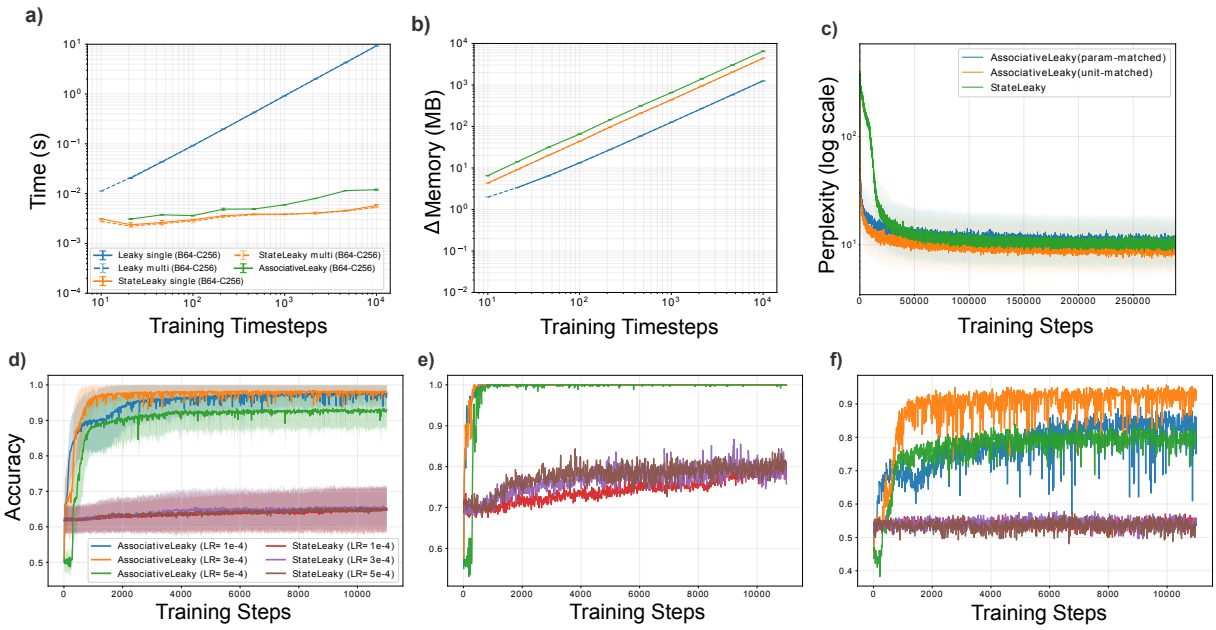

Fig. 5. Benchmarks for the proposed `AssociativeLeaky` neuron compared to the diagonal `StateLeaky` baseline. (a) Training-time runtime versus sequence length for all neuron models. Leaky and StateLeaky show both a single beta and a per-channel beta. (b) Peak activation memory during training versus sequence length. Panels (a,b) extend the Gen1-style scaling study to include the richer matrix-state recurrence of `AssociativeLeaky`. Both state and unit matched `AssociativeLeaky` are plotted. (d) Mean bit-accuracy $\pm$ one standard deviation on the Neural Turing Machines (NTM) Associative Recall task across evaluation batches, shown for multiple learning rates. (e) Maximum accuracy across evaluation batches. (f) Minimum accuracy across evaluation batches for the same experiment. Panels (e,f) visualize the full range of accuracy variability on the associative-memory task.

A key detail for interpreting this comparison is what we hold constant. `AssociativeLeaky` adds learned structure (key/value/decay projections and a readout), so there is no single notion of "matched size." In Fig. 5(c) we therefore report two regimes. In the *unit-matched* setting we match the total number of spiking neurons between models. In the *parameter-matched* setting we match the total number of learned parameters by shrinking the matrix-state configuration.

Figure 5(c) shows the resulting validation perplexity curves. Under the unit-matched comparison, `AssociativeLeaky` reaches the lowest perplexity and remains below the diagonal `StateLeaky` baseline across training. This is the regime most aligned with the neuromorphic setting we care about: if the dominant cost is the number of active spiking units and their spike-driven communication, then spending extra parameters per neuron can be acceptable when it improves modeling quality.

Under the parameter-matched comparison, however, the trend reverses. When `AssociativeLeaky` is constrained to the same parameter budget as `StateLeaky`, its perplexity is slightly higher than the diagonal baseline. Therefore matrix–state recurrence is not a pure inductive-bias win on this language workload under tight parameter budgets. Instead, its advantage shows up in the unit-matched regime, where we hold spiking activity capacity fixed and allow the richer state update to use additional learned structure.

This suggests a more precise interpretation: `AssociativeLeaky` is best viewed as an architectural knob for injecting *associative-memory capacity* when that capacity is worth paying for, rather than a drop-in replacement that should dominate across all tasks and all budget regimes. To probe that regime directly, we next evaluate both neurons on a benchmark where key–value style storage and content-based retrieval are the limiting factor.

C. Associative memory benchmark: NTM Associative Recall

The two neuron models explored thus far, `StateLeaky` and `AssociativeLeaky` differ most sharply in how they treat state. A natural place to probe that difference is on a task that really needs associative memory, rather than just a long temporal receptive field. For this, we use a benchmark introduced in *Neural Turing Machines* (NTM) [20], where the model must store several key–value bindings and, after a delay, return the value that matches a queried key.

Each training sequence is built from N key–value pairs and a final query. Keys and values are binary vectors of width W. They are presented as a single stream: one timestep for the key, one timestep for its value, repeated N times. A dedicated delimiter channel marks the end of the list, and the last timestep carries the query key whose associated value the model must output. Each input example therefore has length $(2N+2)$ and shape $(2N+2) \times B \times (W+1)$, where the extra channel encodes the delimiter bit. The target is a single value vector of shape $1 \times B \times W$, corresponding to the value bound to the query key.

Both `AssociativeLeaky` and `StateLeaky` are dropped into the same small sequence model: an input projection, one recurrent block, a two-layer MLP, a second recurrent block, and a final output projection. Hidden dimension, optimizer, batch size, and all other training details are kept fixed. For each learning rate in a small sweep, we evaluate on a held-out set and record bit-accuracy statistics across evaluation batches (mean, spread, and extrema).

Panels (d)–(f) of Fig. 5 show how the two neurons behave under this setup. `AssociativeLeaky` reaches high bit-accuracy across all batches, with low variance and minimum accuracy climbing towards 1.0; performance improves steadily not just on average, but on every sequence in the evaluation set. `StateLeaky` tells a different story. It can achieve good accuracy on a subset of batches (panel (e)), but the minimum batch-wise accuracy curve in panel (f) barely moves. A nontrivial fraction of sequences remains poorly solved even when the model does well on others.

This gap reflects a structural optimization mismatch. A diagonal, elementwise decay can only maintain independent leaky traces and has no internal mechanism for binding keys to values. It is therefore ill-suited to a task that explicitly demands associative storage and content-based retrieval. `AssociativeLeaky`, by contrast, implements rank-1 writes $v_t k_t^\top$ into a matrix-valued state with input-dependent decay, exactly the kind of structure needed to store and selectively access key–value pairs. The clean separation in Fig. 5(d)–(f) is consistent with this: among the two spiking SSMs tested here, `AssociativeLeaky` is the only one that truly solves the NTM Associative Recall task.

VII. Discussion

The two neurons in this paper trace a path through the state-space landscape: from a diagonal, single-timescale update (`StateLeaky`, Generation 1) to a matrix-valued associative memory (`AssociativeLeaky`, Generation 2), both written in the same language of linear dynamics and both admitting parallel evaluation over sequence length.

A structural constraint underlies both designs. To keep the recurrence amenable to convolution or prefix scans, we place no nonlinearity inside the time loop—the spike-triggered reset is removed during training, and spikes act on the membrane only after the linear update. This trades temporal nonlinearity for parallelism. The TinyStories experiments suggest that nonlinearities outside the recurrence (thresholding, channel mixing, output layers) are sufficient to recover useful modeling behavior, while the cheaper updates yield faster wall-clock training. Crucially, both neurons remain compatible with event-driven inference: parallel evaluation is a training-time optimization, and iterative dynamics can be restored for neuromorphic deployment.

Generation 3 SSMs couple decay and write operations through projectors like $(I - \beta_t k_t k_t^\top)$, requiring associative-scan kernels not yet available in PyTorch eager mode. These sit beyond the current SNNTORCH implementation, but represent a natural continuation once tooling catches up.

VIII. Conclusion

We introduced two time-parallel spiking neurons for SNNTORCH: a convolutional diagonal variant of Leaky/LIF dynamics, and a matrix-state neuron that supports associative key–value updates while remaining linear through time. Together, they outline a practical path toward spiking models that operate over long contexts without sacrificing representational capacity. Looking ahead, structured sparsity and future scan primitives in PyTorch could carry event-based computation deeper into the recurrence, pushing toward a future where spiking models inherit SSM scalability while retaining their low-power advantages.

References

[1] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[2] J. K. Eshraghian, M. Ward, E. Neftci, X. Wang, G. Lenz, G. Dwivedi, M. Bennamoun, D. S. Jeong, and W. D. Lu, "Training spiking neural networks using lessons from deep learning," 2023.

[3] E. O. Neftci, H. Mostafa, and F. Zenke, "Surrogate gradient learning in spiking neural networks," 2019.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[5] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, "Long range arena: A benchmark for efficient transformers," 2020.

[6] Y. Bai, X. Lv, J. Zhang, H. Lyu, J. Tang, Z. Huang, Z. Du, X. Liu, A. Zeng, L. Hou, Y. Dong, J. Tang, and J. Li, "Longbench: A bilingual, multitask benchmark for long context understanding," 2024.

[7] A. Voelker, I. Kajić, and C. Eliasmith, "Legendre memory units: Continuous-time representation in recurrent neural networks," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[8] A. Gu, K. Goel, and C. Ré, "Efficiently modeling long sequences with structured state spaces," 2022.

[9] J. T. H. Smith, A. Warrington, and S. W. Linderman, "Simplified state space layers for sequence modeling," 2023.

[10] A. Gu and T. Dao, "Mamba: Linear-time sequence modeling with selective state spaces," 2024.

[11] B. Peng, E. Alcaide, Q. Anthony, A. Albalak, S. Arcadinho, S. Biderman, H. Cao, X. Cheng, M. Chung, M. Grella, K. K. GV, X. He, H. Hou, J. Lin, P. Kazienko, J. Kocon, J. Kong, B. Koptyra, H. Lau, K. S. I. Mantri, F. Mom, A. Saito, G. Song, X. Tang, B. Wang, J. S. Wind, S. Wozniak, R. Zhang, Z. Zhang, Q. Zhao, P. Zhou, Q. Zhou, J. Zhu, and R.-J. Zhu, "Rwkv: Reinventing rnns for the transformer era," 2023.

[12] D. Y. Fu, T. Dao, K. K. Saab, A. W. Thomas, A. Rudra, and C. Ré, "Hungry hungry hippos: Towards language modeling with state space models," 2023.

[13] M. Poli, S. Massaroli, E. Nguyen, D. Y. Fu, T. Dao, S. Baccus, Y. Bengio, S. Ermon, and C. Ré, "Hyena hierarchy: Towards larger convolutional language models," 2023.

[14] W. Gerstner, W. M. Kistler, R. Naud, and L. Paninski, *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition*. Cambridge University Press, 2014.

[15] W. Fang, Z. Yu, Z. Zhou, D. Chen, Y. Chen, Z. Ma, T. Masquelier, and Y. Tian, "Parallel spiking neurons with high efficiency and ability to learn long-term dependencies," 2024.

[16] Y. Li, Y. Sun, X. He, Y. Dong, D. Zhao, and Y. Zeng, "Parallel spiking unit for efficient training of spiking neural networks," 2024.

[17] M. Bal and A. Sengupta, "P-spikessm: Harnessing probabilistic spiking state space models for long-range dependency tasks," 2025.

[18] D. Wang, R.-J. Zhu, S. Abreu, Y. Shan, T. Kergan, Y. Pan, Y. Chou, Z. Li, G. Zhang, W. Huang, and J. Eshraghian, "A systematic analysis of hybrid linear attention," 2025.

[19] R. Eldan and Y. Li, "Tinystories: How small can language models be and still speak coherent english?," 2023.

[20] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," 2014.