HYBRID TEMPORAL CREDIT ASSIGNMENT VIA INTERNAL AND EXTERNAL RECURRENCE DECOMPOSITION

Anonymous authors
Paper under double-blind review

ABSTRACT

Recurrent neural networks process sequential input through their internal dynamics, but deploying them in neuromorphic settings requires training methods that are both memory efficient and capable of accurate temporal credit assignment. Backpropagation through time (BPTT) computes exact gradients, but its memory demands rise linearly with sequence length, making it unsuitable for long sequences and on-chip learning. Truncated BPTT (TBPTT) approximates BPTT while reducing memory demands, but it discards long-range dependencies. Approximate forward-propagation methods such as e-prop and HYPR, on the other hand, can retain long-range gradient information, but they restrict gradient computations to neuron-internal recurrence while neglecting pathways through recurrent connections. We propose Forward-Backward Propagation Through Time (FBPTT), an algorithm that computes gradients by combining forward propagation of neuron-internal sensitivities through the entire sequence with TBPTT within sequence chunks. Forward-propagated sensitivities preserve long-range learning signals, while TBPTT captures short-term dependencies across recurrent connections. This results in a semi-online algorithm with chunk-wise updates and bounded memory. We evaluate FBPTT on both standard artificial recurrent networks and spiking neural network models. Across all settings, FBPTT consistently outperforms pure forward approximations and standard TBPTT, showing that the combination of long-term neuron-internal credit with short-term intralayer credit is an effective strategy for scalable semi-online learning.

KEYWORDS

Recurrent Neural Networks, Temporal Credit Assignment, Spiking Neural Networks, Neuromorphic Computing, Backpropagation Through Time, Real-Time Recurrent Learning

INTRODUCTION

Temporal credit assignment is the central challenge of learning on sequential inputs. Recurrent neural networks address this by maintaining evolving hidden states, but no training algorithm has yet resolved the resulting tension between gradient accuracy, memory efficiency, and online applicability. Backpropagation through time (BPTT) (Werbos, 1990) resolves this tension in favor of accuracy: it computes exact gradients by storing and traversing the full unrolled computational graph. The cost grows linearly with sequence length, making it un-scalable for continual learning and neuromorphic settings (Zenke & Neftci, 2020). Truncated BPTT (TBPTT) (Williams & Peng, 1990) regains efficiency by processing sequences in fixed-length chunks, but at the price of biased gradients: any dependency crossing a chunk boundary is simply ignored.

The alternative family of online algorithms avoids backward unrolling entirely. Real-time recurrent learning (RTRL) (Williams & Zipser, 1989) propagates exact state-parameter Jacobians forward in time, achieving true online learning but at a cost of $\mathcal{O}(n^4)$ time and $\mathcal{O}(n^3)$ memory per step, where n is the number of hidden units in the recurrent network.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

This has motivated a range of approximate forward methods (Marschall et al., 2020) that trade Jacobian completeness for scalability, including low-rank and Kronecker-factored approximations of the influence matrix (Tallec & Ollivier, 2017; Mujika et al., 2018) and fixed diagonal approximations such as RFLO (Murray, 2019), e-prop (Bellec et al., 2020), and HYPR (Baronig et al., 2026). The latter restrict gradient flow to neuron-internal dynamics and discard intra-layer recurrent pathways. This restriction is the key limitation we address. Intra-layer recurrent connections carry information about how neurons within a layer influence one another's dynamics. Discarding this pathway during training, as existing approximate forward methods do, means the resulting gradients fail to reflect these interactions, and the network loses information that is not recoverable from neuron-internal dynamics alone. Neuron-internal recurrence captures how each neuron accumulates its own state over time, a signal that is quite distinct from the interaction effects carried between neurons.

We argue that both credit paths carry information that training should preserve. Our method, which we call Forward-Backward Propagation Through Time (FBPTT), forward-propagates neuron-internal sensitivities across chunk boundaries so that this information is not lost when memory is bounded, while applying TBPTT locally within each chunk to consider the intra-layer recurrent gradients that pure forward methods discard. The resulting algorithm is semi-online, fully sequential in time, with bounded memory, which recovers both sources of information instead of sacrificing one for the other. We instantiate this framework for both, artificial recurrent networks, including LSTMs (Hochreiter & Schmidhuber, 1997), GRUs (Chung et al., 2014), and HORN (Effenberger et al., 2025), and spiking networks (Higuchi et al., 2024; Baronig et al., 2025).

Our contributions are:

- A structural decomposition of recurrent gradient flow into neuron-internal and intra-layer recurrence, motivating different credit assignment to each so that neither source of information is discarded during training.
- A semi-online learning algorithm FBPTT that combines forward-propagated neuron-internal sensitivities across chunks with local TBPTT within chunks, achieving bounded memory and improved gradient.
- A unified formulation covering both spiking and artificial recurrent architectures.
- Results showing FBPTT improves over pure forward methods and conventional TBPTT across multiple architectures and tasks.

Related Work

A broader class of online learning algorithms for RNNs has been systematically analyzed by (Marschall et al., 2020), which clarifies the relationships among several algorithms relevant to our setting. Real-Time Recurrent Learning (RTRL; (Williams & Zipser, 1989)) is the exact past-facing method: it recursively maintains the full influence matrix, which tracks how the current hidden state depends on all past parameter applications. Concretely, RTRL maintains a sensitivity matrix $P_t = \partial h_t / \partial \theta \in \mathbb{R}^{n \times n^2}$, where $h_t \in \mathbb{R}^n$ is the hidden state and θ has $\mathcal{O}(n^2)$ parameters (the recurrent weight matrix); storing P_t requires $\mathcal{O}(n^3)$ memory. At each step, updating P_t requires multiplying the $n \times n$ state Jacobian $\partial h_t / \partial h_{t-1}$ by P_{t-1}, which takes $\mathcal{O}(n^4)$ time. While exact, these $\mathcal{O}(n^3)$ memory and $\mathcal{O}(n^4)$ compute costs make RTRL impractical at scale. Approximations such as UORO (Tallec & Ollivier, 2017) and KF-RTRL (Mujika et al., 2018) reduce both costs to $\mathcal{O}(n^2)$ by compressing the influence matrix into low-rank or Kronecker-factored form, at the cost of introducing stochastic noise into the gradient estimates.

Truncated BPTT computes gradients by unrolling the network over a finite chunk λ, with memory scaling as $\mathcal{O}(n\lambda)$. Decoupled Neural Interfaces (DNI; (Jaderberg et al., 2017)) avoid explicit unrolling by learning a synthetic gradient module that predicts future credit assignment from the current hidden state, enabling fully online updates at the cost of approximation error in the predicted gradients. A simpler deterministic approximation, Random-Feedback Online Learning (RFLO; (Murray, 2019)), replaces the full Jacobian propagation in RTRL with a fixed diagonal approximation, reducing memory and compute to $\mathcal{O}(n^2)$ while retaining a closed-form update. Marschall et al. (2020) argue that RFLO is effective largely because it trains a high-capacity feedforward mapping over the hidden state's natural memory traces. A more bio-inspired version of RFLO is e-prop (Bellec et al., 2020), with its parallelizable version HYPR (Baronig et al., 2026).

A closely related line of work targets online training specifically for spiking neural networks (SNNs) by decomposing gradients into local temporal and spatial factors. Online Spatio-Temporal Learning (OSTL (Bohnstingl et al., 2020)) derives this decomposition directly from BPTT, maintaining a per-layer eligibility trace (local temporal information, same as RTRL's influence matrix but computed layer-locally) combined with a backpropagated learning signal at each time step. DECOLLE (Kaiser et al., 2020) pushes locality further by attaching each layer to a fixed random readout trained on a local target, computing gradients only within-layer and within-timestep. OTTT (Xiao et al., 2022) instead approximates standard surrogate-gradient BPTT by detaching the neuron-reset path so the temporal Jacobian reduces to a scalar leak term, enabling a forward-tracked presynaptic trace to be paired with an instantaneous spatial gradient at each step.

While various approximations to BPTT have been introduced, the study of the combination of backward- and forward-propagation methods has been surprisingly scarce. The combination has specifically been proposed for LSTMs (Weber, 2024). In this article, we generalize this idea to general stateful RNNs.

METHODS

BACKGROUND: EXACT AND APPROXIMATE GRADIENT COMPUTATION

We consider a recurrent network layer consisting of m neurons. Let $\mathbf{h}_i^t \in \mathbb{R}^k$ denote the k-dimensional hidden state of neuron i at time step t. We denote by $\mathbf{h}^t \in \mathbb{R}^{mk}$ the vector obtained by concatenating the k-dimensional state vectors of all m neurons in the layer. We can write the state transition function as $\mathbf{h}^t = f_\theta(\mathbf{h}^{t-1}, \mathbf{x}^t, \mathbf{y}^{t-1})$, where $\mathbf{x}^t \in \mathbb{R}^d$ is the external input at time t, $\mathbf{y}^{t-1} \in \mathbb{R}^m$ is the network output at the previous time step, and θ indicates the network parameters. The total loss over a sequence of length T is $L = \sum_{t=1}^T L^t$, where L^t denotes the loss at time step t. The exact parameter gradient can be computed in two equivalent ways, that is, in backward mode (BPTT) and in forward mode (RTRL).

Backward mode (BPTT): In BPTT, the gradient is computed as

$$\nabla_\theta L = \sum_{t=1}^{T} \boldsymbol{\delta}_t F^t, \qquad \boldsymbol{\delta}_t = \boldsymbol{\ell}^t + H^{t+1}\boldsymbol{\delta}_{t+1}, \tag{1}$$

where $F^t = \partial \mathbf{h}^t / \partial \theta^t$ is the immediate parameter influence, $\boldsymbol{\delta}_t = dL/d\mathbf{h}^t$ is the backpropagated error, and $H^t = d\mathbf{h}^t / d\mathbf{h}^{t-1}$ is the state Jacobian. Note that we distinguish in our notation between total derivatives dx/dy and partial derivatives $\partial x/\partial y$, see Appendix A.1. Here $\boldsymbol{\delta}_t$ is computed recursively backward in time: it combines the immediate loss gradient $\boldsymbol{\ell}^t = \partial L^t / \partial \mathbf{h}^t$, which captures how the loss incurred at time t depends on $\mathbf{h}^t$ directly, with the future error $\boldsymbol{\delta}_{t+1}$ propagated back through the state Jacobian H^{t+1}, which captures how $\mathbf{h}^t$ influences all subsequent losses through $\mathbf{h}^{t+1}$. The recursion is initialized at $\boldsymbol{\delta}_{T+1} = 0$ and proceeds from $t = T$ down to $t = 1$, so the full forward trajectory must already be available before the backward sweep can begin. This requires storing the full unrolled computation graph, giving $\mathcal{O}(nT)$ memory, where n is the total hidden state size, which in our formulation is $n = mk$. See Appendix A.2.1 for the full derivation.

Truncated BPTT: A common approximation divides the sequence into chunks of length λ and backpropagates only within each chunk, discarding gradient contributions that cross chunk boundaries. This reduces the memory demand to $\mathcal{O}(n\lambda)$, but it introduces a truncation error that grows with the strength of long-range dependencies, see Figure 1 (red arrows). We formalize this approximation, and quantify exactly what it discards, in Appendix A.3.1.

Forward mode (RTRL): Alternatively to Equation (1), the gradient can be computed as

$$\nabla_\theta L = \sum_{t=1}^{T} \boldsymbol{\ell}^t G^t, \qquad G^t = H^t G^{t-1} + F^t, \tag{2}$$

where $G^t = d\mathbf{h}^t / d\theta$ is the sensitivity matrix, propagated forward in time. It combines the propagated past sensitivity $H^t G^{t-1}$, which carries forward everything the parameters have already influenced through the state Jacobian H^t, with the new immediate influence F^t of the parameters on $\mathbf{h}^t$

at the current step. The recursion is initialized at $G^0 = 0$ and proceeds forward from $t = 1$ to $t = T$, so each step's gradient contribution $\boldsymbol{\ell}^t G^t$ can be computed immediately without waiting for future time steps. This enables true online learning without storing the past, but G^t scales with the number of parameters, giving $\mathcal{O}(n^4)$ time complexity per step for n hidden units. See Appendix A.2.2 for the full derivation.

Approximate forward propagation: Stateful RNNs typically involve two distinct sources of recurrence: a neuron-internal recurrence through each neuron's own state $\mathbf{h}_i^{t-1}$ (e.g., membrane potential decay in a spiking neuron, or the cell state in an LSTM), and an intra-layer recurrence arising from the recurrent connections between neurons in the layer, mediated by the layer output $\mathbf{y}^{t-1}$ (Figure 1). We make this explicit by rewriting the state transition as

$$\mathbf{h}_i^t = f^{\text{combine}}\big(f^{\text{I}}_{\theta_i^{\text{I}}}(\mathbf{h}_i^{t-1}),\ f^{\text{in}}_{\theta_i^{\text{in}}}(\mathbf{x}^t),\ f^{\text{E}}_{\theta_i^{\text{E}}}(\mathbf{y}^{t-1})\big), \tag{3}$$

where f^{I}, f^{in}, and f^{E} denote the internal, input-driven, and external state transition functions, respectively. Crucially, f^{I} depends only on the neurons own previous state, so it does not mix information across neurons, whereas f^{E} depends on the outputs $\mathbf{y}^{t-1}$ of *all* neurons in the layer via the recurrent weights. This separation carries over to the state Jacobian: since $\mathbf{y}^{t-1}$ depends on $\mathbf{h}^{t-1}$, the chain rule splits H^t into an internal and an external part,

$$H^t = \frac{d\mathbf{h}^t}{d\mathbf{h}^{t-1}} = \underbrace{\frac{\partial \mathbf{h}^t}{\partial \mathbf{h}^{t-1}}}_{H^{I,t}} + \underbrace{\frac{\partial \mathbf{h}^t}{\partial \mathbf{y}^{t-1}} \frac{\partial \mathbf{y}^{t-1}}{\partial \mathbf{h}^{t-1}}}_{H^{E,t}}, \tag{4}$$

such that $H^t = H^{I,t} + H^{E,t}$. Because f^{I} acts independently per neuron, $H^{I,t}$ is block-diagonal (Appendix A.5.2) and cheap to propagate; $H^{E,t}$ is dense, since every neuron can influence every other through the recurrent weights, which makes exact RTRL expensive. Thus, approximate forward propagation algorithms such as e-prop and HYPR only forward-propagate $H^{I,t}$. Substituting into the RTRL recursion $G^t = H^t G^{t-1} + F^t$ and dropping the external term yields the partial sensitivity matrix we track forward in time (see also Appendix A.3.2)

$$\tilde{G}^t = H^{I,t}\tilde{G}^{t-1} + F^t, \qquad \tilde{G}^0 = 0. \tag{5}$$

This reduces the computation to a manageable time complexity of $\mathcal{O}(n^2)$ per time step, but it completely neglects the gradients through the recurrent connections, see Figure 1 (blue arrows).

Forward-Backward Propagation Through Time (FBPTT)

The simple idea of FBPTT is to forward-propagate only the partial sensitiviy matrix $\tilde{G}^t$, and recover the contribution of $H^{E,t}$ locally in time via truncated backpropagation within each chunk. This approach is able to track long-range dependencies through neuron-internal dynamics while taking gradients through recurrent connections into account within the chunk. Our proposed gradient estimator for a sequence split into k chunks of length λ is (full derivation in Appendix A.4)

$$\nabla_\theta L \approx \underbrace{\sum_{i=0}^{k-1} \sum_{t=i\lambda+1}^{(i+1)\lambda} \boldsymbol{\delta}_t^{i\lambda+1:(i+1)\lambda} F^t}_{\text{truncated BPTT (local)}} + \underbrace{\sum_{i=1}^{k-1} \boldsymbol{\delta}_{i\lambda+1}^{i\lambda+1:(i+1)\lambda} H^{I,i\lambda+1} \tilde{G}^{i\lambda}}_{\text{long-term correction}}, \tag{6}$$

where the sum over i is summing over chunks. $\boldsymbol{\delta}_t^{r:s}$ generalizes $\boldsymbol{\delta}_t$, which is the derivative of the total loss L w.r.t. hidden state $\mathbf{h}^t$, to the derivative of the summed losses over time steps r to s w.r.t. $\mathbf{h}^t$, see Equation (A15). Hence, $\boldsymbol{\delta}_t^{i\lambda+1:(i+1)\lambda}$ accounts for losses in chunk i and $\boldsymbol{\delta}_{i\lambda+1}^{i\lambda+1:(i+1)\lambda}$ is the boundary error signal obtained from backpropagating within chunk $i{+}1$. The first term is standard truncated BPTT, capturing short-range dependencies. The second term corrects for long-range dependencies by injecting the accumulated internal-dynamics sensitivity $\tilde{G}^{i\lambda}$ at each chunk boundary, see Figure 1 (red and blue arrows).

The internal Jacobians $H^{I,t}$ and $\tilde{G}^t$ are both block-diagonal, which allows the correction term to be computed efficiently. Efficient, low-memory representations of the underlying F^t and $\tilde{G}^t$ matrices are given in Appendix A.5.1 and A.5.2. The pseudocode is provided in Appendix A.6. In Appendix A.7, we compare FBPTT's time and memory complexity against full BPTT, RTRL, TBPTT, and HYPR.

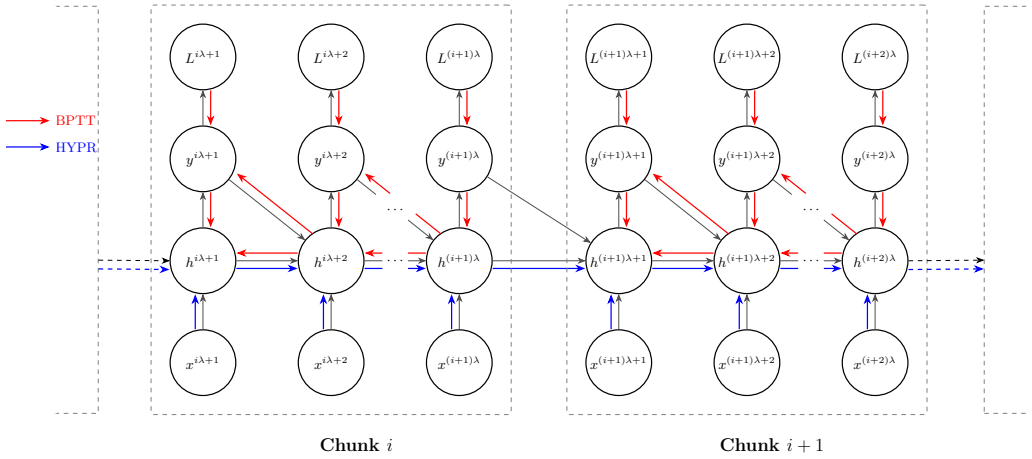


Figure 1: Computational graph of a stateful RNN (black arrows) divided in chunks of length λ. Red: Schematic of the backpropagation of $\boldsymbol{\delta}$ in TBPTT (Eq. (1)). Note that errors are not backpropagated between chunk boundaries. Blue: Schematic of forward-propagation of $\tilde{G}$ in HYPR (Eq. (5)). Note that it neglects recurrent connections, but crosses chunk boundaries. FBPTT combines both gradient paths (red and blue) accounting for both, long-range dependencies through neuron-internal recurrencies and gradients through recurrent connections.

Parts of the algorithm can be parallelized over the time dimension. $\tilde{G}^{\lambda}$ is obtained via cumulative products $\Phi_{(\lambda:t)} = \prod_{s=t}^{\lambda} H^{I,s}$, which can be computed in parallel across time using an associative scan (Blelloch, 1990), with $\mathcal{O}(\log \lambda)$ time complexity and $\mathcal{O}(\lambda)$ memory, avoiding the sequential $\mathcal{O}(\lambda)$ recursion. We elaborate on parallelization in Appendix A.5.3.

Experiments

FBPTT applied to recurrent ANN models

We first evaluated FBPTT on standard recurrent ANN architectures: LSTM networks and recurrent networks consisting of GRUs. In addition, we evaluated Harmonic Oscillator Recurrent Networks (HORN) (Effenberger et al., 2025). We included this less common network variant because it uses oscillating neurons, which is closely related to the SNN models considered below. Since the performance of HORN models was in general worse, we provide results for HORN in Appendix A.10. For evaluating these ANN models, we used three standard benchmark tasks: sMNIST, psMNIST, and sCIFAR (see Appendix A.9 for dataset descriptions). We compared FBPTT to BPTT on the one hand, which provides the true gradient, and to the two common approximation variants truncated BPTT (TBPTT) and approximate forward propagation. For the latter, we used HYPR (Baronig et al., 2026), which is an efficient parallel implementation of approximate forward propagation that neglects intra-layer recurrence in the gradient computation. Its computed gradient is equivalent to the one of e-prop (Bellec et al., 2020).

Table 1 summarizes the results for networks of GRUs. FBPTT outperformed HYPR and TBPTT across every task and both tested chunk numbers. For LSTMs (Table 2), FBPTT clearly outperformed HYPR. While the performance was comparable to TBPTT for 2 chunks, it exhibited better performance for 10 chunks. While HYPR performed comparably well on sMNIST for both architectures, it fell behind FBPTT and TBPTT alike on sCIFAR, indicating that the utilization of intra-layer gradients matters more in the latter task due to the task's richer spectral structure. Similarly, the gap between HYPR and the BPTT-based methods widens on psMNIST relative to sMNIST, indicating that the permutation of the input seqeunce and the resulting destroyed local temporal structure necessitate stronger intra-layer credit assignment that TBPTT and FBPTT provide.

Table 1: Test accuracy (%) of GRU networks on benchmark tasks (mean ± SD over 10 runs). Bold numbers indicate the best result out of {HYPR, 2-chunk TBPTT, 2-chunk FBPTT} and {HYPR, 10-chunk TBPTT, 10-chunk FBPTT}.

Method	**#chunks**	sMNIST	psMNIST	sCIFAR
BPTT	–	98.62 ± 0.19	93.81 ± 0.37	63.92 ± 0.71
TBPTT	2	98.17 ± 0.32	92.39 ± 0.22	59.61 ± 1.51
	10	98.03 ± 0.40	90.81 ± 1.03	52.74 ± 1.86
HYPR	–	94.91 ± 0.67	64.27 ± 1.48	35.76 ± 2.51
FBPTT	2	**98.49 ± 0.27**	**93.21 ± 0.56**	**62.31 ± 1.31**
(ours)	10	**98.42 ± 0.32**	**92.17 ± 0.83**	**58.43 ± 1.43**

Table 2: Test accuracy (%) of LSTM networks on benchmark tasks (mean ± SD over 10 runs). Bold numbers indicate the best result out of {HYPR, 2-chunk TBPTT, 2-chunk FBPTT} and {HYPR, 10-chunk TBPTT, 10-chunk FBPTT}.

Method	**#chunks**	sMNIST	psMNIST	sCIFAR
BPTT	–	98.88 ± 0.19	94.31 ± 0.35	64.07 ± 0.76
TBPTT	2	98.31 ± 0.36	92.21 ± 0.63	60.11 ± 1.58
	10	98.18 ± 0.34	91.16 ± 0.24	48.83 ± 1.19
HYPR	–	96.84 ± 0.61	61.68 ± 0.52	31.47 ± 2.67
FBPTT	2	**98.34 ± 0.30**	**93.65 ± 0.51**	**62.52 ± 1.49**
(ours)	10	**98.51 ± 0.26**	**93.07 ± 0.43**	**54.12 ± 1.71**

The advantage of FBPTT over TBPTT is more pronounced when the gradient is truncated in 10 chunks as compared to 2 chunks, in particular for the harder tasks psMNIST and sCIFAR. Overall, the ANN results indicate that FBPTT combines the advantages of truncated backprop with long-range credit-assignment of approximate forward propagation. By doing so, it tightens the gap to full BPTT compared to both approximation strategies, and this benefit becomes more pronounced as the gradient truncation becomes more frequent.

FBPTT APPLIED TO RECURRENT SPIKING NEURAL NETWORK MODELS

Spiking neural networks have recently gained interest due to the possibility to implement these networks on energy-efficient neuromorphic hardware (Davies et al., 2021). Since approximate traning techniques are in particular interesting for SNNs (Bellec et al., 2020; Baronig et al., 2026; Bohnstingl et al., 2020; Xiao et al., 2022; Ortner et al., 2025), we tested FBPTT on two state-of-the-art recurrent SNN (RSNN) architectures, that are, RSNNs based on SE-ADLIF neurons (Baronig et al., 2025), and RSNNs based on BRF neurons (Higuchi et al., 2024), see Appendix A.8.3 and A.8.4 for the neuron model dynamics. We evaluated these network models again on sMNIST, psMNIST and sCIFAR as well as on the neuromorphic benchmark NeuroMorse (Walters et al., 2025) and on the long-range arena task Pathfinder-E (see Appendix A.9 for dataset descriptions).

Results are summarized in Table 3 for BRF networks and in Table 4 for SE-ADLIF networks. Across the SNN benchmarks, FBPTT consistently outperformed both HYPR and TBPTT at matched chunk length, for both the BRF and SE-adLIF networks. The gap is largest on NeuroMorse, where HYPR collapsed to near-chance accuracy, indicating that neuron-internal recurrence alone is insufficient for this task. sMNIST is the simplest task of the test suite and here, all methods stay close to the BPTT upper bound for 2 chunks, reflecting the comparatively weak dependence of this task on long-range or intra-layer credit assignment. Nevertheless, it is interesting that TBPTT shows a drop in performance below HYPR at chunks 10 for both network model. As FBPTT combines TBPTT with forward propagation, it rescues performance in this setup.

As observed for the ANN models, the advantage of FBPTT over TBPTT also grows with chunk number: Performance of TBPTT degrades substantially when moving from 2 chunks to 10 chunks, while performance degradation of FBPTT is more graceful, indicating that forward-propagated in-

Table 3: Test accuracy (%) of BRF networks on benchmark tasks (mean ± SD over 10 runs). Bold numbers indicate the best result out of {HYPR, 2-chunk TBPTT, 2-chunk FBPTT} and {HYPR, 10-chunk TBPTT, 10-chunk FBPTT}.

Method	**#chunks**	sMNIST	psMNIST	sCIFAR	NeuroMorse	Pathfinder-E
BPTT	–	98.87 ± 0.20	94.41 ± 0.34	63.94 ± 0.86	72.08 ± 0.49	85.06 ± 0.30
TBPTT	2	96.21 ± 0.39	89.82 ± 0.51	58.41 ± 1.47	64.53 ± 1.76	73.18 ± 1.09
	10	90.37 ± 0.72	65.91 ± 1.06	56.21 ± 1.98	56.79 ± 1.38	64.11 ± 0.98
HYPR	–	96.04 ± 0.57	87.08 ± 0.69	56.91 ± 1.76	2.00 ± 0.00	66.31 ± 2.31
FBPTT	2	**97.64 ± 0.32**	**90.94 ± 0.76**	**63.28 ± 1.29**	**68.79 ± 1.17**	**78.34 ± 0.89**
(ours)	10	**96.11 ± 0.48**	**89.05 ± 0.39**	**57.76 ± 1.55**	**62.14 ± 0.82**	**72.27 ± 0.69**

Table 4: Test accuracy (%) of SE-ADLIF networks on benchmark tasks (mean ± SD over 10 runs). Bold numbers indicate the best result out of {HYPR, 2-chunk TBPTT, 2-chunk FBPTT} and {HYPR, 10-chunk TBPTT, 10-chunk FBPTT}.

Method	**#chunks**	sMNIST	psMNIST	sCIFAR	NeuroMorse
BPTT	–	99.01 ± 0.19	88.34 ± 0.54	62.61 ± 0.71	51.08 ± 0.62
TBPTT	2	97.39 ± 0.40	78.28 ± 0.76	52.91 ± 1.82	44.38 ± 1.59
	10	86.31 ± 0.84	61.41 ± 1.17	51.84 ± 2.08	42.67 ± 1.47
HYPR	–	94.23 ± 0.67	62.41 ± 1.08	23.25 ± 2.91	2.00 ± 0.00
FBPTT	2	**97.61 ± 0.34**	**87.38 ± 0.41**	**55.76 ± 1.37**	**49.62 ± 1.22**
(ours)	10	**94.81 ± 0.58**	**70.01 ± 0.67**	**52.11 ± 1.72**	**48.61 ± 1.33**

ternal sensitivities continuing to assign long-range credit even as the local TBPTT window shrinks. This effect is most visible on psMNIST, where TBPTT degrades sharply from 2 chunks to 10 chunks while FBPTT's drop is comparatively minor. A similar, although less pronounced divergence of performances at longer chunk length can be observed for Pathfinder-E.

Approximate optimization on a state tracking task

Recurrent neural networks (RNNs) possess the ability to store and update a hidden state. This distinguishes RNNs from transformers (Vaswani et al., 2017) and standard state space models (Gu et al., 2022) which are provably incapable to do so (Merrill & Sabharwal, 2023; Merrill et al., 2024). Hence, it is of particular interest to evaluate which training procedures enable RNNs to learn such tasks. To this end, we considered a state tracking task where a state variable s_t always starts in the same initial state s_0. The network then sees a sequence of actions $a_1, \dots a_T$, each one changing the state in deterministic manner. The task of the network is then to output the state after the final action a_T (see Appendix A.9.1). Figure 2 shows test accuracies of both ANN and SNN models on this task with sequence length of 400. For all network models, BPTT was able to learn the task perfectly. HYPR did not manage to achieve optimal performance on any of the models, and the performance of TBPTT degraded significantly at 50 or more chunks. The performance of FBPTT degraded more gracefully and was better than both approximation methods in every tested setting. Interestingly, optimal performance was achieved with FBPTT on LSTMs even for 100 chunks, which corresponds to a chunk size of 4, i.e., every fourth time step, the gradient was truncated. It is remarkable that, although this setup is very close to the HYPR setup, the four gradient steps through the recurrent connections suffices for full performance recovery. Similarly, for the BRF network, FBPTT exhibited optimal performance for 50 chunks and a close to optimal one for 100 chunks.

Approximate optimization on a long-range memory tasks

We aimed to evaluate the performance of FBPTT on a task that demands long-range memory on a long input sequence. To this end, we considered the Add task (Hochreiter & Schmidhuber, 1997) (see Appendix A.9.2). In this classic task, the network has to add marked input values and output the sum after the sequence. We considered a version with a very long sequence length of 5000 in

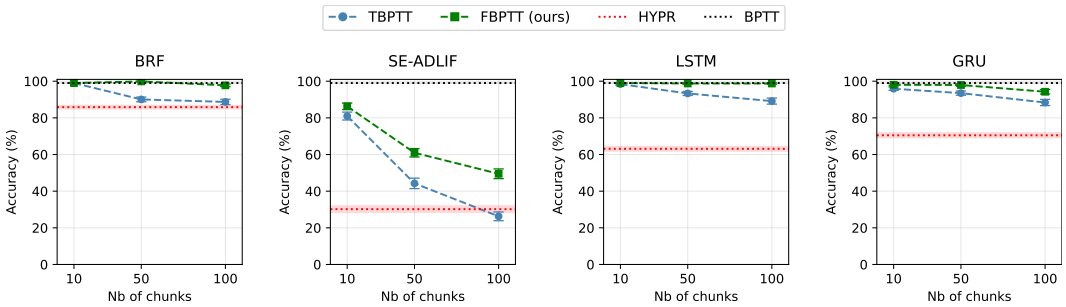


Figure 2: Test accuracy (%) across different numbers of chunks for different models on a state tracking task with sequence length 400. For HYPR and BPTT, performance is independent of chunk length.

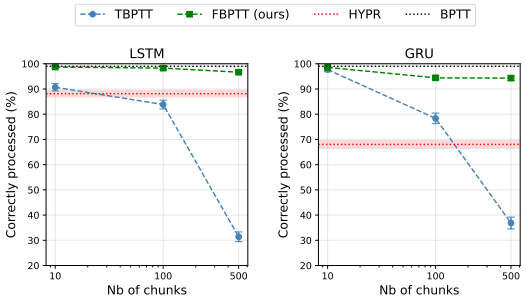


Figure 3: Performance of LSTM and GRU networks on the Add task at sequence length 5000. Shown is the percentage of correctly processed test sequences (i.e., $|\hat{y}_L - y| < 0.04$, see Sec. A.9.2) across different numbers of chunks.

order to evaluate the performance of approximate gradient methods on tasks that demand long-rang memory in the presence of frequent gradient truncation. Figure 3 shows the test results of ANN models on this task in dependence of the the number of chunks. While BPTT is able to solve the task perfectly, HYPR struggles and performance of TBPTT degrades significantly at at least 100 chunks or more. Again, FBPTT combines the advantages of TBPTT and forward propagation such that its performance is always above both approximation schemes, with close to perfect performance on the LSTM even for 500 chunks.

Gradient Alignment with BPTT

Since FBPTT combines long-range gradients through neuron-internal pathways (as computed by HYPR) with local gradients through intra-layer recurrent connections (as computed by TBPTT), we hypothesized that its approximate gradients should in general be better aligned with the true gradient than those of TBPTT and HYPR. To test this hypothesis, we re-considered the state tracking task discussed above. We trained each network model (BRF, adLIF, LSTM, GRU) on this task for 20 epochs, using each gradient evaluation method (BPTT, TBPTT, HYPR, FBPTT). For each combination, we then considered the network after epoch 1, 5, 10, and 20. Using these checkpoints, we computed the gradient of network weights with BPTT and compared it to the gradient of each other method via the cosine similarity (see Appendix A.11). This cosine similarity for LSTM networks and SNNs with BRF neurons is shown in Figure 4 (a figure with all network models is provided as Figure A1 in Appendix A.11).

Interestingly, when the network was trained with TBPTT (second column), the gradient of TBPTT was in general better aligned with BPTT than the gradient of HYPR. Reversely, when the network was trained with HYPR (third column), HYPR was in general better aligned with BPTT than TBPTT. When the network was trained with BPTT (first column), TBPTT was in general better aligned with BPTT than HYPR. Regardless, in all considered scenarios, FBPTT was better aligned

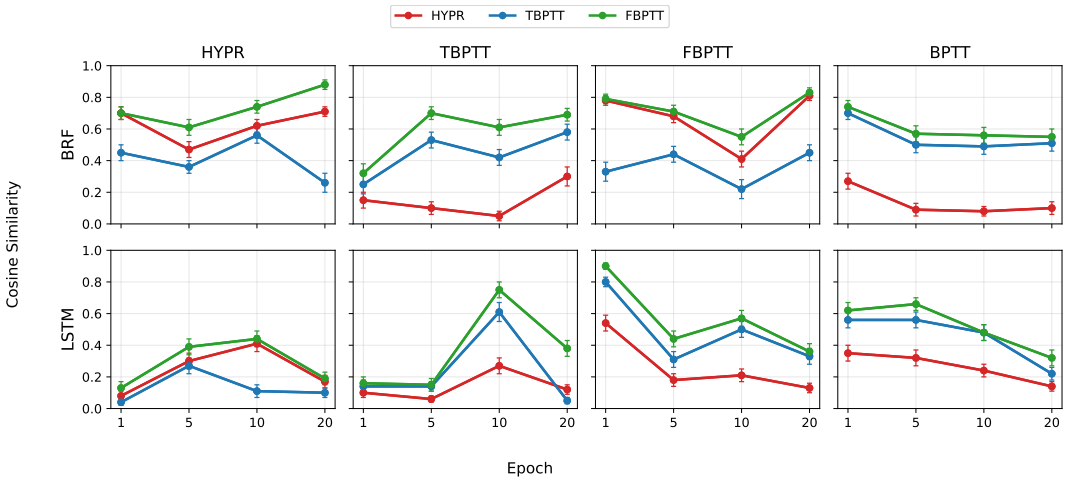


Figure 4: Gradient alignment on the state tracking task (sequence length 400) for BRF and LSTM model. Each network model (rows) was trained with each gradient evaluation method (columns). Each graph shows the cosine similarity between the weight gradients of BPTT and HYPR (red), TBPTT (blue), and FBPTT (green) at the given epoch. For TBPTT and FBPTT, chunk length is 10

with BPTT than any of the two. Typically, the alignment was similar to the better approximation or somewhat better (see also Figure A1 in the Appendix).

Conclusion

We introduced Forward-Backward Propagation Through Time (FBPTT), a hybrid training algorithm for recurrent neural networks that combines truncated backpropagation through time with a forward-propagated sensitivity matrix to recover long-range gradient information discarded at chunk boundaries. By restricting forward propagation to neuron-internal dynamics only, FBPTT maintains bounded memory and constant per-step compute, making it as scalable as TBPTT while substantially improving temporal credit assignment. Experiments across standard and artificial spiking recurrent neural network models on long-range sequence benchmarks showed that FBPTT consistently outperforms both HYPR and TBPTT, sometimes approaching the accuracy of full BPTT at a fraction of the memory cost. The gains are particularly pronounced on tasks that require tracking of long-range dependencies, such as the state tracking task and the Add task, where the performance of TBPTT degrades significantly at larger chunk numbers while FBPTT's performance remains stable.

Our proposed training algorithm has several limitations. First, it remains an approximation of the exact gradient. Because the forward-propagated sensitivity matrix only captures neuron-internal recurrences, the contributions of recurrent interactions between neurons across chunk boundaries are discarded, so some gradient information is inevitably lost. This explains the remaining gap to full BPTT on some tasks. Second, FBPTT still relies on TBPTT within each chunk, which requires storing the network states over the chunk and performing a backward pass through time. As a result, the algorithm is not fully online and is not local in time, which makes it difficult to implement on neuromorphic hardware, where learning rules are typically expected to operate online using only locally available information. Future work could address these limitations by capturing a larger part of the intra-layer recurrences at low cost, and by replacing the within-chunk TBPTT component with a fully online, hardware-friendly learning rule to enable on-chip learning on neuromorphic hardware.

AI use statement

In this work, we used generative AI tools for editing the text for readability. We have not used generative AI tools for any other task, including: generate synthetic data sets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for

proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, assist with translation, clean and reformat dataset, support qualitative and thematic data analysis, interpret results and create or edit software code. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text and software code produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

The complete formulation of the proposed method is described in Appendix A.4, and all experimental settings, including model architectures, datasets, training details and hyperparameters are provided in Appendix A.9, A.8, A.12. We used standard datasets which are publicly available. The source code of our experiments is publicly available at <github link will be inserted here after acceptance>.

ACKNOWLEDGEMENTS

REFERENCES

Maximilian Baronig, Romain Ferrand, Silvester Sabathiel, and Robert Legenstein. Advancing spatio-temporal processing through adaptation in spiking neural networks. *Nature Communications*, 16(1), 2025.

Maximilian Baronig, Yeganeh Bahariasl, Ozan Özdenizci, and Robert Legenstein. A scalable hybrid training approach for recurrent spiking neural networks. *Neuromorphic Computing and Engineering*, 6(1):014017, mar 2026. doi: 10.1088/2634-4386/ae46d4.

Guillaume Bellec, Darjan Salaj, Anand Subramoney, Robert Legenstein, and Wolfgang Maass. Long short-term memory and learning-to-learn in networks of spiking neurons. *Advances in Neural Information Processing Systems*, 31, 2018.

Guillaume Bellec, Franz Scherr, Anand Subramoney, Elias Hajek, Darjan Salaj, Robert Legenstein, and Wolfgang Maass. A solution to the learning dilemma for recurrent networks of spiking neurons. *Nature Communications*, 11(1):3625, 2020. doi: 10.1038/s41467-020-17236-y.

Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, 1990.

Thomas Bohnstingl, Stanisław Woźniak, Wolfgang Maass, Angeliki Pantazi, and Evangelos Eleftheriou. Online spatio-temporal learning in deep neural networks, 2020.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Mike Davies, Andreas Wild, Garrick Orchard, Yulia Sandamirskaya, Gabriel A Fonseca Guerra, Prasad Joshi, Philipp Plank, and Sumedh R Risbud. Advancing neuromorphic computing with loihi: A survey of results and outlook. *Proceedings of the IEEE*, 109(5):911–934, 2021.

Felix Effenberger, Pedro Carvalho, Igor Dubinin, and Wolf Singer. The functional role of oscillatory dynamics in neocortical circuits: A computational perspective. *Proceedings of the National Academy of Sciences*, 122(4):e2412830122, 2025. doi: 10.1073/pnas.2412830122.

Steven K. Esser, Paul A. Merolla, John V. Arthur, Andrew S. Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J. Berg, Jeffrey L. McKinstry, Timothy Melano, Davis R. Barch, Carmelo di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D. Flickner, and Dharmendra S. Modha. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences*, 113(41):11441–11446, 2016. doi: 10.1073/pnas.1604850113.

Arthur Gill. On the bound to the memory of a sequential machine. *IEEE Transactions on Electronic Computers*, 14(3):464–466, 1965. doi: 10.1109/PGEC.1965.264154.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*, 2022.

Saya Higuchi, Sebastian Kairat, Sander M. Bohte, and Sebastian Otte. Balanced resonate-and-fire neurons. *arXiv preprint arXiv:2402.14603*, 2024.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

Max Jaderberg, Wojciech Marian Czarnecki, Simon Osindero, Oriol Vinyals, Alex Graves, David Silver, and Koray Kavukcuoglu. Decoupled neural interfaces using synthetic gradients. *arXiv preprint arXiv:1608.05343*, 2017.

Jacques Kaiser, Hesham Mostafa, and Emre Neftci. Synaptic plasticity dynamics for deep continuous local learning (decolle). *Frontiers in Neuroscience*, Volume 14 - 2020, 2020. ISSN 1662-453X. doi: 10.3389/fnins.2020.00424.

Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Yann LeCun and Corinna Cortes. The mnist database of handwritten digits. 2005.

Owen Marschall, Kyunghyun Cho, and Cristina Savin. A unified framework of online learning algorithms for training recurrent neural networks. *Journal of Machine Learning Research*, 21 (135):1–34, 2020.

William Merrill and Ashish Sabharwal. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023.

William Merrill, Jackson Petty, and Ashish Sabharwal. The illusion of state in state-space models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pp. 35492–35506, 2024.

Asier Mujika, Florian Meier, and Angelika Steger. Approximating real-time recurrent learning with random kronecker factors. *Advances in Neural Information Processing Systems*, 31:6594–6603, 2018.

James M. Murray. Local online learning in recurrent networks with random feedback. *eLife*, 8 (e43299), 2019.

Thomas Ortner, Horst Petschenig, Athanasios Vasilopoulos, Roland Renner, Špela Brglez, Thomas Limbacher, Enrique Piñero, Alejandro Linares-Barranco, Angeliki Pantazi, and Robert Legenstein. Rapid learning with phase-change memory-based in-memory computing through learning-to-learn. *Nature Communications*, 16(1):1243, 2025.

Sumit B Shrestha and Garrick Orchard. Slayer: Spike layer error reassignment in time. *Advances in Neural Information Processing Systems*, 31, 2018.

Corentin Tallec and Yann Ollivier. Unbiased online recurrent optimization. *arXiv preprint arXiv:1702.05043*, 2017.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena: A benchmark for efficient transformers. *arXiv preprint arXiv:2011.04006*, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Ben Walters, Yeshwanth Bethi, Taylor Kergan, Binh Nguyen, Amirali Amirsoleimani, Jason K. Eshraghian, Saeed Afshar, and Mostafa Rahimi Azghadi. Neuromorse: A temporally structured dataset for neuromorphic computing. *Neuromorphic Computing and Engineering*, 5(2):027001, 2025. doi: 10.1088/2634-4386/add36c.

Jakob Stefan Weber. Combining truncated bptt and truncated rtrl for lstm training. Master's thesis, Johannes Kepler University Linz, 2024.

Paul J. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990. doi: 10.1109/5.58337.

Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for online training of recurrent network trajectories. *Neural Computation*, 2(4):490–501, 1990. doi: 10.1162/neco.1990.2.4.490.

Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989. doi: 10.1162/neco.1989.1.2.270.

Mingqing Xiao, Qingyan Meng, Zongpeng Zhang, Di He, and Zhouchen Lin. Online training through time for spiking neural networks. In *Advances in Neural Information Processing Systems*, volume 35, pp. 20717–20730, 2022.

Bojian Yin, Federico Corradi, and Sander M Bohté. Accurate and efficient time-domain classification with adaptive spiking recurrent neural networks. *Nature Machine Intelligence*, 3(10): 905–913, 2021.

Friedemann Zenke and Emre O. Neftci. Brain-inspired learning on neuromorphic substrates. *arXiv preprint arXiv:2010.11931*, 2020.

A Appendix

This appendix develops the gradient-computation framework used throughout the paper and provides the experimental details needed to reproduce our results. We first introduce the general recurrent neuron model and network setup, together with the notational conventions (total vs. partial derivatives) used throughout. We then review the two standard *exact* gradient computation methods, backpropagation through time (BPTT) and real-time recurrent learning (RTRL), and show that they compute the same gradient via different recursions. Next, we describe the two classical *approximations* used in practice: truncated BPTT (TBPTT) and a block-diagonal (neuron-internal) approximation to the RTRL sensitivity matrix. Building on these two approximations, we derive our proposed algorithm, Forward-Backward Propagation Through Time (FBPTT), which combines local TBPTT gradients with a long-range correction term, first for a single truncation boundary and then for the general multi-chunk case. We then present the technical details needed for an efficient implementation: a low-memory representation of the immediate parameter derivative F^t (Section A.5.1), the resulting block-diagonal structure of the approximate sensitivity matrix $\tilde{G}^t$ (Section A.5.2), and the parallel computation of $\tilde{G}^\lambda$. This is followed by pseudocode summarizing the full algorithm (Section A.6) and an analysis of its computational complexity (Section A.7).

The remaining sections describe the experimental setup. We define the neuron models used in our experiments (LSTM, GRU, BRF, SE-adLIF, and HORN) and the datasets (state tracking, add task, NeuroMorse, Pathfinder-E, sequential MNIST, and sequential CIFAR), and we report extended results. We then present a detailed gradient analysis, including the experimental setup, initialization, gradient representation, and comparison metrics, with results for all network models. Finally, we provide training details and the hyperparameters and tuning procedures used for all experiments.

A.1 Neuron Model and Network Setup

We consider a single layer of m recurrently connected stateful neurons, where each neuron i has an arbitrary, possibly nonlinear, state transition function $f_{\boldsymbol{\theta}_i}$ and an output function g,

$$\mathbf{h}_i^t = f_{\boldsymbol{\theta}_i}(\mathbf{h}_i^{t-1},\ \mathbf{x}^t,\ \mathbf{y}^{t-1}), \qquad y_i^t = g(\mathbf{h}_i^t), \tag{A1}$$

where $\mathbf{h}_i^t \in \mathbb{R}^k$ is the k-dimensional state of neuron i at time step t, $\mathbf{x}^t \in \mathbb{R}^d$ is the input vector at time t, $\boldsymbol{\theta}_i \in \mathbb{R}^p$ is the parameter vector of neuron i, and $\mathbf{y}^t = (y_1^t, \dots, y_m^t)^\top \in \mathbb{R}^m$ is the vector of neuron outputs. The state dimension k varies across neuron models. We denote by $\mathbf{h}^t \in \mathbb{R}^{mk}$ the vector obtained by concatenating the k-dimensional state vectors of all m neurons in the layer. Similarly, we concatenate all neuron parameters $\boldsymbol{\theta}_i$ into a vector $\boldsymbol{\theta}$, which thus subsumes all parameters of the layer. Using this notation, we can write

$$\mathbf{h}^t = f_{\boldsymbol{\theta}}(\mathbf{h}^{t-1},\ \mathbf{x}^t,\ \mathbf{y}^{t-1}). \tag{A2}$$

In spiking neuron models, the output function g is the Heaviside step function applied to the membrane potential, which is not differentiable. Following common practice, we approximate its derivative using a surrogate gradient (Esser et al., 2016; Bellec et al., 2018). In Long Short-Term Memory (LSTM; (Hochreiter & Schmidhuber, 1997)) neurons, the state vector $\mathbf{h}_i^t = (c_i^t, z_i^t)^\top$ consists of a cell state and a hidden state, updated through input, forget, and output gates. In Gated Recurrent Unit (GRU; (Chung et al., 2014)) neurons, the state reduces to a single hidden state controlled by a reset and an update gate, achieving a similar gating mechanism to the LSTM with fewer parameters. A detailed description of all three neuron models is given in Appendix A.8.

Given a temporal sequence task, the total loss over a sequence of length T is

$$L = \sum_{t=1}^{T} L^t, \tag{A3}$$

where L^t is the per-time-step loss. In the following sections, we derive gradient computation methods for minimizing L with respect to $\boldsymbol{\theta}$.

Notational convention: total vs. partial derivatives. We distinguish in our notation between total and partial derivatives. The notation $\frac{d\mathbf{h}^t}{d\boldsymbol{\theta}}$, for example, denotes the total derivative of the hidden

state $\mathbf{h}^t$ at time t with respect to the parameters $\boldsymbol{\theta}$ (more precisely, the matrix of derivatives of the components of $\mathbf{h}^t$ with respect to the components of $\boldsymbol{\theta}$). Since $\mathbf{h}^t$ depends on previous hidden states $\mathbf{h}^{t'}$, which in turn depend on $\boldsymbol{\theta}$, all these dependencies are taken into account. In contrast, the partial derivative $\frac{\partial \mathbf{h}^t}{\partial \boldsymbol{\theta}}$ denotes the derivative of $\mathbf{h}^t$ with respect to $\boldsymbol{\theta}$ in its explicit appearance at time step t according to Eq. (A2), treating the arguments $\mathbf{h}^{t-1}$ and $\mathbf{y}^{t-1}$ of the function $f_{\boldsymbol{\theta}}$ as constants. This corresponds to the local parameter influence in the unrolled computational graph. To emphasize this local influence and to improve readability, we write $\frac{\partial \mathbf{h}^t}{\partial \boldsymbol{\theta}^t}$ for this partial derivative in what follows.

A.2 Exact Gradient Computation

We first review the two standard ways of computing the *exact* gradient $\nabla_\theta L$: the backward-mode recursion (BPTT) and the forward-mode recursion (RTRL). Both compute the same gradient; the two subsections below present the corresponding recursions and definitions, and are followed by a short remark on their equivalence. This is the material summarized in the "Background: Exact and Approximate Gradient Computation" part of the main text.

A.2.1 Backward-Mode Gradient (Backpropagation Through Time)

The exact parameter-gradient of the loss can be computed as

$$\nabla_\theta L = \frac{dL}{d\theta} = \sum_{t=1}^{T} \frac{dL}{d\mathbf{h}^t} \frac{\partial \mathbf{h}^t}{\partial \theta^t}. \tag{A4}$$

We define the error $\boldsymbol{\delta}_t \in \mathbb{R}^{mk}$ and the immediate parameter influence $F^t \in \mathbb{R}^{mp \times mk}$ as

$$\boldsymbol{\delta}_t = \frac{dL}{d\mathbf{h}^t}, \qquad F^t = \frac{\partial \mathbf{h}^t}{\partial \theta^t}. \tag{A5}$$

Using these definitions, Eq. (A4) becomes

$$\nabla_\theta L = \sum_{t=1}^{T} \boldsymbol{\delta}_t F^t. \tag{A6}$$

This equation is essential for BPTT since F^t can be computed instantaneously at time t, and $\boldsymbol{\delta}_t$ can be backpropagated as

$$\boldsymbol{\delta}_t = \boldsymbol{\ell}^t + H^{t+1} \boldsymbol{\delta}_{t+1}, \qquad \boldsymbol{\delta}_{T+1} = 0, \tag{A7}$$

where $\boldsymbol{\ell}^t$ is the immediate loss derivative and H^t is the Jacobian of the network dynamics at time t,

$$\boldsymbol{\ell}^t = \frac{\partial L^t}{\partial \mathbf{h}^t}, \qquad H^t = \frac{d\mathbf{h}^t}{d\mathbf{h}^{t-1}}. \tag{A8}$$

A.2.2 Forward-Mode Gradient (Real-Time Recurrent Learning)

The parameter gradient $\nabla_\theta L$ can also be computed in a forward manner using the RTRL algorithm, since it can equivalently be written as

$$\nabla_\theta L = \sum_{t=1}^{T} \frac{\partial L^t}{\partial \mathbf{h}^t} \frac{d\mathbf{h}^t}{d\theta}, \tag{A9}$$

compare to Eq. (A4). By defining the sensitivity matrix $G^t \in \mathbb{R}^{mk \times mp}$ as

$$G^t = \frac{d\mathbf{h}^t}{d\theta}, \tag{A10}$$

the RTRL Eq. (A9) becomes

$$\nabla_\theta L = \sum_{t=1}^{T} \boldsymbol{\ell}^t G^t, \tag{A11}$$

compare to Eq. (A6). This equation is essential for RTRL, as the sensitivity matrix can be forward propagated,

$$G^t = H^t G^{t-1} + F^t, \qquad G^0 = 0. \tag{A12}$$

As a side remark, we note that from Eq. (A6) and Eq. (A11), it follows that

$$\sum_{t=1}^{T} \boldsymbol{\delta}_t F^t = \sum_{t=1}^{T} \boldsymbol{\ell}^t G^t. \tag{A13}$$

BPTT and RTRL compute the exact same gradient, differing only in whether the sum over t is accumulated backward (via $\boldsymbol{\delta}_t$) or forward (via G^t).

A.3 Approximate Gradient Computation

Exact BPTT requires storing the full computational graph and is not causal; exact RTRL requires forward-propagating the full sensitivity matrix G^t, which is prohibitively expensive for large layers. We review two standard approximations before combining them into our proposed algorithm in Section A.4.

A.3.1 Truncated Backpropagation Through Time (Approximate Backward Pass)

In truncated BPTT, the sequence of length T is divided into k non-overlapping chunks of length λ (we assume for simplicity that T is a multiple of λ, so $T = k\lambda$). The error is then backpropagated only within each chunk. This is the approximation referred to as "Truncated BPTT" in the main text.

To formalize this, we first introduce some notation. Note that

$$\boldsymbol{\delta}_t = \frac{dL}{d\mathbf{h}^t} = \sum_{t'=1}^{T} \frac{dL^{t'}}{d\mathbf{h}^t}. \tag{A14}$$

We generalize this to the sum of losses over a temporal interval from r to s,

$$\boldsymbol{\delta}_t^{r:s} = \sum_{t'=r}^{s} \frac{dL^{t'}}{d\mathbf{h}^t}. \tag{A15}$$

Note that since $\frac{dL^{t'}}{d\mathbf{h}^t} = 0$ for $t' < t$, we have $\boldsymbol{\delta}_t^{t:s} = \boldsymbol{\delta}_t^{1:s}$. Furthermore, for a sequence of length T, we have $\boldsymbol{\delta}_t = \boldsymbol{\delta}_t^{1:T} = \boldsymbol{\delta}_t^{1:\lambda} + \boldsymbol{\delta}_t^{\lambda+1:T}$ for any $\lambda \in \{1, \dots, T-1\}$.

The gradient is approximated by truncating the backward recursion after each chunk,

$$\nabla_\theta L \approx \sum_{i=0}^{k-1} \sum_{t=i\lambda+1}^{(i+1)\lambda} \boldsymbol{\delta}_t^{i\lambda+1:(i+1)\lambda} F^t. \tag{A16}$$

Gradients are not propagated across chunk boundaries. Hence, TBPTT only captures dependencies within the current chunk and ignores long-range dependencies.

The following property of the partial loss sum will be convenient when we derive the FBPTT algorithm in Section A.4. For $t < s$, we have

$$\boldsymbol{\delta}_t^{r:s} = \sum_{t'=r}^{s} \frac{dL^{t'}}{d\mathbf{h}^r} \frac{d\mathbf{h}^r}{d\mathbf{h}^t} = \boldsymbol{\delta}_r^{r:s} \frac{d\mathbf{h}^r}{d\mathbf{h}^t}. \tag{A17}$$

A.3.2 Approximate Forward Propagation (Neuron-Internal Sensitivity)

Stateful RNNs often involve two sources of recurrence: the neuron-internal (implicit) recurrence through the neuron states $\mathbf{h}^{t-1}$, and the intra-layer (explicit) recurrence that arises from the layer outputs at the previous time step $\mathbf{y}^{t-1}$ through the recurrent connections. This is the decomposition summarized in the "Decomposing the recurrence" part of the main text. One can make this explicit by rewriting the state transition function (A1) as

$$\mathbf{h}_i^t = f_{\theta_i}(\mathbf{h}_i^{t-1},\ \mathbf{x}^t,\ \mathbf{y}^{t-1}) = f^{\text{combine}}(f_{\theta_i^{\text{I}}}^{\text{I}}(\mathbf{h}_i^{t-1}),\ f_{\theta_i^{\text{in}}}^{\text{in}}(\mathbf{x}^t),\ f_{\theta_i^{\text{E}}}^{\text{E}}(\mathbf{y}^{t-1})), \tag{A18}$$

where $f^{\mathrm{I}}_{\theta^{\mathrm{I}}_i}$, $f^{\mathrm{in}}_{\theta^{\mathrm{in}}_i}$, and $f^{\mathrm{E}}_{\theta^{\mathrm{E}}_i}$ denote the implicit, input-driven, and explicit state transition functions with parameters θ^{I}_i (neuron dynamics parameters), θ^{in}_i (input weights), and θ^{E}_i (recurrent weights), respectively. For standard recurrent LIF networks, the combine function f^{combine} is simply the sum.

Furthermore, note that the layer outputs $\mathbf{y}^{t-1}$ depend on the hidden states $\mathbf{h}^{t-1}$, while $\mathbf{x}^t$ does not. Hence, the exact state-to-state derivative $H^t \in \mathbb{R}^{mk\times mk}$ for a layer of m neurons is given by the chain rule,

$$H^t = \frac{d\mathbf{h}^t}{d\mathbf{h}^{t-1}} = \underbrace{\frac{\partial \mathbf{h}^t}{\partial \mathbf{h}^{t-1}}}_{H^{I,t}} + \underbrace{\frac{\partial \mathbf{h}^t}{\partial \mathbf{y}^{t-1}}\frac{\partial \mathbf{y}^{t-1}}{\partial \mathbf{h}^{t-1}}}_{H^{E,t}}. \tag{A19}$$

The partial derivatives $\partial\mathbf{h}^t/\partial\mathbf{h}^{t-1}$ and $\partial\mathbf{h}^t/\partial\mathbf{y}^{t-1}$ are obtained by differentiating the function f in Eq. (A18) with respect to $\mathbf{h}^{t-1}$ and $\mathbf{y}^{t-1}$, respectively. The partial derivative $\partial\mathbf{y}^{t-1}/\partial\mathbf{h}^{t-1}$ is obtained by computing the derivative of the function g from Eq. (A1), or by using a surrogate derivative if g is not differentiable.

Hence, one can decompose the state-to-state Jacobian H^t into a block-diagonal neuron-internal component $H^{I,t}$ and a dense intra-layer component $H^{E,t}$, such that $H^t = H^{I,t} + H^{E,t}$. Using this decomposition, Eq. (A12) becomes

$$G^t = \underbrace{H^{I,t}G^{t-1}}_{\text{neuron-internal}} + \underbrace{H^{E,t}G^{t-1}}_{\text{intra-layer}} + F^t. \tag{A20}$$

In various algorithms (Murray, 2019; Bellec et al., 2020; Baronig et al., 2026), only the neuron-internal block-diagonal part of G^t is forward-propagated,

$$\tilde{G}^t = H^{I,t}\tilde{G}^{t-1} + F^t, \quad \tilde{G}^0 = 0. \tag{A21}$$

An approximation of $\nabla_\theta L$ is then computed as

$$\nabla_\theta L \approx \sum_{t=1}^{T} \boldsymbol{\ell}^t \tilde{G}^t. \tag{A22}$$

This forward-propagated, neuron-internal sensitivity matrix $\tilde{G}^t$ is the key building block reused in Section A.4 to construct the FBPTT correction term.

A.4 Forward-Backward Propagation Through Time (FBPTT)

We now combine the two approximations from Section A.3 truncated BPTT (Section A.3.1) and the neuron-internal forward sensitivity $\tilde{G}^t$ (Section A.3.2) into our proposed algorithm. This is the full derivation of the gradient estimator stated in Eq. 6 of the main text. We first derive the correction term at a single truncation boundary (Section A.4.1), generalize it to the full multi-chunk sequence (Section A.4.2).

A.4.1 Gradient Decomposition at a Single Truncation Boundary

To motivate our algorithm, we start with TBPTT, make its approximation explicit, and then show how FBPTT improves upon it.

We first introduce some notation. Similar to the partial error sum $\boldsymbol{\delta}_t^{r:s}$ defined in Eq. (A15), we define the partial sum of loss gradients with respect to the parameters at time t as

$$\frac{dL^{r:s}}{d\theta^t} = \sum_{t'=r}^{s} \frac{dL^{t'}}{d\mathbf{h}^t}\frac{\partial \mathbf{h}^t}{\partial \theta^t} = \boldsymbol{\delta}_t^{r:s}\frac{\partial \mathbf{h}^t}{\partial \theta^t} = \boldsymbol{\delta}_t^{r:s}F^t. \tag{A23}$$

Note that since $\frac{dL^{t'}}{d\mathbf{h}^t} = 0$ for $t' < t$, we have $\frac{dL^{t:s}}{d\theta^t} = \frac{dL^{1:s}}{d\theta^t}$. Furthermore, for a sequence of length T, we have $\frac{dL}{d\theta^t} = \frac{dL^{1:T}}{d\theta^t} = \frac{dL^{1:\lambda}}{d\theta^t} + \frac{dL^{\lambda+1:T}}{d\theta^t}$ for any $\lambda \in \{1,\dots,T-1\}$.

To understand the approximation made by TBPTT, we consider a single truncation boundary after step λ, splitting the sequence into two chunks: chunk 0 ($t \in \{1,\dots,\lambda\}$) and chunk 1 ($t \in \{\lambda +$

$1, \ldots, T\}$). For $t > \lambda$, TBPTT computes the correct gradient $\frac{dL}{d\theta^t}$. For any time step $t \leq \lambda$, the exact gradient decomposes as

$$\frac{dL}{d\theta^t} = \underbrace{\frac{dL^{1:\lambda}}{d\theta^t}}_{\text{within chunk 0}} + \underbrace{\frac{dL^{\lambda+1:T}}{d\theta^t}}_{\text{cross-chunk}}, \tag{A24}$$

where the second term is ignored by TBPTT. Combining Eq. (A23) with Eq. (A17), the cross-chunk term can be written as

$$\frac{dL^{\lambda+1:T}}{d\theta^t} = \boldsymbol{\delta}_t^{\lambda+1:T} F^t = \boldsymbol{\delta}_{\lambda+1}^{\lambda+1:T} \frac{d\mathbf{h}^{\lambda+1}}{d\mathbf{h}^t} F^t. \tag{A25}$$

This shows that the terms ignored by TBPTT could in principle be recovered by backpropagating the errors in chunk 1 to obtain $\boldsymbol{\delta}_{\lambda+1}^{\lambda+1:T}$, and then multiplying the result by $\frac{d\mathbf{h}^{\lambda+1}}{d\mathbf{h}^t} F^t$, which can be obtained by forward propagation through the Jacobians. However, forward-propagating the full Jacobian is computationally too expensive. We therefore propose to forward-propagate only through the Jacobians of the neuron-internal dynamics (Baronig et al., 2025; Bellec et al., 2020), which can be computed efficiently,

$$\frac{d\mathbf{h}^{\lambda+1}}{d\mathbf{h}^t} \approx \frac{\hat{d}\mathbf{h}^{\lambda+1}}{\hat{d}\mathbf{h}^t} \stackrel{\text{def}}{=} \prod_{s=t+1}^{\lambda+1} H^{I,s}. \tag{A26}$$

With this approximation, the cross-chunk term becomes

$$\frac{dL^{\lambda+1:T}}{d\theta^t} \approx \boldsymbol{\delta}_{\lambda+1}^{\lambda+1:T} \frac{\hat{d}\mathbf{h}^{\lambda+1}}{\hat{d}\mathbf{h}^t} F^t. \tag{A27}$$

Summing over $t = 1, \ldots, \lambda$ in chunk 0 and factoring out $\boldsymbol{\delta}_{\lambda+1}^{\lambda+1:T}$, we obtain

$$\sum_{t=1}^{\lambda} \frac{dL^{\lambda+1:T}}{d\theta^t} \approx \boldsymbol{\delta}_{\lambda+1}^{\lambda+1:T} \underbrace{\sum_{t=1}^{\lambda} \frac{\hat{d}\mathbf{h}^{\lambda+1}}{\hat{d}\mathbf{h}^t} F^t}_{= H^{I,\lambda+1}\tilde{G}^\lambda} = \boldsymbol{\delta}_{\lambda+1}^{\lambda+1:T} H^{I,\lambda+1} \tilde{G}^\lambda, \tag{A28}$$

where the identity $\sum_{t=1}^{\lambda} \frac{\hat{d}\mathbf{h}^{\lambda+1}}{\hat{d}\mathbf{h}^t} F^t = H^{I,\lambda+1}\tilde{G}^\lambda$ follows from unrolling the recursion $\tilde{G}^t = H^{I,t}\tilde{G}^{t-1} + F^t$ from Eq. (A21) up to step λ, and then applying one additional forward step via $H^{I,\lambda+1}$:

$$\begin{aligned}
\tilde{G}^1 &= F^1, \\
\tilde{G}^2 &= H^{I,2} F^1 + F^2, \\
&\vdots \\
\tilde{G}^\lambda &= \sum_{t=1}^{\lambda} \left(\prod_{s=t+1}^{\lambda} H^{I,s} \right) F^t, \\
H^{I,\lambda+1}\tilde{G}^\lambda &= \sum_{t=1}^{\lambda} \left(\prod_{s=t+1}^{\lambda+1} H^{I,s} \right) F^t.
\end{aligned} \tag{A29}$$

Here, each term $\left(\prod_{s=t+1}^{\lambda+1} H^{I,s}\right) F^t$ is the contribution of the parameters at step t, propagated forward through the internal dynamics to $\mathbf{h}^{\lambda+1}$. The total approximate gradient for chunk 0 is

$$\sum_{t=1}^{\lambda} \frac{dL}{d\theta^t} \approx \underbrace{\sum_{t=1}^{\lambda} \boldsymbol{\delta}_t^{1:\lambda} F^t}_{\text{truncated BPTT}} + \underbrace{\boldsymbol{\delta}_{\lambda+1}^{\lambda+1:T} H^{I,\lambda+1} \tilde{G}^\lambda}_{\text{long-term correction}}. \tag{A30}$$

A.4.2 General Multi-Chunk Case

Applying the same decomposition to each chunk $i \in \{0, \dots, k-1\}$ and summing over all k chunks gives the proposed gradient approximation

$$\nabla_\theta L \approx \underbrace{\sum_{i=0}^{k-1} \sum_{t=i\lambda+1}^{(i+1)\lambda} \boldsymbol{\delta}_t^{i\lambda+1:(i+1)\lambda} F^t}_{\text{truncated BPTT (local)}} + \underbrace{\sum_{i=1}^{k-1} \boldsymbol{\delta}_{i\lambda+1}^{i\lambda+1:(i+1)\lambda} H^{I,i\lambda+1} \tilde{G}^{i\lambda}}_{\text{long-term correction}}, \tag{A31}$$

where the second sum starts at $i = 1$ since $\tilde{G}^0 = 0$. Here, $\boldsymbol{\delta}_{i\lambda+1}^{i\lambda+1:(i+1)\lambda}$ is the boundary error signal obtained by backpropagating the loss from chunk $i+1$, and $H^{I,i\lambda+1}$ is the neuron-internal recurrence matrix. This is the gradient estimator given in the main text Equation (6).

The correction term at each chunk boundary involves three approximations:

1. $\boldsymbol{\delta}_{i\lambda+1}^{i\lambda+1:(i+1)\lambda}$ approximates the exact loss signal using truncated BPTT within chunk $i+1$.
2. $H^{I,i\lambda+1}$ approximates the full Jacobian $H^{i\lambda+1}$ at the boundary step using internal dynamics only.
3. $\tilde{G}^{i\lambda}$ approximates the exact sensitivity matrix $G^{i\lambda} = \frac{d\mathbf{h}^{i\lambda}}{d\theta}$ by propagating only through internal dynamics within chunk i.

The two terms in Eq. (A31) are complementary: truncated BPTT uses the full Jacobian H^t within each chunk, capturing short-range dependencies including intra-layer recurrence, while the correction term uses $\tilde{G}^{i\lambda}$, accumulated continuously across all previous chunks via neuron-internal dynamics $H^{I,t}$, to recover the long-range gradient information that truncation discards.

A.5 Efficient Implementation Details

This section provides the technical building blocks needed to implement FBPTT efficiently: a compact representation of the per-neuron parameter derivative F^t (Section A.5.1) and the resulting block-diagonal structure of $\tilde{G}^t$ (Section A.5.2). For clarity and consistency, we follow the notation and derivations of Baronig et al. (2026), which provides a more detailed explanation of these technical aspects.

A.5.1 Efficient Representation of the Immediate Parameter Derivative F^t

For a layer of m neurons with input dimension d, state dimension k, and $p = d + m + 1$ parameters per neuron (feed-forward weights $\mathbf{w}_{\text{ff},i} \in \mathbb{R}^d$, recurrent weights $\mathbf{w}_{\text{rec},i} \in \mathbb{R}^m$, and bias $b_i \in \mathbb{R}$), the full matrix $F^t \in \mathbb{R}^{mk \times mp}$ would naively require $\mathcal{O}(m^2kp)$ memory. We show below that this can be reduced to $\mathcal{O}(d+m+mk)$ elements by exploiting block-diagonality and a low-rank factorization.

Since the parameters of neuron j have no direct influence on the state of neuron $i \neq j$, the matrix F^t is block-diagonal,

$$F^t = \begin{pmatrix} F_1^t & 0 & \cdots & 0 \\ 0 & F_2^t & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & F_m^t \end{pmatrix}, \tag{A32}$$

where $F_i^t = \frac{d\mathbf{h}_i^t}{d\theta_i^t} \in \mathbb{R}^{k \times p}$ is the local parameter derivative for neuron i.

Decomposing the state transition into a scalar input $I_i^t = \mathbf{w}_{\text{ff},i}^\intercal \mathbf{x}^t + \mathbf{w}_{\text{rec},i}^\intercal \mathbf{y}^{t-1} + b_i$ followed by an internal update $\mathbf{h}_i^t = \hat{f}(\mathbf{h}_i^{t-1}, I_i^t)$, the chain rule gives

$$F_i^t = \frac{d\mathbf{h}_i^t}{d\theta_i^t} = \frac{\partial \mathbf{h}_i^t}{\partial I_i^t} \frac{\partial I_i^t}{\partial \theta_i^t}, \tag{A33}$$

where $\frac{\partial \mathbf{h}_i^t}{\partial I_i^t} \in \mathbb{R}^{k\times 1}$ and $\frac{\partial I_i^t}{\partial \theta_i^t} \in \mathbb{R}^{1\times p}$ are both low-dimensional. Furthermore, the second factor is identical across all neurons,

$$\frac{\partial I_i^t}{\partial \mathbf{w}_{\text{ff},i}} = \mathbf{x}^t, \qquad \frac{\partial I_i^t}{\partial \mathbf{w}_{\text{rec},i}} = \mathbf{y}^{t-1}, \qquad \frac{\partial I_i^t}{\partial b_i} = 1, \tag{A34}$$

since these quantities do not depend on the neuron index i. Storing only these low-dimensional factors reduces the effective memory footprint of F^t to $\mathcal{O}(d + m + mk)$ elements per time step.

A.5.2 Block-Diagonality of the Approximate Sensitivity Matrix $\tilde{G}^t$

The block-diagonality of $\tilde{G}^t$ follows directly from the block-diagonality of both $H^{I,t}$ and F^t established above. The internal Jacobian $H^{I,t}$ is block-diagonal,

$$H^{I,t} = \begin{pmatrix} H_{I,1}^t & 0 & \cdots & 0 \\ 0 & H_{I,2}^t & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & H_{I,m}^t \end{pmatrix}, \tag{A35}$$

where $H_{I,i}^t = \frac{\partial \mathbf{h}_i^t}{\partial \mathbf{h}_i^{t-1}} \in \mathbb{R}^{k\times k}$ is the neuron-internal state-to-state Jacobian of neuron i, obtained by treating the outputs $\mathbf{y}^{t-1}$ of other neurons as constant. All off-diagonal blocks are zero, since $\frac{\partial \mathbf{h}_i^t}{\partial \mathbf{h}_j^{t-1}} = \mathbf{0}$ for $i \neq j$. This is the block-diagonal structure of $H^{I,t}$ referenced in the "Decomposing the recurrence" part of the main text.

Since both $H^{I,t}$ and F^t are block-diagonal, the recurrence $\tilde{G}^t = H^{I,t}\tilde{G}^{t-1} + F^t$ preserves block-diagonality, giving

$$\tilde{G}^t = \begin{pmatrix} \tilde{G}_1^t & 0 & \cdots & 0 \\ 0 & \tilde{G}_2^t & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \tilde{G}_m^t \end{pmatrix}, \tag{A36}$$

where each block satisfies $\tilde{G}_i^t = H_{I,i}^t \tilde{G}_i^{t-1} + F_i^t$, approximating the local sensitivity $\frac{d\mathbf{h}_i^t}{d\theta_i}$ of neuron i's state with respect to its own parameters.

A.5.3 Parallel Computation of $\tilde{G}^\lambda$

Unrolling the recurrence $\tilde{G}^t = H^{I,t}\tilde{G}^{t-1} + F^t$ from Eq. (A21) explicitly gives

$$\tilde{G}^\lambda = F^\lambda + \underbrace{H^{I,\lambda}}_{\Phi_{(\lambda:\lambda)}} F^{\lambda-1} + \underbrace{H^{I,\lambda}H^{I,\lambda-1}}_{\Phi_{(\lambda:\lambda-1)}} F^{\lambda-2} + \ldots + \underbrace{H^{I,\lambda}\cdots H^{I,2}}_{\Phi_{(\lambda:2)}} F^1 + \underbrace{H^{I,\lambda}\cdots H^{I,1}}_{\Phi_{(\lambda:1)}} \tilde{G}^0, \tag{A37}$$

which can be written compactly by introducing the cumulative internal transition matrix from step t to step λ,

$$\Phi_{(\lambda:t)} = \prod_{s=t}^{\lambda} H^{I,s}, \tag{A38}$$

as

$$\tilde{G}^\lambda = F^\lambda + \sum_{t=1}^{\lambda-1} \Phi_{(\lambda:t+1)}\, F^t + \Phi_{(\lambda:1)}\, \tilde{G}^0. \tag{A39}$$

The immediate parameter influence matrices $F^t = \frac{\partial \mathbf{h}^t}{\partial \theta^t}$ and the internal Jacobians $H^{I,t}$ can both be computed in parallel across all time steps t, and admit efficient sparse representations due to their block-diagonal structure (see Section A.5 below).

Since $H^{I,t}$ is block-diagonal, all products $\Phi_{(\lambda:t)}$ are likewise block-diagonal. The full set $\Phi_{(\lambda:1)}, \ldots, \Phi_{(\lambda:\lambda)}$ can be computed in parallel with time complexity $\mathcal{O}(\log \lambda)$ and memory complexity $\mathcal{O}(\lambda)$ using an associative scan (Blelloch, 1990). This is the parallel-scan procedure referenced in the main text.

Algorithm A1: FBPTT: Forward-Backward Propagation Through Time

Input: Input sequence $X^{1:T}$
Input: Network $\mathcal{N}_\theta$ with parameters θ
Input: Chunk length λ, number of chunks $k = T/\lambda$
Output: Updated parameters θ

$\tilde{G}_{\text{prev}} \leftarrow \mathbf{0}$
$\nabla_\theta L \leftarrow \mathbf{0}$
for $i \leftarrow 0$ **to** $k-1$ **do**
  `// Chunk boundaries`
  $t_{\text{start}} \leftarrow i\lambda + 1, \quad t_{\text{end}} \leftarrow (i+1)\lambda$
  `/* (1) Forward pass: compute states, outputs, and losses */`
  for $t \leftarrow t_{start}$ **to** t_{end} **do**
    $\mathbf{h}^t \leftarrow f_\theta(\mathbf{h}^{t-1}, \mathbf{x}^t, \mathbf{y}^{t-1})$
    $\mathbf{y}^t \leftarrow g(\mathbf{h}^t)$
    $L^t \leftarrow \texttt{loss}(\mathbf{y}^t, \hat{y}^t)$
  `/* (2) Forward pass: update approximate sensitivity matrix` $\tilde{G}$ `(shown sequentially; computed in parallel via associative scan, Section A.5.3) */`
  $\tilde{G}^{t_{\text{start}}-1} \leftarrow \tilde{G}_{\text{prev}}$
  for $t \leftarrow t_{start}$ **to** t_{end} **do**
    $H^{I,t} \leftarrow \left.\frac{\partial \mathbf{h}^t}{\partial \mathbf{h}^{t-1}}\right|_{\text{block-diag}}$
    $F^t \leftarrow \frac{\partial \mathbf{h}^t}{\partial \theta^t}$
    $\tilde{G}^t \leftarrow H^{I,t}\,\tilde{G}^{t-1} + F^t$
  `/* (3) Backward pass: truncated BPTT within chunk */`
  $\tilde{\boldsymbol{\delta}}^{t_{\text{end}}} \leftarrow \frac{\partial L^{t_{\text{end}}}}{\partial \mathbf{h}^{t_{\text{end}}}}$
  for $t \leftarrow t_{end} - 1$ **to** t_{start} **do**
    $\tilde{\boldsymbol{\delta}}^t \leftarrow \frac{\partial L^t}{\partial \mathbf{h}^t} + (H^{t+1})^\top \tilde{\boldsymbol{\delta}}^{t+1}$
  `/* (4) Accumulate gradients */`
  for $t \leftarrow t_{start}$ **to** t_{end} **do**
    $\nabla_\theta L \mathrel{+}= \tilde{\boldsymbol{\delta}}^t F^t$
  if $i > 0$ **then**
    $\nabla_\theta L \mathrel{+}= \tilde{\boldsymbol{\delta}}^{t_{\text{start}}}\, H^{I,t_{\text{start}}}\, \tilde{G}_{\text{prev}}$
    `// long-term correction at chunk boundary`
  $\tilde{G}_{\text{prev}} \leftarrow \tilde{G}^{t_{\text{end}}}$
    `// carry sensitivity matrix to next chunk`
$\theta \leftarrow \texttt{optimizer}(\theta, \nabla_\theta L)$
return θ

Equation (A39) has two practical advantages over the sequential recurrence. First, all terms in the sum are independent of each other and can be evaluated in parallel on a GPU. Second, it avoids repeatedly reading and writing the full sensitivity matrix $\tilde{G}^t$ to memory at every time step. Instead, one collects the compact factors F^t across all steps and combines them in a single operation at the end of the chunk, substantially reducing memory overhead.

A.6 PSEUDOCODE

Algorithm A1 summarizes FBPTT. At each chunk i, the algorithm proceeds in three stages. First, a forward pass computes the hidden states $\mathbf{h}^t$, outputs $\mathbf{y}^t$, and local losses L^t for all time steps in the chunk. Second, the approximate sensitivity matrix $\tilde{G}^t$ is updated forward in time using only the block-diagonal internal Jacobian $H^{I,t}$, accumulating long-range parameter sensitivity across chunk boundaries. Third, the truncated error signal $\tilde{\boldsymbol{\delta}}^t$ is backpropagated within the chunk using the full Jacobian H^t, and reset at each chunk boundary. The gradient accumulator receives two contributions: the standard TBPTT term $\tilde{\boldsymbol{\delta}}^t F^t$ summed over all steps in the chunk, and the long-term correction term $\tilde{\boldsymbol{\delta}}^{t_{\text{start}}} H^{I,t_{\text{start}}} \tilde{G}^{t_{\text{start}}-1}$ at the start of each chunk except the first. $\tilde{G}$ is carried forward across chunk boundaries and never reset, while $\tilde{\boldsymbol{\delta}}$ is restarted at each chunk. After all k chunks are processed, the accumulated gradient is passed to the optimizer.

For clarity, Algorithm A1 presents the update of $\tilde{G}^t$ in stage (2) as a sequential for-loop, matching the recurrence in Eq. (A21). In practice, this loop is not executed sequentially: as derived in Section A.5.3, the block-diagonal structure of $H^{I,t}$ allows the entire sequence of $\tilde{G}^t$ values within a chunk to be computed in parallel via an associative scan, in $\mathcal{O}(\log \lambda)$ time rather than $\mathcal{O}(\lambda)$. The sequential form shown here is mathematically equivalent and is used only to keep the recursion easy to follow.

A.7 COMPUTATIONAL COMPLEXITY

Table A1 summarizes the time and memory complexity of the proposed method in comparison with full BPTT, RTRL, truncated BPTT (TBPTT), and HYPR. We denote by T the sequence length, λ the chunk length, m the number of neurons, k the internal state dimension per neuron, and p the number of parameters per neuron. The total hidden state size is $n = mk$ and the total parameter size is mp.

Table A1: Computational complexity comparison. T: sequence length, λ: chunk length, m: number of neurons, k: internal state dimension, p: parameters per neuron.

Method	**Time**	**Memory**	**Long-range deps.**	**External recurrence**
Full BPTT	$\mathcal{O}(T \cdot m^2k^2)$	$\mathcal{O}(T \cdot mk)$	✓	✓
RTRL	$\mathcal{O}(T \cdot m^3k^2p)$	$\mathcal{O}(m^2kp)$	✓	✓
TBPTT	$\mathcal{O}(T \cdot m^2k^2)$	$\mathcal{O}(\lambda \cdot mk)$	×	✓
HYPR	$\mathcal{O}(T \cdot mk^2p)$	$\mathcal{O}(mkp)$	✓	×
FBPTT	$\mathcal{O}\big(T \cdot (m^2k^2 + mk^2p)\big)$	$\mathcal{O}(\lambda \cdot mk + mkp)$	✓	✓

The time complexity of full BPTT is $\mathcal{O}(T \cdot m^2k^2)$, dominated by the matrix-vector products in the backward recursion, but it requires storing all T hidden states simultaneously, giving $\mathcal{O}(T \cdot mk)$ memory. RTRL maintains the full sensitivity matrix $G^t \in \mathbb{R}^{mk \times mp}$ and updates it at every step, incurring $\mathcal{O}(m^3k^2p)$ time per step and $\mathcal{O}(m^2kp)$ memory, making it computationally impractical for large networks. TBPTT reduces memory to $\mathcal{O}(\lambda \cdot mk)$ by truncating the backward recursion to λ steps, but loses all gradient information beyond the chunk boundary.

HYPR propagates only the internal (block-diagonal) sensitivity $\tilde{G}^t$. Each per-neuron update $H_i^{I,t}\tilde{G}_i^{t-1}$ multiplies a $k \times k$ block by a $k \times p$ block, costing $\mathcal{O}(k^2p)$ per neuron and $\mathcal{O}(mk^2p)$ per step across all m neurons; summed over the sequence this gives $\mathcal{O}(T \cdot mk^2p)$ total compute, which HYPR parallelizes across time steps within a chunk via an associative scan (Blelloch, 1990), reducing the critical path to $\mathcal{O}(\log \lambda)$ parallel steps. Its memory cost is $\mathcal{O}(mkp)$, since $\tilde{G}^t$ consists of m independent $k \times p$ blocks.

The proposed method combines TBPTT with the long-term correction term $\tilde{\boldsymbol{\delta}}^{t_{\text{start}}} \tilde{G}^{t_{\text{start}}-1}$ at each chunk boundary. Its time complexity combines the $\mathcal{O}(T \cdot m^2k^2)$ cost of the local TBPTT backward pass within each chunk with the $\mathcal{O}(T \cdot mk^2p)$ cost of maintaining $\tilde{G}^t$ across the full sequence, giving $\mathcal{O}\big(T \cdot (m^2k^2 + mk^2p)\big)$. The memory cost is $\mathcal{O}(\lambda \cdot mk + mkp)$: the first term comes from storing

hidden states within the current chunk for the backward pass, and the second from maintaining the block-diagonal $\tilde{G}^t$ across chunk boundaries.

For standard RNNs, the number of parameters per neuron p is dominated by the recurrent connection weights, which is $\Theta(m)$. Using this, we can simplify the asymptotic bounds, as shown in Table A2: in particular, the FBPTT time complexity $\mathcal{O}\big(T \cdot (m^2k^2 + mk^2p)\big)$ reduces to $\mathcal{O}(T \cdot m^2k^2)$, since $mk^2p = \Theta(m^2k^2)$ under this substitution. We see that FBPTT matches the time complexity of both TBPTT and HYPR. In terms of memory, one can observe a tradeoff between chunk length λ and layer size m. For λ much smaller than m, it can be considered an improved version of HYPR with negligible memory overhead. For λ much larger than m, it can be viewed as an improved version of TBPTT which respects long-range dependencies with negligible memory overhead. For $\lambda = \Theta(m)$, it maintains the order of the time and memory complexity of both while combining their computational advantages.

Table A2: Computational complexity comparison for $p = \Theta(m)$. T: sequence length, λ: chunk length, m: number of neurons, k: internal state dimension, p: parameters per neuron.

Method	Time	Memory	Long-range deps.	External recurrence
Full BPTT	$\mathcal{O}(T \cdot m^2k^2)$	$\mathcal{O}(T \cdot mk)$	✓	✓
RTRL	$\mathcal{O}(T \cdot m^4k^2)$	$\mathcal{O}(m^3k)$	✓	✓
TBPTT	$\mathcal{O}(T \cdot m^2k^2)$	$\mathcal{O}(\lambda \cdot mk)$	×	✓
HYPR	$\mathcal{O}(T \cdot m^2k^2)$	$\mathcal{O}(m^2k)$	✓	×
FBPTT	$\mathcal{O}(T \cdot m^2k^2)$	$\mathcal{O}(\lambda \cdot mk + m^2k)$	✓	✓

A.8 Neuron models

In this section, we describe the dynamics of the neuron models used throughout our experiments. We consider a range of recurrent units, spanning from standard gated architectures to biologically-inspired spiking models with oscillatory dynamics. For each model, we detail the underlying update equations and highlight the mechanisms governing information retention and temporal processing.

A.8.1 Long Short-Term Memory (LSTM)

The LSTM (Hochreiter & Schmidhuber, 1997) neuron is a recurrent neural network unit designed to model long-range dependencies in sequential data using gated memory mechanisms. It employs input, forget, and output gates to regulate the flow of information through a persistent cell state, allowing the network to retain relevant information over long time intervals while mitigating the vanishing gradient problem.

In an LSTM cell, the state at time t consists of a cell state c_i^t and a hidden state h_i^t. The cell state is updated using an input gate ι_i^t and a forget gate ϕ_i^t, which determine how much new information is written to the cell and how much previous information is retained, respectively. The hidden state is then computed by applying the output gate o_i^t to the updated cell state:

$$c_i^t = \phi_i^t \odot c_i^{t-1} + \iota_i^t \odot \tilde{c}_i^t, \qquad h_i^t = o_i^t \odot \tanh(c_i^t), \tag{A40}$$

where $\tilde{c}_i^t$ is the candidate cell state and $\odot$ denotes element-wise multiplication. The input, forget, and output gates are computed by applying the sigmoid function to affine transformations of the current input x^t and the previous hidden state h_i^{t-1}.

A.8.2 Gated Recurrent Unit (GRU)

The Gated Recurrent Unit (GRU) (Chung et al., 2014) is a simplified recurrent neural network unit that combines the memory and gating mechanisms of the LSTM into a more compact architecture. It employs update and reset gates to regulate information retention and state updates. Compared with the LSTM, the GRU requires fewer parameters while retaining the ability to model long-term temporal dependencies.

In a GRU cell, the state at time t consists only of a hidden state $h_i^t \in \mathbb{R}$. The hidden state is updated using a reset gate r_i^t and an update gate z_i^t:

$$h_i^t = (1 - z_i^t) \odot h_i^{t-1} + z_i^t \odot \tilde{h}_i^t, \tag{A41}$$

where

$$\tilde{h}_i^t = \tanh\big(W_h x^t + U_h(r_i^t \odot h_i^{t-1}) + b_h\big) \tag{A42}$$

is the candidate hidden state. The reset gate controls how much of the previous hidden state contributes to the candidate hidden state, while the update gate determines how much of the candidate state replaces the previous hidden state. Compared with the LSTM, the GRU achieves a similar gating mechanism with fewer parameters by combining the cell state and hidden state into a single hidden state.

A.8.3 Balanced Resonate-and-Fire (BRF)

We adopt the balanced resonate-and-fire neuron introduced by Higuchi et al. (2024), whose state evolves according to

$$b^t = p_\omega - b_{\text{offset}} - q^{t-1} \tag{A43}$$

$$u^t = u^{t-1} + \boldsymbol{\delta t} \cdot (b^t \cdot u^{t-1} - \omega \cdot v^{t-1} + \mathbf{I}^t) \tag{A44}$$

$$v^t = v^{t-1} + \boldsymbol{\delta t} \cdot (\omega \cdot u^{t-1} + b^t \cdot v^{t-1}) \tag{A45}$$

$$z^t = \Theta(u^t - \theta - q^{t-1}) \tag{A46}$$

$$q^t = \alpha \cdot q^{t-1} + z^t. \tag{A47}$$

Each neuron is described by a pair of coupled state variables, the membrane potential u^t and an auxiliary component v^t, which oscillate at an angular frequency ω. The rate at which these oscillations decay is governed by b^t, which combines three contributions: the divergence boundary $p_\omega = \big(-1+\sqrt{1-(\boldsymbol{\delta t}\cdot\omega)^2}\big)/\boldsymbol{\delta t}$, a damping offset b_{offset}, and an adaptation variable q^t. The neuron receives the input current $\mathbf{I}^t$, whose i-th entry is $I_i^t = \mathbf{w}_{\text{ff},i}^\intercal \mathbf{x}^t + \mathbf{w}_{\text{rec},i}^\intercal \mathbf{y}^{t-1} + b_i$, and the dynamics are discretized with a step size of $\boldsymbol{\delta t} = 0.01$ throughout all experiments.

A spike $z^t \in \{0, 1\}$ is emitted whenever u^t crosses the effective threshold $\theta + q^{t-1}$, where θ denotes the baseline threshold. Unlike many spiking neuron models, the BRF neuron does not reset its membrane potential after firing. Spiking activity is instead fed into the adaptation variable q^t, which decays with factor $\alpha = 0.9$ and grows by one with every spike. Because q^t enters both the firing threshold and the damping term b^t, recent activity simultaneously raises the threshold and strengthens the damping of subsequent oscillations. The parameters ω and b_{offset} are learned jointly with the synaptic weights, and all weights are initialized following Higuchi et al. (2024).

A.8.4 Symplectic-Euler adaptive Leaky Integrate-and-Fire (SE-adLIF)

For the adaptive leaky integrate-and-fire neuron, we follow Baronig et al. (2025) and discretize the continuous dynamics with the symplectic Euler method, which yields the update rules

$$\hat{u}^t = \alpha u^{t-1} + (1 - \alpha)(\mathbf{I}^t - w^{t-1}) \tag{A48}$$

$$z^t = \Theta(\hat{u}^t - \theta) \tag{A49}$$

$$u^t = \hat{u}^t(1 - z^t) \tag{A50}$$

$$w^t = \beta w^{t-1} + (1 - \beta)(a u^t + b z^t). \tag{A51}$$

At each step, the membrane potential is first integrated to a provisional value $\hat{u}^t$, driven by the input current $\mathbf{I}^t$ and reduced by the adaptation current w^{t-1}. The i-th component of the input is $I_i^t = \mathbf{w}_{\text{ff},i}^\intercal \mathbf{x}^t + \mathbf{w}_{\text{rec},i}^\intercal \mathbf{y}^{t-1} + b_i$. If $\hat{u}^t$ exceeds the threshold θ, the neuron emits a spike and its potential is reset to zero. The adaptation current w^t is then updated from the post-reset potential and the spike output, which couples subthreshold and spike-triggered adaptation within a single step. The decay factors α and β are determined by the membrane and adaptation time constants τ_u and τ_w, respectively.

The adaptation strengths are parameterized as $a = \rho\hat{a}$ and $b = \rho\hat{b}$, with a fixed scaling factor $\rho = 120$ across all experiments. The trainable parameters $\hat{a}$ and $\hat{b}$ are drawn from $\mathcal{U}(0, 1)$ at initialization, optimized jointly with the remaining network parameters, and constrained to $[0, 1]$ by clipping throughout training. The time constants are parameterized by the interpolation scheme of Baronig et al. (2025), restricted to $\tau_u \in [5, 25]$ and $\tau_w \in [60, 300]$ in every experiment. Weight initialization also follows Baronig et al. (2025).

A.8.5 Harmonic Oscillator Recurrent Network (HORN)

The HORN (Effenberger et al., 2025) neuron models a damped, driven harmonic oscillator (DHO), discretized using symplectic (semi-implicit) Euler integration. Each node maintains two state variables, a position x_i^t and a velocity y_i^t, which are updated as:

$$y_i^t = y_i^{t-1} + h \cdot \left(\alpha \tanh(I_i^t) - \omega^2 x_i^{t-1} - 2\gamma y_i^{t-1}\right) \tag{A52}$$

$$x_i^t = x_i^{t-1} + h \cdot y_i^t \tag{A53}$$

where h is the integration step size, α is the forcing gain, ω is the natural (angular) frequency, and γ is the damping coefficient of the oscillator. The input vector $\mathbf{I}^t$ has components $I_i^t = \mathbf{w}_{\text{ff},i}^\intercal \mathbf{x}^t + g_{\text{rec}} \cdot \mathbf{w}_{\text{rec},i}^\intercal \mathbf{y}^{t-1} + b_i$, where $g_{\text{rec}} = 1/\sqrt{N}$ is a fixed recurrent gain factor that normalizes the recurrent drive by the number of nodes N. Unlike the LSTM and GRU, the HORN neuron has no explicit gating mechanism; instead, oscillatory and stable dynamics arise directly from the symplectic discretization of the DHO equation, in which the velocity y_i^t is updated first from the previous position x_i^{t-1}, and the position x_i^t is then updated using the newly computed velocity y_i^t. As with $\boldsymbol{\delta}t$ in the BRF and SE-adLIF models, the hyperparameters $h, \alpha, \omega, \gamma$ are kept fixed for all nodes and are not trained together with the network weights. The network is unrolled by feeding one input vector $\mathbf{x}^t$ per time step, and the final network output is read out linearly from the position state x_i^t at the last time step.

A.9 Datasets

We used the predefined train/test splits provided. The training data was then further divided into a training and validation subset, allocated at a $90\%/10\%$ ratio, with the validation subset used for model selection. For the *Pathfinder-E* dataset, which lack predefined splits, we instead partition the data into training, validation, and test sets using an $80\%/10\%/10\%$ split.

A.9.1 State tracking task

The state tracking task is based on a Moore machine (Gill, 1965) and tests a model's ability to track a hidden discrete state $s_t \in \{0, \ldots, n-1\}$ over long sequences, where n is the number of states. At each time step, the model receives a one-hot encoded action a_t and must predict the current state, with accuracy evaluated on the final state of the sequence. Four actions are available: *up*, *down*, *stay*, and *mirror*. The *up* and *down* actions increment or decrement the state by one, *stay* leaves it unchanged, and *mirror* reflects the state around the center of the state space.

At the boundaries of the state space, the *up* and *down* actions are saturated: applying *up* in the maximum state $s_{t-1} = n - 1$, or *down* in the minimum state $s_{t-1} = 0$, leaves the state unchanged. Formally, the state transition induced by each action is

$$f_{\text{up}}(s_{t-1}) = \min(n - 1,\ s_{t-1} + 1), \tag{A54}$$

$$f_{\text{down}}(s_{t-1}) = \max(0,\ s_{t-1} - 1), \tag{A55}$$

$$f_{\text{stay}}(s_{t-1}) = s_{t-1}, \tag{A56}$$

$$f_{\text{mirror}}(s_{t-1}) = |s_{t-1} - n + 1| = n - 1 - s_{t-1}. \tag{A57}$$

The state after t steps is obtained by composing the transitions of all actions in the sequence,

$$s_t = f_{a_t}\big(f_{a_{t-1}}(\cdots f_{a_1}(s_0))\big), \tag{A58}$$

where f_{a_t} denotes the transformation that action a_t induces on the state s_{t-1}. The initial state is $s_0 = n/2$ (we always choose an even n).

The challenge lies in learning the long-range cumulative effect of actions. Mirror operation at the boundaries make the effect of an action depend on the current state, it reverses the position of the state. As a result, the final state cannot be obtained by simply summing the actions, and the model must maintain an accurate internal representation of the state over the full sequence. Task difficulty scales with the number of states n and the sequence length.

A.9.2 ADD TASK

In the adding problem, each training example is a sequence

$$z_t = (x_t, m_t), \qquad t = 1, \dots, L. \tag{A59}$$

The first component x_t is a real value and the second component m_t is a marker. Exactly two positions are marked as relevant, and the model must output, at the end of the sequence, a scaled sum of the two relevant values. This is a sequence-to-one regression problem.

The first component is sampled from

$$x_t \sim \mathcal{U}[0, 1], \tag{A60}$$

and the marker takes values in

$$m_t \in \{0, 1\}. \tag{A61}$$

Each sequence contains exactly two marked values: one index i_1 sampled uniformly from the first half of the sequence, $i_1 \sim \mathcal{U}\{1, \dots, L/2\}$, and one index i_2 sampled uniformly from the second half, $i_2 \sim \mathcal{U}\{L/2 + 1, \dots, L\}$, with $m_{i_1} = m_{i_2} = 1$ and $m_t = 0$ otherwise. The target output is

$$y = 0.5 + \frac{x_{i_1} + x_{i_2}}{4}, \tag{A62}$$

so that $y \in [0.5, 1]$.

The loss is computed only at the final time step:

$$\mathcal{L} = (\hat{y}_L - y)^2. \tag{A63}$$

A sequence is considered correctly processed if

$$|\hat{y}_L - y| < 0.04. \tag{A64}$$

Training and test sequences of length L are generated online, with a fresh batch sampled independently at every step; no fixed dataset is stored or reused. Test accuracy is reported over a freshly sampled held-out set of sequences, using $L = 5000$ in the experiments shown in Fig. 3.

A.9.3 NEUROMORSE

The NeuroMorse dataset (Walters et al., 2025), is a temporally structured benchmark for neuromorphic computing. NeuroMorse encodes the top 50 most frequent words in the English language as Morse code spike sequences, using only two input channels corresponding to dots and dashes. Despite this minimal spatial dimensionality, the dataset embeds rich temporal structure: words are distinguished purely by the timing and ordering of spikes across three characteristic timescales, between consecutive dots/dashes, between consecutive characters, and between consecutive words. This places nearly all task-relevant information in the temporal domain rather than the spatial (channel) domain, making NeuroMorse a useful stress test for a network's ability to exploit long-range temporal dependencies. We use the NeuroMorse training set, in which each of the 50 words is presented as an isolated spike sequence with an associated class label, and evaluate our method on this classification task.

A.9.4 PATHFINDER-E

Pathfinder (Tay et al., 2020) is a synthetic binary classification benchmark in which the model must determine whether two marked endpoints in a grayscale image are connected by a continuous curved path. Positive samples contain a valid connecting path, while negative samples contain visually similar distractor contours with no such connection. Since the correct decision requires tracing a path across large spatial distances, the task provides a challenging test of long-range dependency

Table A3: Test accuracy (%) of HORN networks on benchmark tasks (mean ± SD over 10 runs). Bold numbers indicate the best result out of {HYPR, 2-chunk TBPTT, 2-chunk FBPTT} and {HYPR, 10-chunk TBPTT, 10-chunk FBPTT}.

Method	**#chunks**	sMNIST	psMNIST	sCIFAR
BPTT	–	60.74±0.24	54.12±1.05	30.51±0.59
TBPTT	2	41.38±0.68	33.18±0.28	25.71±0.29
	10	32.41±0.23	25.08±0.93	17.73±0.18
HYPR	–	23.11±1.07	14.67±1.08	9.76±0.48
FBPTT	2	**43.36±0.59**	**37.47±0.68**	**28.69±0.47**
(ours)	10	**35.41±0.96**	**27.67±0.79**	**20.24±0.91**

modeling. In the Long Range Arena (LRA) formulation, images are flattened into pixel sequences, transforming the problem into a sequential long-context task. The Pathfinder dataset is provided at three difficulty levels, which differ in the length of the connecting path and of the distractor contours: easy, intermediate, and hard. In this work, we use the easy variant, which is called Pathfinder-E. It consists of 32×32 images, resulting in input sequences of length 1024, and same task structure and sequential formulation as the standard benchmark while using shorter contours.

A.9.5 sMNIST

MNIST (LeCun & Cortes, 2005) is a standard image classification dataset consisting of 28×28 grayscale images of handwritten digits from 10 classes, with 60,000 training and 10,000 test samples. We use its sequential variant, sequential MNIST (S-MNIST) (Yin et al., 2021), in which the pixels of each image are presented to the model one at a time, resulting in a sequence of length 784. The model must classify the image into one of the 10 digit categories after processing the full sequence. Pixel values are normalized to $[0, 1]$.

We additionally evaluate on permuted sequential MNIST (PS-MNIST) (Yin et al., 2021), in which a single fixed random permutation is applied to the pixel order of every image before it is presented as a sequence. The same permutation is used for all training and test samples. As a result, the task requires the model to integrate information across long time and is considered substantially harder than S-MNIST.

A.9.6 sCIFAR

The sequential CIFAR-10 (sCIFAR) task is based on the CIFAR-10 image classification dataset (Krizhevsky, 2009). It presents the pixels of a 32×32 color image as a sequence of length 1024, with three input channels (RGB). The model must classify the image into one of 10 categories after processing the full sequence. Pixel values are normalized to $[0, 1]$. A grayscale version of this task is part of the Long Range Arena benchmark (Tay et al., 2020); we use the color version, which provides three input channels per time step.

A.10 Extended results

We evaluated our method on the HORN model (Effenberger et al., 2025), which is based oscillatory recurrent units, on three standard sequential image-classification benchmarks: sequential MNIST (sMNIST), permuted sequential MNIST (psMNIST), and sequential CIFAR-10 (sCIFAR). These tasks require propagating gradient information over hundreds to thousands of timesteps, making them a natural stress test for the long-range credit assignment properties of truncated and approximate gradient methods. For TBPTT and FBPTT we report results at two chunk counts (2 and 10).

Results are shown in Table A3. As expected, full BPTT achieved the strongest performance across all three benchmarks, since it keeps exact gradient information over the entire sequence. HYPR, which discards all intra-layer gradient flow in favor of local block-diagonal sensitivities, performed worst. TBPTT fell in between, but its accuracy degraded sharply as the number of chunks increases from 2 to 10: with more, shorter chunks, gradients are truncated at increasingly short intervals, and the network fails to credit-assign errors to inputs from distant timesteps.

FBPTT consistently outperformed TBPTT at matched chunk counts across all three tasks and both settings (2 and 10 chunks), confirming that the long-term correction term recovers part of the gradient signal that TBPTT discards at chunk boundaries.

A.11 Gradient Analysis

This experiment evaluates how different gradient estimation strategies for recurrent neural networks approximate the true sequence gradient, under matched model parameters and inputs. We compared gradients from approximation methods (TBPTT, FBPTT, HYPR) with the exact gradients from BPTT. The central question is whether FBPTT approximates full BPTT more closely than HYPR and TBPTT, by recovering gradient information lost at truncation boundaries.

A.11.1 Experimental Setup

We used the same network architecture as in the hidden state tracking task, evaluated on both the spiking (SNN) and non-spiking (ANN) neuron models. For a given gradient computation, model parameters were held fixed and identical across all gradient estimation methods: no parameter optimization was performed at this stage. The model was used solely for forward propagation of inputs, loss evaluation, and gradient computation with respect to the fixed parameters. This isolates differences arising purely from how each method assigns temporal credit, not from learning dynamics. Gradients were computed on the hidden state tracking task: after processing the full input sequence, the model's predictions are compared against target labels to obtain a scalar loss $\mathcal{L}$, which is used for all gradient computations described below.

A.11.2 Initialization

Rather than evaluating gradients at a single, arbitrarily chosen parameter configuration, we assessed each method's behavior across multiple initializations drawn from real training trajectories. While training the network with each of TBPTT, HYPR, and our FBPTT method, we periodically saved model checkpoints at different epochs. Each checkpoint defines a distinct, trained set of weights, which we then use as the fixed initialization for the gradient comparison described above: parameters were frozen at that checkpoint, and the forward pass, loss evaluation, and gradient computation proceeded as in the setup above, without further training. For each such initialization, we computed the gradient under all three approximate methods (TBPTT, HYPR, FBPTT) as well as the full-BPTT reference, and calculated cosine similarity between each approximate gradient and the reference. This was repeated separately for each neuron model (SNN and ANN), giving a distribution of similarity scores per method rather than a single point estimate, and avoiding conclusions that depend on the choice of initialization or training method.

A.11.3 Gradient Representation

For each method, the gradient is returned in the same structure as the model parameters: each parameter tensor has a corresponding gradient tensor,

$$g = \{g^{(1)}, g^{(2)}, \ldots, g^{(m)}\}.$$

To compare gradients numerically across methods, all gradient tensors were flattened and concatenated into a single vector:

$$\mathbf{g} = [g_1, g_2, \ldots, g_N],$$

where N is the total number of scalar trainable parameters in the model (equivalently, the total number of scalar gradient entries after flattening and concatenation). Each entry g_i is the gradient of the loss with respect to one scalar parameter.

The analysis and results reported in this section, including Figure A1, are restricted to the *weight parameters* of the network (the synaptic weight matrices). Bias terms and neuron parameters (e.g., time constants or thresholds in the SNN case) were excluded from this comparison.

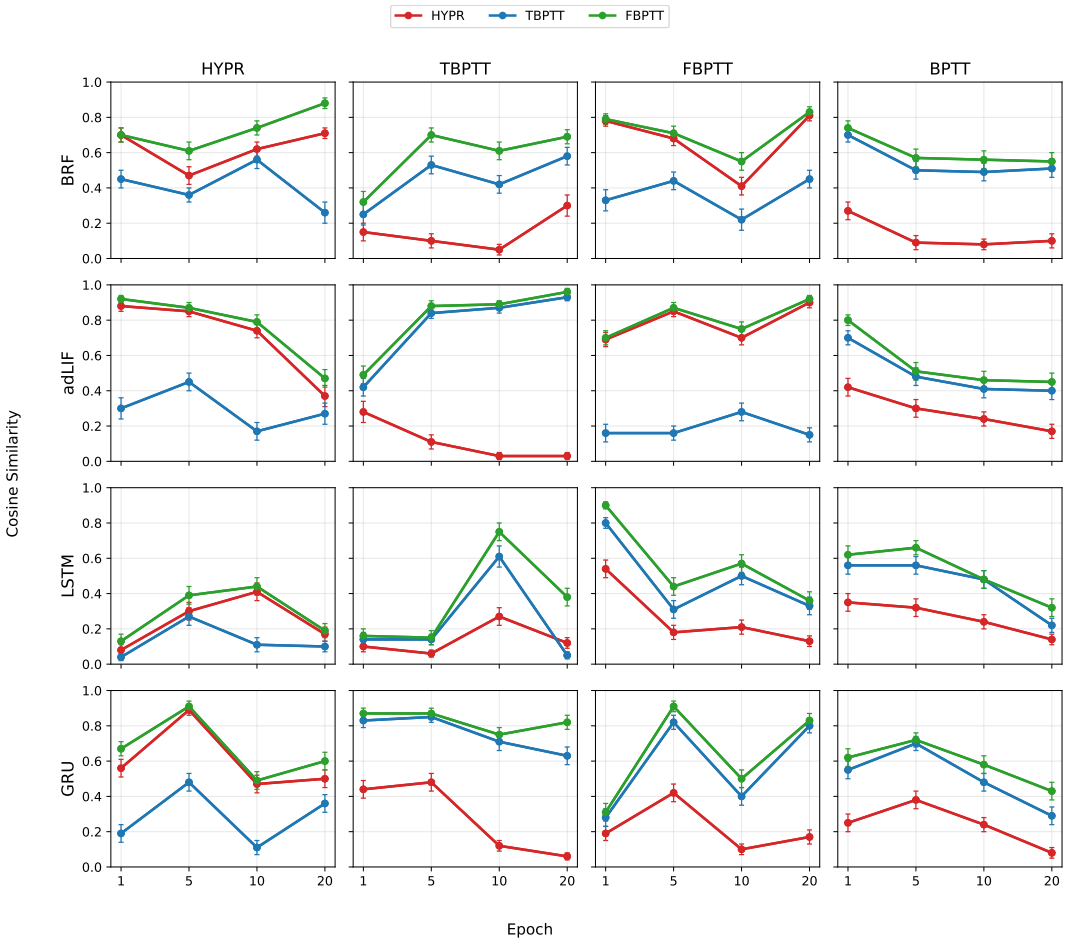


Figure A1: Gradient alignment on the state tracking task (sequence length 400) for all neuron models. Each network model (rows) was trained with each gradient evaluation method (columns). Each graph shows the cosine similarity between the weight gradients of BPTT and HYPR (red), TBPTT (blue), and FBPTT (green) at the given epoch. For TBPTT and FBPTT, chunk length was 10.

A.11.4 Gradient Comparison Metrics

Each estimated gradient was compared against the full-BPTT reference using cosine similarity

$$\cos(\mathbf{g}_A, \mathbf{g}_B) = \frac{\mathbf{g}_A \cdot \mathbf{g}_B}{\|\mathbf{g}_A\| \, \|\mathbf{g}_B\|}, \tag{A65}$$

where $\mathbf{g}_A, \mathbf{g}_B$ are two gradient vectors. This measures directional alignment between an approximate gradient and the reference, independent of magnitude.

A.11.5 Gradient Analysis for all network models

The result of the gradient analysis for all network models is shown in Figure A1

A.12 Training details

A.12.1 Surrogate gradients

Since the spiking activation function $\Theta(x)$ is non-differentiable, we replace its derivative with a smooth surrogate during the backward pass in the SNN models. Three surrogate gradient shapes are used across our experiments. The SLAYER surrogate (Shrestha & Orchard, 2018) approximates the

derivative with a scaled exponential,

$$\frac{d\Theta(x)}{dx} \approx \alpha\, c\, e^{-\alpha|x|}, \tag{A66}$$

with sharpness $\alpha = 5$ and amplitude $c = 0.2$. The double-Gaussian surrogate (Yin et al., 2021) instead subtracts a wide Gaussian from a narrow one,

$$\frac{d\Theta(x)}{dx} \approx \gamma\Big[(1+p)\,\mathrm{G}(x;0,\sigma_1) - 2p\,\mathrm{G}(x;0,\sigma_2)\Big], \tag{A67}$$

where $\mathrm{G}(x;\mu,\sigma)$ is the Gaussian density function, $\sigma_1 = 0.5$, $\sigma_2 = 6\sigma_1$, $p = 0.15$, and $\gamma = 0.5$. Finally, for the NeuroMorse experiments we use a fast-sigmoid surrogate,

$$\frac{d\Theta(x)}{dx} \approx \frac{1}{(1+s|x|)^2}, \tag{A68}$$

with slope parameter $s = 15$. The surrogate used for each model and dataset is listed in Section A.13.

A.12.2 Optimizer

All models were trained with Adam (Kingma, 2014) ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon =$ 1e-8); for the LSTM and GRU baselines on sCIFAR we instead used AdamW with decoupled weight decay of 10^{-4}, which we found to stabilize training for these models. Learning rates were tuned individually per model and dataset and are reported in Section A.13. To prevent instabilities from occasional large gradients, we clipped the gradient norm to a maximum value m_{grad} whenever it was exceeded; the corresponding values are also given in Section A.13.

A.12.3 Loss Functions

At each time step t, the readout layer generates a vector $\bar{\mathbf{y}}^t \in \mathbb{R}^C$, where C is the output dimensionality: the number of classes in classification settings, or the dimension of the scalar target in the adding task. Depending on the task and architecture (see Section A.13), the output sequence $\bar{\mathbf{y}}^1, \ldots, \bar{\mathbf{y}}^T$ is converted into a training loss using one of three formulations. In all of them, the first t_0 steps serve as a warm-up phase whose outputs are left out of the loss; the value of t_0 is set separately for each task and model.

The first formulation, which we call the *accumulated-softmax* loss, converts the output at every step into class probabilities, sums these probabilities over the evaluation window, and compares the normalized result to the one-hot label $\mathbf{y}^* \in \mathbb{R}^C$ via cross entropy:

$$\mathcal{L} = \mathrm{CE}\left(\mathrm{softmax}\Big(\sum_{t=t_0+1}^{T} \mathrm{softmax}(\bar{\mathbf{y}}^t)\Big), \mathbf{y}^*\right). \tag{A69}$$

The second formulation evaluates the cross entropy separately at each step and then takes the temporal average:

$$\mathcal{L} = \frac{1}{T-t_0} \sum_{t=t_0+1}^{T} \mathrm{CE}\big(\mathrm{softmax}(\bar{\mathbf{y}}^t), \mathbf{y}^{*t}\big). \tag{A70}$$

Here the target $\mathbf{y}^{*t}$ may either be identical for all steps or vary over time, depending on the task.

Finally, since the adding task requires predicting a continuous value rather than a class, we train it with a squared-error objective averaged over the evaluation window:

$$\mathcal{L} = \frac{1}{T-t_0} \sum_{t=t_0+1}^{T} \big(\bar{y}^t - y^{*t}\big)^2. \tag{A71}$$

A.12.4 Class Prediction

To obtain a class label $\hat{y}$ from the sequence of readout vectors $\bar{\mathbf{y}}^1, \ldots, \bar{\mathbf{y}}^T$, we apply the decision rule that corresponds to the training objective of each model. Three variants are considered. The

Table A4: List of hyperparameters corresponding to the **BRF** model simulations on the PATHFINDER and SCIFAR datasets.

	PathFinder	**sCIFAR**
learning rate (lr)	0.002	0.02
lr scheduler	cosine	cosine
optimizer	Adam	Adam
nb. layers	1	1
nb. neurons	256	256
epochs	100	100
batch size	32	32
parameter initialization	$\omega \in \mathcal{U}(15, 50)$, $b \in \mathcal{U}(0.0001, 0.1)$	$\omega \in \mathcal{U}(15, 50)$, $b \in \mathcal{U}(0.0001, 0.1)$
loss aggregation	per timestep CE	per timestep CE
prediction mode	mean over sequence	mean over sequence
t_0	1023	300
surrogate gradient	double Gaussian	SLAYER
grad. clipping m_{grad}	1	1

Table A5: List of hyperparameters corresponding to the **BRF** model simulations on the NEUROMORSE and PSMNIST datasets.

	NeuroMorse	**psMNIST**
learning rate (lr)	0.001	0.01
lr scheduler	constant	linear
optimizer	Adam	Adam
nb. layers	2	1
nb. neurons	128, 256	256
epochs	2000	100
batch size	50	64
parameter initialization	$\omega \in \mathcal{U}(5, 10)$, $b \in \mathcal{U}(0.1, 1.0)$	$\omega \in \mathcal{U}(15, 50)$, $b \in \mathcal{U}(0.1, 1.0)$
loss aggregation	per timestep CE	per timestep CE
prediction mode	mean over sequence	mean over sequence
t_0	0	500
surrogate gradient	fast sigmoid (slope 15)	Double Guassian
grad. clipping m_{grad}	10	1

first accumulates class probabilities over the evaluated time window,

$$\hat{y} = \arg\max_c \sum_{t=t_0+1}^{T} \big[\text{softmax}(\bar{\mathbf{y}}^t)\big]_c, \tag{A72}$$

the second first averages the raw outputs over the same window and then selects the largest entry,

$$\hat{y} = \arg\max_c \Big[\frac{1}{T-t_0} \sum_{t=t_0+1}^{T} \bar{\mathbf{y}}^t\Big]_c, \tag{A73}$$

and the third relies exclusively on the output at the final step,

$$\hat{y} = \arg\max_c \big[\bar{\mathbf{y}}^T\big]_c. \tag{A74}$$

In other words, the second rule performs temporal averaging prior to classification, whereas the third ignores all intermediate outputs. We use the final-step rule for the LSTM and GRU baselines on sCIFAR as well as for the adding task.

A.13 HYPERPARAMETERS AND TUNING

Table A6: List of hyperparameters corresponding to the **SE-adLIF** model simulations on the NEUROMORSE, SCIFAR, and PSMNIST datasets.

	NeuroMorse	**sCIFAR**	**psMNIST**
learning rate (lr)	0.001	0.001	0.001
lr scheduler	cosine	constant	linear
optimizer	Adam	Adam	Adam
nb. layers	2	2	2
nb. neurons	256	360	360
epochs	300	300	300
batch size	50	512	128
loss aggregation	per timestep CE	per timestep CE	sum of softmax
prediction mode	mean over sequence	mean over sequence	sum of softmax
t_0	0	500	500
grad. clipping m_{grad}	1	1	5

Table A7: List of hyperparameters corresponding to the **LSTM** model simulations on the SCIFAR and PSMNIST datasets.

	sCIFAR	**psMNIST**
learning rate (lr)	0.0003	0.001
lr scheduler	cosine	constant
optimizer	AdamW (wd 1e-4)	Adam
nb. layers	1	2
nb. neurons	256	256
dropout	0.3	–
epochs	100 (patience 30)	100
batch size	64	256
loss aggregation	per timestep CE	per timestep CE
prediction mode	last timestep	mean over sequence
t_0	900	0
grad. clipping m_{grad}	0.25	5

Table A8: List of hyperparameters corresponding to the **GRU** model simulations on the SCIFAR and PSMNIST datasets.

	sCIFAR	**psMNIST**
learning rate (lr)	0.0003	0.001
lr scheduler	cosine	constant
optimizer	AdamW (wd 1e-4)	Adam
nb. layers	1	2
nb. neurons	256	256
dropout	0.3	–
epochs	300 (patience 12)	100
batch size	128	128
loss aggregation	per timestep CE	per timestep CE
prediction mode	last timestep	mean over sequence
t_0	900	0
grad. clipping m_{grad}	1	5

Table A9: List of hyperparameters corresponding to the **LSTM** and **GRU** model simulations on the ADDING TASK dataset.

	LSTM	**GRU**
learning rate (lr)	0.001	0.001
lr scheduler	constant	constant
optimizer	Adam	Adam
nb. layers	2	2
nb. neurons	128	128
epochs	100	100
batch size	128	128
loss aggregation	per timestep MSE	per timestep MSE
prediction mode	last timestep	last timestep
grad. clipping m_{grad}	1	1