A Supplementary Section A

A.1 Computation of Lyapunov Exponents

We compute LE by adopting the well-established algorithm $[57, 58]$ and follow the implementation in $[48, 55]$. For a particular task, each batch of input sequences is sampled from a set of fixed-length sequences of the same distribution. We chose this set to be the validation set. For each input sequence in a batch, a matrix $\mathbf{Q}$ is initialized as the identity to represent an orthogonal set of nearby initial states. The hidden states h_t are initialized as zeros.

To track the expansion and the contraction of the vectors of $\mathbf{Q}$, the Jacobian of the hidden states at step $\mathrm{t}, \mathbf{J}_t$, is calculated and then applied to the vectors of $\mathbf{Q}$. The Jacobian $\mathbf{J}_t$ can be found by taking the partial derivatives of the RNN hidden states at time t, h_t, with respect to the hidden states at time at $t-1, h_{t-1}$

$$[\mathbf{J}_t]_{ij} = \frac{\partial \mathbf{h}_t^j}{\partial \mathbf{h}_{t-1}^i}.$$

Beyond the hidden states, the Jacobian will depend on the input x_t. This dependence allows us to capture the dynamics of a network as it responds to input. The expansion factor of each vector is calculated by updating $\mathbf{Q}$ by computing the QR decomposition at each time step.

$$\mathbf{Q}_{t+1}, \mathbf{R}_{t+1} = QR\left(\mathbf{J}_t \mathbf{Q}_t\right).$$

If r_t^i is the expansion factor of the i^{th} vector at time step t - corresponding to the i^{th} diagonal element of $\mathbf{R}$ in the QR decomposition- then the i^{th} LE λ_i resulting from an input signal of length T is given by

$$\lambda_k = \frac{1}{T} \sum_{t=1}^{T} \log\left(r_t^k\right)$$

The LE resulting from each input x^m in the batch of input sequences is calculated in parallel and then averaged. For each experiment, the LE were calculated over a fixed number of time steps with n different input sequences. The mean of n resulting LE spectra is reported as the LE spectrum. To normalize the spectra across different network sizes and, consequently the number of LE in the spectrum, we interpolate the spectrum such that it retains the shape of the largest network size.

We follow the principles outlined by Engelken et al. for calculating the Lyapunov spectrum of the discrete-time firing-rate network. Conducting numerical simulations with small perturbations in Recurrent Spiking Neural Networks (RSNNs), particularly in the context of discrete spiking events, requires a specific approach to capture these networks' dynamics accurately.

1. **Network Initialization:** Set up an RSNN with a defined architecture, synaptic weights, and initial neuronal states.
2. **Creating Perturbations**: Generate a slightly perturbed version of the network. This could involve minor adjustments to the initial membrane potentials or other state variables of a subset of neurons.
3. **Simulating Network Dynamics:** Run parallel simulations of the original and the perturbed RSNNs, ensuring they receive identical input stimuli. Since the networks operate on discrete spike events, the state of each neuron is updated based on the inputs it receives and its current potential
4. **Measuring Divergence**: At each time step, measure the difference between the states of the two networks. In the context of spiking neurons, this could involve comparing the spike trains of corresponding neurons in each network. This difference could be quantified using various metrics, such as spike-timing difference, spike count difference, or membrane potential differences.
5. **Tracking the Evolution of the Perturbation**: Observe how the initial small differences evolve over time. These differences might lead to significantly divergent spiking patterns in a network exhibiting chaotic dynamics.

6. **Estimating Lyapunov Exponents:** Calculate the rate at which the trajectories of the original and perturbed networks diverge. This involves fitting an exponential curve to the divergence data over time. The slope of this curve gives an estimate of the Lyapunov exponent. A positive exponent indicates sensitivity to initial conditions and potential chaotic dynamics. The evolution of the map is given by

$$h_i(t_s + \Delta t) = f_i = (1 - \Delta t) h_i(t_s) + \Delta t \sum_{j=1}^{N} J_{ij} \phi(h_j(t_s))$$

In the limit $\Delta t \to 0$, a continuous-time dynamics is recovered. For $\Delta t = 1$, the discrete-time network is obtained. The Jacobian for the discrete-time map is

$$D_{ij}(t_s) = \left. \frac{\partial f_i}{\partial h_j} \right|_{t=t_s} = (1 - \Delta t)\delta_{ij} + \Delta t \cdot J_{ij} \phi'(h_j(t_s)) .$$

The full Lyapunov spectrum is again obtained by a reorthonormalization procedure of the Jacobians along a numerical solution of the map.

A.2 Linearization around Critical Points

Recurrent neural networks (RNNs) are useful tools for learning nonlinear relationships between time-varying inputs and outputs with complex temporal dependencies. Sussillo et al. Sussillo & Barak (2013) have explored the hypothesis that fixed points, both stable and unstable, and the linearized dynamics around them, can reveal crucial aspects of how RNNs implement their computations. Further, they explored the utility of linearization in areas of phase space that are not true fixed points but merely points of very slow movement and presented a simple optimization technique that is applied to trained RNNs to find the fixed and slow points of their dynamics. Linearization around these slow regions can be used to explore, or reverse-engineer, the behavior of the RNN. Similarly, other recent works Sadeh & Rotter (2014) have shown that, for a wide variety of connectivity patterns, a linear theory based on firing rates accurately approximates the outcome of direct numerical simulations of networks of spiking neurons.

Building on these works, we can linearize the HRSNN model around the critical points using the differential equation:

$$\frac{d\boldsymbol{x}}{dt} = -D\boldsymbol{x} + L\boldsymbol{x} + \boldsymbol{b}(t) = A\boldsymbol{x} + \boldsymbol{b}(t) \tag{1}$$

In this equation:

- $\boldsymbol{x}$ represents the vector of firing rates of neurons in the network. - D is a diagonal matrix indicating neurons' intrinsic leak or excitability. - L is the Lyapunov matrix. - $\boldsymbol{b}(t)$ denotes external inputs, including biases. - A is a matrix defined as $A = -D + L$.

The spike frequency of untrained SNNs is used to approximate the firing rates ($\boldsymbol{x}$) in the network. In an untrained SNN, the firing rates can be considered as raw or initial responses to inputs before any learning or adaptation has occurred. This approximation is useful for constructing a linearized model as it provides a baseline from which the effects of learning, pruning, and other dynamics can be analyzed. Each diagonal element D_{ii} of the matrix D quantifies how the firing rate of neuron i changes over time without external input or interaction. These elements can be determined based on the inherent properties of the neurons in the network, such as their leakiness or excitability. The Lyapunov matrix L encapsulates the impact of neighboring nodes on each edge regarding their Lyapunov exponents. The elements of L are calculated using the harmonic mean of the Lyapunov exponents of the neighbors of nodes i and j as detailed in your method. This matrix represents how the dynamics of one neuron affect its neighbors, influenced by the network's overall stability and dynamical behavior. In summary, the linearized model provided by equation 1 is a simplification that helps to understand the fundamental dynamics of the SNN. It uses the initial, untrained spike frequencies to establish a baseline for the network's behavior, and the matrices D and L are calculated based on the intrinsic properties of the neurons and their interactions, respectively.

We aim to create a sparse network (A^{sparse}) with fewer edges while maintaining dynamics similar to the original network. The sparse network is thus represented as:

$$\frac{d\boldsymbol{x}}{dt} = A^{\text{sparse}}\boldsymbol{x} + \boldsymbol{b}(t) \quad \text{such that} \quad \left|\boldsymbol{x}^T \left(A^{\text{sparse}} - A\right)\boldsymbol{x}\right| \le \epsilon \left|\boldsymbol{x}^T A \boldsymbol{x}\right| \quad \forall \boldsymbol{x} \in \mathbb{R}^N \tag{2}$$

for some small $\epsilon > 0$. When the network in Eq. 1 is driven by independent noise at each node, we define $\boldsymbol{b}(t) = \boldsymbol{b} + \sigma\boldsymbol{\xi}(t)$, where $\boldsymbol{b}$ is a constant input vector, $\boldsymbol{\xi}$ is a vector of IID Gaussian white noise, and σ is the noise standard deviation. Let Σ be the covariance matrix of the firing rates in response to this input. The probability p_{ij} for the synapse from neuron j to neuron i with the Lyapunov exponent l_{ij} is defined as:

$$p_{ij} = \begin{cases} \rho l_{ij}(\Sigma ii + \Sigma jj - 2\Sigma ij) & \text{for } w_{ij} > 0 \text{ (excitatory)} \\ \rho |l_{ij}|(\Sigma ii + \Sigma jj + 2\Sigma ij) & \text{for } w_{ij} < 0 \text{ (inhibitory)} \end{cases} \tag{3}$$

Here, ρ determines the density of the pruned network. The pruning process independently preserves each edge with probability p_{ij}, yielding A^{sparse}, where $A_{ij}^{\text{sparse}} = A_{ij}/p_{ij}$, with probability p_{ij} and 0 otherwise. For the diagonal elements, denoted as A_{ii}^{sparse}, representing leak/excitability, we either retain the original diagonal, setting $A_{ii}^{\text{sparse}} = A_{ii}$, or we introduce a perturbation, Δ_i, defined as the difference in total input to neuron i, and adjust the diagonal as $A_{ii}^{\text{sparse}} = A_{ii} - \Delta_i$. Specifically, $\Delta_i = \sum_{j \neq i} |A_{ij}^{\text{sparse}}| - \sum_{j \neq i} |A_{ij}|$. This perturbation, Δ_i, is typically minimal with a zero mean and is interpreted biologically as a modification in the excitability of neuron i due to alterations in total input, aligning with the known homeostatic regulation of excitability.

A.3 Baseline Pruning Methods

Activity pruning is a technique employed to optimize neural network models by iteratively removing the least active neurons. In this approach, outlined as the Iterative Activity Pruning algorithm, the process starts with an initial Recurrent Spiking Neural Network (RSNN) model M. The algorithm operates by first evaluating each neuron's activity level, followed by pruning a certain percentage (determined by the pruning rate r) of neurons that exhibit the lowest activity. Post pruning, the model M is retrained to compensate for the loss of neurons, forming an updated model M'. This cycle of pruning and retraining continues until either a maximum number of iterations T is reached, or the performance of the pruned model drops below 10% of the original, unpruned model's performance. The goal of this method is to refine the model by removing less critical neurons while maintaining or enhancing overall performance. This technique is validated by comparing its efficacy in classifying datasets like CIFAR10 & CIFAR100 against other state-of-the-art pruning algorithms. The detailed algorithm is given in Algorithm 3. We also evaluate our model for classifying the CIFAR10 & CIFAR100 datasets and compare the results with current task-dependent state-of-the-art pruning algorithms. The results are shown in Table **??**.

A.4 Datasets

A.4.1 Lorenz system

The Lorenz system is a non-linear, three-dimensional system that can be described as follows:

$$\begin{aligned} dx/dt &= \sigma(y - x) \\ dy/dt &= x(\rho - z) - y \\ dz/dt &= xy - \beta z \end{aligned}$$

when $\sigma = 10, \beta = 8/3$, and $\rho = 28$, the system has chaotic solutions. The experimental setup, same as Xu et al. (2018a), was used in this paper, and the fourth-order Runge-Kutta method was used to generate samples. Table 3 summarizes the details of the experimental setup.

Table 1: Details of the experimental setup for the Lorenz system.

Parameter	**Value**
Number of samples	20000
Initial state	$[12, 2, 9]$
Step size	0.01
Number of training samples	11250
Number of validation samples	3750
Number of test samples	5000

Table 2: Details of the experimental setup for the Rossler system.

Parameter	**Value**
Number of samples	12700
Initial state	$[1, 1, 1]$
Step size	0.03
Number of discarded samples	7700
Number of training samples	3000
Number of validation samples	1000
Number of test samples	1000

A.4.2 Rossler system

The Rossler system is a classical system, consisted of three nonlinear ordinary differential equations and can be defined by:

$$dx/dt = -y - z$$
$$dy/dt = x + ay$$
$$dz/dt = b + z(x - c)$$

when $a = 0.15, b = 0.2$, and $c = 10$, the system shows chaotic behavior. To compare the performance of MFRFNN with other methods under the same condition, the experimental setup, same as Xu et al. (2018b), was used for the Rossler system. In this setup, the fourth-order Runge-Kutta method was employed for sample generation. Some of the samples were discarded to eliminate the transient influence of the initial condition. Table 5 presents the details of the experimental setup for the Rossler system.

A.4.3 Google Stock price prediction problem

Stock price prediction is a non-linear and highly volatile problem. In this problem, the future value of Google stock price is predicted using the current price as defined by.

$$\hat{y}(t) = f(y(t - 1))$$

The dataset was obtained from Yahoo Finance during a six-year period from 19-August-2004 to 21-September-2010 as in Samanta et al. (2020). The training set consisted of 1529 samples, and the test set 900 samples. To evaluate the performance of MFRFNN on another real-world time series, we compared its performance with the same RFNNs and FNNs used in Box-Jenkins and wind speed prediction datasets.

A.4.4 Wind Prediction

The wind speed prediction problem is a non-linear, dynamic, and volatile problem in which the future value of wind speed is predicted using the current wind speed and wind direction. The dataset is obtained from the Iowa Department of Transport's web- site.1 The data was collected from the Washington station during a one-month period (February 2011), sampled every ten minutes, and averaged hourly. There are 500 samples in the training set and 1000 samples in the test set Samanta et al. (2020). This dataset is more challenging than the Box–Jenkins dataset due to the existence of

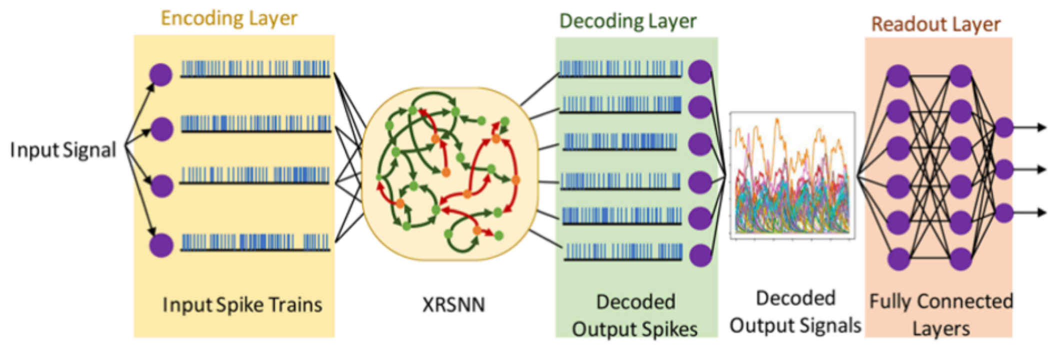

Figure 1: Block Diagram showing the methodology using HRSNN for prediction

noise. This experiment compared the proposed method's performance with the same algorithms we used in the Box–Jenkins dataset.

A.5 Calculation of SOPs

In evaluating the energy efficiency of neuromorphic chips, a key metric is the average energy consumption for transmitting a single spike across a synapse, as highlighted by some recent works Furber (2016),Anonymous (2023). This metric is particularly important due to the substantial energy expenditure involved in synapse processing, which significantly impacts the overall energy usage. For a theoretical analysis that is independent of specific hardware, we consider the average energy for a single spike-synapse transmission as a fixed constant. The total energy consumption of a Spiking Neural Network (SNN) model can be approximated by tallying the synaptic operations (SOPs) required, analogous to counting floating-point operations (FLOPs) in Artificial Neural Networks (ANNs). Our model for calculating the SNN's energy consumption is expressed as:

$$E = C_E \cdot \text{SOP} = C_E \sum_i s_i c_i \tag{4}$$

Here, C_E represents the energy usage per SOP, and SOP = $\sum_i s_i c_i$ is the cumulative count of synaptic operations. In any given presynaptic neuron i, s_i signifies the spike count emitted by that neuron, while c_i indicates its synaptic connections. Each spike transmission from this neuron triggers a synaptic operation as it reaches the postsynaptic neurons, contributing to the energy expenditure.

In the context of sparse SNNs, the energy consumption model is modified as follows:

$$E = C_E \sum_i \left(s_i \sum_j n_i^{\text{pre}} \wedge \theta_{ij} \wedge n_{ij}^{\text{post}} \right) \tag{5}$$

In this equation, for every synaptic link from the i-th presynaptic neuron to its j-th postsynaptic neuron, the tuple $\left(n_i^{\text{pre}}, \theta_{ij}, n_{ij}^{\text{post}}\right)$, each element of which can be either 0 or 1, represents the states of the presynaptic neuron, the synapse, and the postsynaptic neuron, respectively, here, 1 indicates an active state, and 0 denotes a pruned state, the symbol $\wedge$ stands for the logical AND operation.

A.6 HRSNN Model

HRSNN Model Architecture: Fig. 1 shows the overall architecture of the prediction model. Using a rate-encoding methodology, the time-series data is encoded to a series of spike trains. This high-dimensional spike train acts as the input to HRSNN. The output spike trains from HRSNN act as the input to a decoder and a readout layer that finally gives the prediction results. For the classification task, we use a similar method. However, we do not use the decoding layer for the signal but directly feed the output spike signals from HRSNN into the fully connected layer.

Readout Layer: The read-out layer is task-dependent and uses supervised training. In this paper, we do not explicitly prune the read-out network, but the readout layer is implicitly pruned. The readout layer is a multi-layer (two or three layers) fully connected network. The first layer size is equal to the

Table 3: Table showing the relative performance change when including the readout layer in the calculations

	Neuron Sparsity Change	Synapse Sparsity Change	Avg SOP change CIFAR10	Avg SOP change CIFAR100
HRSNN	-0.13	-0.18	-0.09	-0.14
CHRSNN	-0.08	-0.13	-0.1	-0.16

number of neurons sampled from the recurrent layer. We sample the top 10% of neurons with the greatest betweenness centrality. Thus, as the number of neurons in the recurrent layer decreases, the size of the first layer of the read-out network also decreases. The second layer consists of fixed size with 20 neurons, while the third layer differs between the classification and the prediction tasks such that for classification, the number of neurons in the third layer is equal to the number of classes. On the other hand, the third layer for the prediction task is a single neuron which gives the prediction output. The table 3 shows the relative change in sparsity when including/excluding the readout layer for calculations:

A.7 Dynamic Characterization using Lyapunov Spectra

To show the principal dynamic characteristic of the LNP-HRSNN model, we plot the full Lyapunov spectrum of the HRSNN model for three different cases - the unpruned network, the pruned network, and the trained pruned network. We refer to the methodologies discussed in recent works Vogt et al. (2020); Engelken et al. (2023). The Lyapunov Spectrum provides valuable additional insights into the collective dynamics of firing-rate networks. We plot the evolution of Lyapunov spectra with different initialization parameters. We plotted the Lyapunov spectrum, with three different probabilities of synaptic connection for the initial network (p=0.001, p=0.01, p=0.1). We plot the Lyapunov exponents (λ_i) vs the normalized indices i/N described as follows:

- $\lambda_\mathbf{i}$: This axis represents the Lyapunov exponents. A positive exponent indicates chaos, meaning that two nearby trajectories in the phase space will diverge exponentially. A negative exponent suggests that trajectories converge, and zero would imply neutral stability.
- **i/N**: This axis is likely indexing the normalized Lyapunov exponents, with i being the index of a particular exponent and N being the total number of exponents calculated. For a system with N dimensions, there are N Lyapunov exponents.

The graph shows three plots of the Lyapunov spectrum for different stages of pruning and training:

1. **Unpruned**: This plot represents the Lyapunov spectrum of the neural network before any pruning has been done. The spectrum shows a range of Lyapunov exponents from positive to negative values, indicating that the network likely has both stable and chaotic behaviors. More notably, the more sparse initialized models (p = 0.1) were more unstable as the largest Lyapunov exponent is positive.
2. **Pruned Pre-Training**: This plot shows the Lyapunov spectrum after the network has been pruned but before it has been trained again. Pruning is a process in which less important connections (weights) in a neural network are removed, which can simplify the network and potentially lead to more efficient operation without significantly impacting performance.
3. **Pruned Post-Training**: This plot illustrates the Lyapunov spectrum after the network has been pruned and then trained again. Retraining the network after pruning allows it to adjust the remaining weights to compensate for the removed connections, which can restore or even improve performance.

We can infer the following from the plots: **Impact of Pruning:** Comparing the 'Unpruned' to the 'Pruned Pre-Training' plot, it seems that pruning increases the stability of the system as the Lyapunov exponents become more negative (less chaotic). This might suggest that pruning reduces the complexity of the dynamics.

Training Effect: The 'Pruned Post-Training' plot shows that after re-training, the exponents are less negative compared to 'Pruned Pre-Training', which may indicate that the network is able to regain some dynamical complexity or expressive power through training.

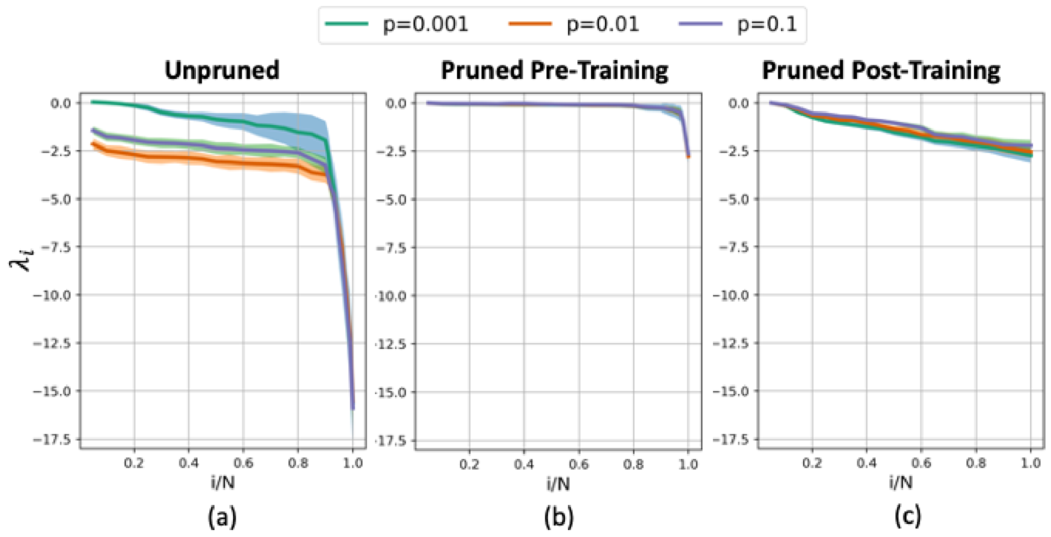

Figure 2: Lyapunov Spectrum of the HRSNN model in three stages of pruning and training and different levels of sparsity for initialization: (a) the Lyapunov Spectrum of unpruned HRSNN model with a probability of connection $p = 0.1, 0.01, 0.001$ (b) the Lyapunov Spectrum after LNP pruning (c) the Lyapunov Spectrum after training of the pruned model

Initialization Synapse Density: Different initialization synapse densities (arising from different p) have different effects on the spectrum. High p) tends to result in more negative exponents, suggesting greater stability but potentially less capacity to model complex patterns.

Network Robustness: The fact that the Lyapunov spectrum does not dramatically change in shape across the three stages, especially in the 'Pruned Post-Training' stage, might imply that the network is robust to pruning, retaining its general dynamical behavior while likely improving its efficiency.

Convergence of Spectra: The convergence of the spectra for different values of p after pruning, both pre-and post-training, suggests that regardless of the initial density of connections, the network may evolve towards a structure that is similarly efficient for the task it is being trained on. This indicates that the LNP algorithm can efficiently prune a network irrespective of the initial density and also make the final model more stable.

Overall, the Lyapunov Spectrum serves as useful tool for understanding how network pruning and re-training affect the dynamics of neural networks, which is important for optimizing neural networks for better performance and efficiency.

Comparison with Randomly Sparse Initialization: We can get an idea of the principal dynamic characteristic of LNP-HRSNN from the Lyapunov spectrum of the randomly initialized network with a very low probability of connection (p=0.001). We see that such networks were more unstable as the largest Lyapunov exponent was positive. This unstable behavior might be because randomly generated networks lack structured connections that might otherwise guide or constrain the flow of neural activity. Without these structures, the network is more likely to exhibit erratic behavior as the activity patterns are less predictable and more susceptible to amplification of small perturbations, making them more unstable.

B Supplementary Section B

B.1 Additional Results

Performance Evaluation

For a more extensive performance evaluation of the LNP pruning method, we test the model on 4 different datasets and note the RMSE loss and the VPT. The results are shown in Table 4. From the table, we can see that models undergoing LNP Pruning generally outperform their counterparts across

Table 4: Performance of HRSNN and CHRSNN models using different pruning methods. Each model is trained on the first 200 timesteps and predicts the next 100 timesteps.

Pruning Method	Model	SOPs	Synthetic				Real-world			
			Lorenz63		Rossler		Google Stock		Wind	
			RMSE	VPT	RMSE	VPT	RMSE	VPT	RMSE	VPT
Unpruned	***HRSNN***	710.76 ± 79.65	0.315 ± 0.042	35.75 ± 4.65	0.142 ± 0.019	41.32 ± 5.32	0.905 ± 0.095	32.36 ± 3.14	1.098 ± 0.033	28.36 ± 3.35
	CHRSNN	744.97 ± 80.09	0.285 ± 0.021	40.17 ± 5.13	0.125 ± 0.017	44.17 ± 4.95	1.098 ± 0.091	36.58 ± 3.89	1.181 ± 0.29	27.95 ± 3.02
AP Pruned	***HRSNN***	92.68 ± 10.11	1.718 ± 0.195	21.10 ± 7.22	0.989 ± 0.127	29.55 ± 6.95	1.948 ± 0.179	19.25 ± 3.54	2.146 ± 0.081	14.25 ± 3.74
	CHRSNN	118.77 ± 10.59	1.596 ± 0.194	29.41 ± 7.33	0.879 ± 0.131	37.25 ± 6.42	1.987 ± 0.191	17.68 ± 2.59	2.228 ± 0.075	16.98 ± 3.22
LNP Pruned	***HRSNN***	18.22 ± 2.03	0.705 ± 0.104	32.17 ± 4.62	0.368 ± 0.051	40.15 ± 5.12	0.917 ± 0.124	30.25 ± 3.26	0.314 ± 0.049	27.98 ± 1.28
	CHRSNN	14.79 ± 1.58	0.679 ± 0.098	39.24 ± 4.15	0.314 ± 0.042	47.36 ± 4.89	0.901 ± 0.101	32.14 ± 3.05	0.301 ± 0.035	28.06 ± 2.25

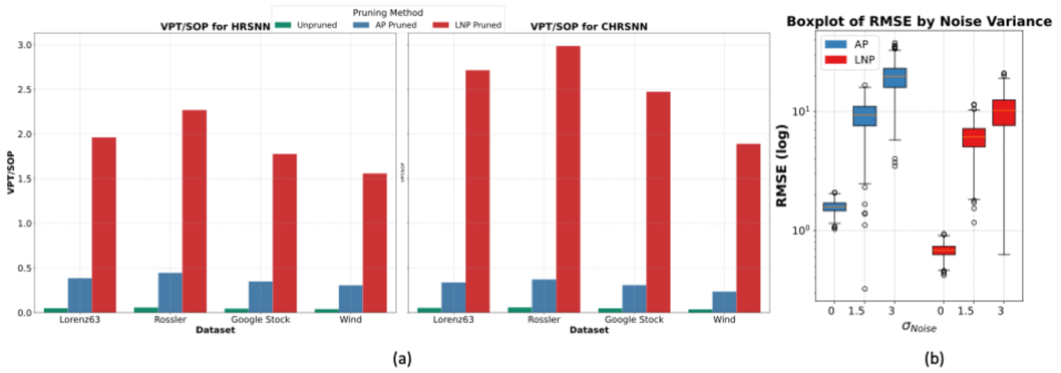

Figure 3: (a) Bar graph showing the comparative analysis of how efficiency (VPT/SOPs) changes for CHRSNN and HRSNN models for the different pruning methods (b) Box plot showing the change in performance of the three trained models obtained by the three pruning methods - AP, and LNP

a variety of datasets and metrics. For instance, focusing on the Real-world dataset 'Wind' under the RMSE metric, the LNP Pruned models, both HRSNN and CHRSNN, exhibit the lowest error rates with 0.314 ± 0.049 and 0.301 ± 0.035, respectively. The underlined values across LNP Pruned models indicate a consistently superior performance, representing the lowest RMSE and VPT values among the pruned and unpruned models across all datasets. Additionally, despite having a smaller FLOPS value, indicative of a lighter model, LNP pruned models, particularly CHRSNN with only 0.018 FLOPS, achieve superior or comparable performance against the unpruned models, demonstrating the efficacy of LNP in maintaining model accuracy while reducing computational complexity. The marked efficiency and performance enhancement accentuate the viability of LNP as a potent pruning strategy, rendering it optimal for environments where computational resources are a constraint.

Comparing Efficiency We analyzed the performance of different pruning methods: Unpruned, Activity Pruned, and LNP Pruned, on four datasets: Lorenz63, Rossler, Google Stock, and Wind, using HRSNN and CHRSNN models, as shown in Suppl. Fig 3. The bar graphs illustrate that the LNP Pruned method performs the best in terms of efficiency (defined as the ratio of VPT to FLOPS) across all datasets and models, highlighting its effectiveness in improving computational efficiency and resource use. The LNP Pruned method is particularly distinguished in CHRSNN models, where it achieves high VPT/FLOPS values in the Rossler and Lorenz63 datasets. The efficacy of this method stems from its capability to retain critical network parameters while discarding the redundant ones, thereby ensuring optimized model performance without compromising accuracy. Such optimization is imperative in practical scenarios where computational resources are constrained, necessitating the development of efficient and effective models.

Comparison of Performance with Noise

Next, we test the stability of the models with varying levels of input noise. The results are plotted in Fig. 4 offers an insightful depiction of the Prediction RMSE, represented on a logarithmic scale, across varying input Signal-to-Noise Ratios (SNRs) for both HRSNN and CHRSNN models, subjected to different pruning methods: Unpruned, AP Pruned, and LNP Pruned. The key observation from the figure is the universal increase in RMSE with the decrease in SNR, reflecting the inherent

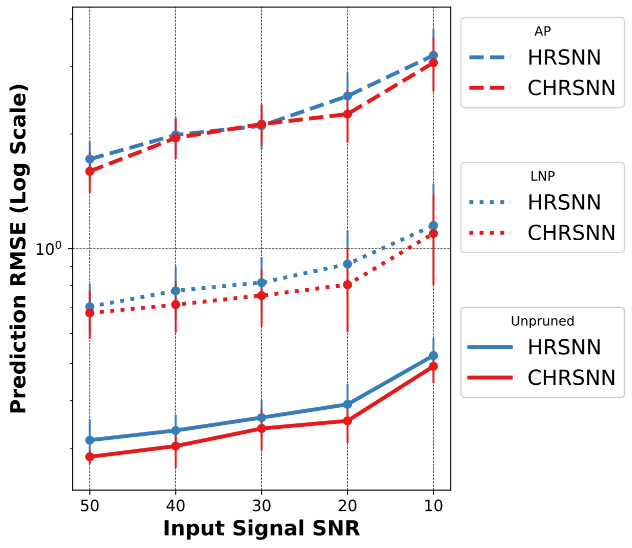

Figure 4: Performance Analysis of Pruned and Unpruned Models with Varied Input SNR Levels

Table 5: Performance of the pruned and unpruned models with different input noise levels.

		SNR (dB)				
		50	40	30	20	10
Unpruned	HRSNN	0.315±0.042	0.334±0.033	0.361±0.041	0.391±0.052	0.425±0.059
	CHRSNN	0.285±0.011	0.304±0.038	0.338±0.042	0.354±0.044	0.362±0.048
AP	HRSNN	1.718±0.195	1.988±0.229	2.101±0.284	2.514±0.386	3.114±0.551
	CHRSNN	1.596±0.194	1.952±0.233	2.122±0.264	2.254±0.357	2.974±0.492
LNP	HRSNN	0.705±0.104	0.776±0.123	0.815±0.136	0.952±0.205	1.421±0.326
	CHRSNN	0.679±0.098	0.714±0.111	0.754±0.129	0.825±0.201	1.397±0.294

challenges associated with noise in input signals. Among the evaluated methods, AP Pruned models exhibit the highest RMSE across all SNR levels, indicating suboptimal performance under noise. In contrast, Unpruned models maintain the lowest RMSE, particularly at higher SNRs, showcasing their robustness to noise. The LNP Pruned method achieves a balanced performance, demonstrating its capability to maintain considerable accuracy under noisy conditions while optimizing computational efficiency, hence affirming its practical applicability where a balance between accuracy and efficiency is essential. The visualization succinctly encapsulates the comparative performance dynamics, providing valuable insights into the interdependencies between noise levels, pruning methods, and model types.

Final Timescales of the Pruned Network For this paper, all the experiments are initiated with a 5000 neuron network. We have two different starting architectures - the HRSNN model or the CHRSNN model. As discussed in the text, the LNP Pruning algorithm, not only prunes the network but also optimizes the timescales. The final resultant timescale of this pruning method for a given initialization of the model is shown in Fig. 5.

B.2 Using the LNP Algorithm on Feedforward Networks

Modified Algorithm: In this section, we extend the Lyapunov Noise Pruning Method for feedforward Networks. We note that this is not a trivial task, as the original method was engineered to exploit the recurrence present in the neuronal dynamics. Hence, for the case of feedforward neural networks, we solely focus on ResNet-based architectures, i.e., models with added skip connections. Thus, the four steps for the LNP-based pruning algorithm are modified as follows:

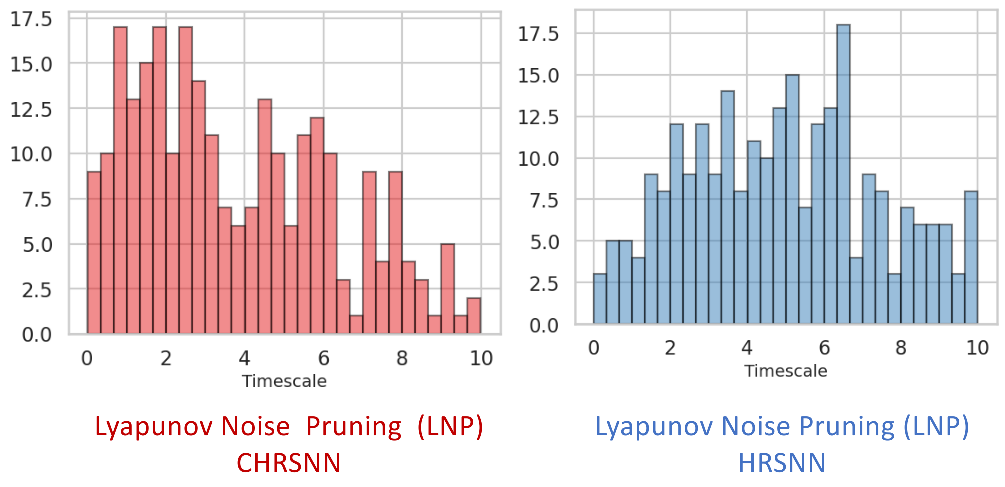

Figure 5: Figure showing the final output timescales of the LNP pruning algorithm on the HRSNN and CHRSNN networks

- **Step 1: Unstructured Pruning of Synapses:** The process of synapse pruning is the same, where the synapses are pruned using the noise-based pruning algorithm described earlier.
- **Step 2: Structured Pruning of Channels:** For feedforward SNNs like ResNets, we use channels instead of neurons for structured pruning. Let us denote the k-th channel of the l-th layer in the network as x_k^l. In each iteration, the average membrane potential of this channel on the training set is computed as

$$x_k^l = \frac{1}{T}(\sum_{t=1}^{T} \|V_k^l(t)\|)$$

Here, T is the timestep of the SNN, and $|V_k^l(t)|$ represents the L1 norm of the membrane potential of the channel's feature map at time t. The feature map's membrane potential for a specific neuron at time t is denoted as V_t^{ij}, with the channel's membrane potential being $|V_k^l(t)| = [V_t^{ij}]_{h\times w}$. A positive V_t^{ij} indicates an excitatory postsynaptic potential (EPSP), whereas a negative value signifies an inhibitory postsynaptic potential (IPSP).

To optimize the network, we prune convolutional kernels based on the calculated channel importance scores. Channels with lower scores are deemed redundant and are more likely to be pruned. This method ensures a balanced reduction of redundant components across the network, enhancing compression and maintaining performance. We use a mask to track the structure of the network after pruning, where a value of 0 indicates a pruned channel and 1 signifies a retained channel.

- **Step 3: Norm Preservation using Skip Connections:** Norm preservation in ResNets is the property that the norm of the gradient of the network with respect to the input is close to the norm of the gradient with respect to the output. This property is desirable because it means that the network can avoid the vanishing or exploding gradient problem, which can hamper the optimization process. Norm preservation is facilitated by the skip connections in the residual blocks, which allow the gradient to flow directly from the output to the input. Norm preservation is also enhanced as the network becomes deeper because the residual blocks act as identity mappings that preserve the norm of the gradient. After the unstructured and structured pruning in Steps 1 and 2, we reintroduce some additional skip connections for norm preservation.
- **Step 4: Neuronal Timescale Optimization:** Similar to the previous case, we optimize the hyperparameters of the spiking ResNet using the Lyapunov spectrum-based Bayesian Optimization technique.

Table 6: Table showing the performance of LNP on Feedforward Neural Networks

Model	CIFAR10				CIFAR100			
	Baseline Accuracy	**Acuracy**	**Neuron Sparsity**	**Synapse Sparsity**	**Baseline Accuracy**	**Acuracy**	**Neuron Sparsity**	**Synapse Sparsity**
ResNet19 (untrained)	92.11± 0.9	-2.15± 0.19	90.48	94.32	73.32± 0.81	-3.56± 0.39	90.48	94.32
ResNet19* (converted)	93.29± 0.74	-0.04± 0.01	93.67	98.19	74.66± 0.65	-0.11± 0.02	94.44	98.07

Experiments and Evaluations: Now, since the model pruning heavily relies on the skip connections for the norm preservation step, hence, for evaluating the performance of the LNP on the feedforward Neural networks, we look into the ResNet-based model. We evaluate the model for CIFAR10 and CIFAR100 datasets with two different initialization: For the first case, we use the untrained ResNet19 network, then prune it and finally train it and get the model performance. For the second case, we use the pre-trained SNN, which is converted from an ANN using ANN-to-SNN conversion techniques, and then use the LNP pruning on that ResNet model. The results are shown in Table 6. We see that LNP gives extremely good results when used on the SNN, which is converted from the ANN model. Hence opening up another possible application of the LNP pruning method.

B.3 Generalization Properties

As discussed earlier, our proposed LNP pruning algorithm is task-agnostic and does not use a dataset to train while pruning. This pre-training pruning algorithm makes the model extremely generalizable as opposed to current state-of-the-art pruning methods, which require iterative pruning and retraining. This iterative process makes these pruned models extremely overfitted to the dataset it is trained on, and would thus require retraining and repruning of the model for each dataset.

Our LNP algorithm starts with an untrained dense network, and the pruning process does not consider task performance while removing network connections (and neurons) and neuronal time scales. Consequently, the end pruned network does not overfit to any particular dataset; rather generalizable to many different datasets, and even different tasks. On the other hand, prior SNN pruning papers (such as the one referred by the reviewer from Panda et al.), start with a pre-trained DNN model on a given task which is converted to an SNN. During pruning, the network is simultaneously pruned and re-trained with the dataset (at each pruning step) to minimize performance loss on that dataset. This makes the model overfitted (and hence, shows good results) to that dataset. We use the following terminology for the different pruning cases:

- *Untrained SNN:* The SNN model is not trained before pruning, and the parameters are randomly initialized. Only the final pruned network is then trained
- *Converted SNN*: A standard DNN was trained on a dataset and then converted to an SNN with the same architecture using DNN-to-SNN conversion methods.

To verify our conjecture, we empirically study the following questions:

1. **How does our pruning method perform on a converted SNN model like prior works?**
    - **Our Approach 1 (ResNet converted):** We consider two cases. First, we use our pruning method on a dense Spiking ResNet, which is converted from a DNN ResNet (trained on the CIFAR10 and CIFAR100 datasets). We continue with our approach where the pruning iterations do not consider task performance. We observe that when starting from a pre-trained dense model like prior works, the performance and sparsity of the network pruned with LNP are better than prior pruning methods.
    - **Our Approach 2 (ResNet untrained):** We start with an untrained ResNet with randomly initialized weights and parameters. They use our LNP algorithm without training the model at any iteration of the pruning process. The final pruned network with optimized hyperparameters (optimized using the Lyapunov spectrum) is then trained and tested on the CIFAR10 and CIFAR100 datasets

Table 7: Table showing the generalization performance of the LNP algorithm compared to other baseline pruning methods

Pruning Method	Architecture	Neuron Sparsity	Synapse Sparsity	CIFAR10 Accuracy	CIFAR100 Accuracy
IMP (Panda et al.)	*ResNet19 (iterative pruning and retraining)*	-	97.54	93.18	68.95
LNP (Ours)	*ResNet19 (untrained)*	90.48	94.32	89.96 ± 0.71	69.76 ± 0.42
LNP (Ours)	*ResNet (pre-trained)*	93.67	98.19	93.25 ± 0.73	74.55 ± 0.63

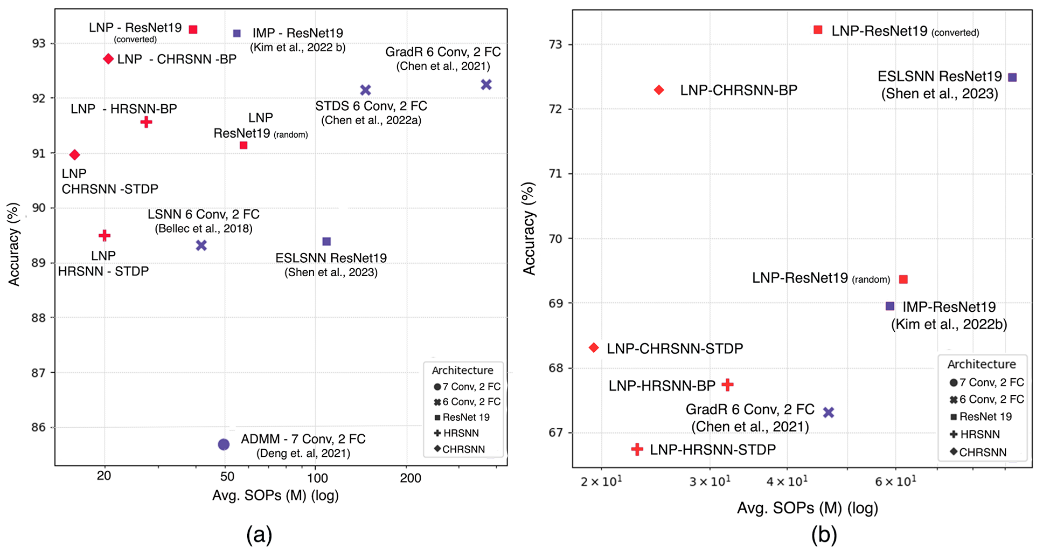

Figure 6: Scatter Plot showing the Performance vs SOPs for (a) CIFAR10 and (b) CIFAR100

- **Approach by IMP (Panda et al.):** Iterative Magnitude Pruning (IMP) is a technique for neural network compression that involves alternating between pruning and retraining steps. In this process, a small percentage of the least significant weights are pruned, and the network is then fine-tuned to recover performance. This iterative cycle continues until a desired level of sparsity is reached, potentially enhancing network efficiency with minimal impact on accuracy.

2. **How do other task-dependent pruning methods perform when the network pruned for one task is used for a different task?**

   - To compare this generalizability property, we took the pruned model from Panda et al. paper which is optimized for CIFAR10. We next train their pruned model on CIFAR100 sparsity, accuracy and SOPs. We compare the performance with the performance of our pruned untrained ResNet model (with random weights) and pruned converted ResNet (as described earlier). We see that the untrained ResNet generated by our LNP-based pruning can generalize better to the CIFAR100 dataset and show better performance. Moreover, we see that using the converted ResNet, we see better performance and efficiency of the pruned ResNet (converted) than the IMP-based pruning model on both CIFAR10 and CIFAR100 datasets.

The results are shown in the Table 7.

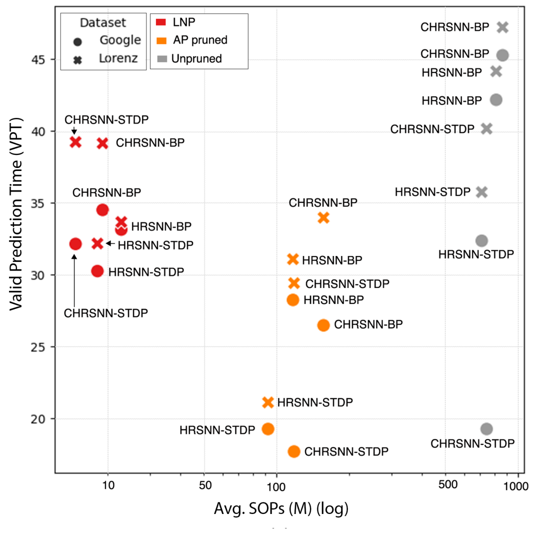

Figure 7: Scatter Plot showing the Performance vs SOPs for Lorenz63 dataset

B.4 Scatter Plots

In order to get a better understanding of the performance vs efficiency of the final pruned models of the LNP method compared to the state-of-the-art pruning models. We plot the accuracy vs average SOPs for the CIFAR10, and CIFAR100 classification tasks and the VPT vs. average. SOPs for the Lorenz63 prediction task. The results are shown in Fig. 6 and 7. We observe that the LNP-based methods always show better performance with lower SOPs compared to other state-of-the-art pruning methods.

B.5 Ablation Study

We conducted an ablation study, where we systematically examined various combinations of the four sequential steps involved in the LNP method. This study's findings are presented graphically, as illustrated in Fig. 8. At each point (A-E), we train the model and obtain the model's accuracy and average. Synaptic Operations (SOPs) to support ablation studies. The ablation study is done for the HRSNN model, which is trained using STDP and evaluated on the CIFAR10 dataset.

In the figure, different line styles and colors represent distinct aspects of the procedure: the blue line corresponds to Steps 1 and 2 of the LNP process, the orange line to Step 3, and the green line to Step 4. Solid lines depict the progression of the original LNP process $(A \rightarrow B \rightarrow C \rightarrow D \rightarrow E)$, while dotted lines represent the ablation studies conducted at stages B and C. This visual representation enables a clear understanding of the individual impact each step exerts on the model's performance and efficiency. Additionally, it provides insights into potential alternative outcomes that might have arisen from employing different permutations of these steps or omitting certain steps altogether.

Below is a detailed discussion of the figure:

1. We start with Point A, which represents the randomly initialized unpruned (dense) HRSNN network
   - The blue line (A → B) indicates Step 1 of the LNP algorithm, where we prune the synapses using the noise-based pruning method.
2. Point B is the outcome of synapse pruning the randomly initialized network. There are 3 different directions we can go from here:
   - Step 2 (The standard LNP step): The blue line B → C indicates Step 2 of the LNP method, where we prune the neurons based on the betweenness centrality. This is the standard step used in the LNP algorithm.

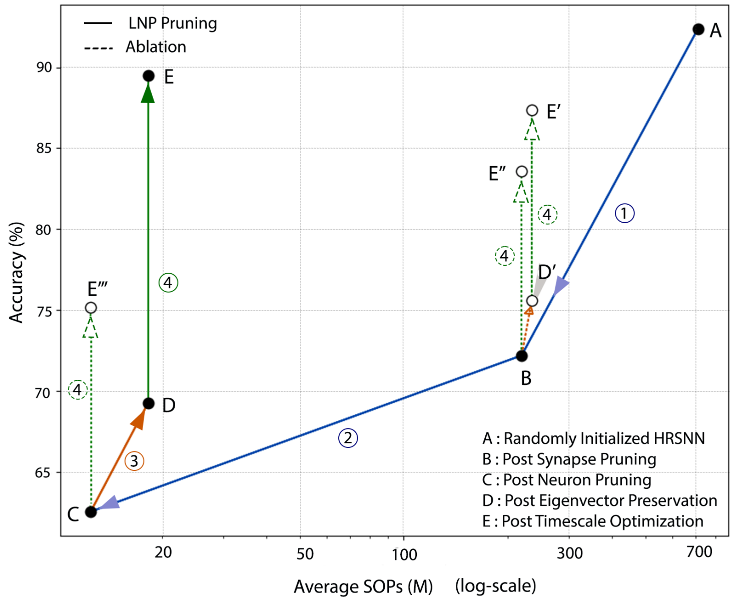

Figure 8: Plot showing Ablation studies of LNP

  - Step 3: The orange line B → D' indicates Step 3 (eigenvector preservation) done directly after Step 1 (skipping Step 2 or the neuron pruning step).
    - This is followed by Step 4 (Timescale Optimization) to reach the ablation model E'. We see the performance of E' is quite high but the Avg. SOPs is much higher than E (LNP solution).
  - Step 4: We can directly optimize the neuron timescales from B (skipping steps 2,3), reaching E". However, the performance is even worse.

3. Point C is derived by using Step 2 (neuron pruning) on Point B. This reduces the SOPs significantly, but also the performance deteriorates. We can go two ways from this point:
    - Step 3 (The standard LNP step): We can do eigenvector preservation to reach point D.
    - Step 4 (Timescale Optimization) directly (skipping Step 3)- This leads to a model with very sparse connections, but performance also takes a big hit.
4. Point D derived by using Step 3 (Eigenvector Preservation) on Point C. The model performance increases a bit, but the SOPs also increase due to the addition of new synapses in this step.
5. Point E: This is the final output of the LNP algorithm where we use Step 4 (Timescale Optimization) at point D. We see this process gives the model with the best performance and SOP.

C SUPPLEMENTARY SECTION C

C.1 RELATED WORKS

Pruning Methods for DNN Pruning methods in Deep Neural Networks (DNNs) are crucial for optimizing models, especially for deployment in resource-constrained environments. One common technique is magnitude-based pruning, where components of the network are selectively pruned based on their magnitudes, often without the need for retraining the model post-pruning Hoefler et al. (2021); Vadera & Ameen (2020). Another prevalent method is filter pruning, which focuses on pruning filters in convolutional layers to reduce the computational cost while maintaining model performance He et al. (2022); Kulkarni et al. (2022). Structured pruning is also employed, where pruning is performed in a structured manner, and the pruning rate for each layer is adaptively derived based on gradient and loss function Sakai et al. (2022). Sensitivity analysis is used to assess the effect of pruning, allowing for the identification of redundant components that can be removed with minimal impact on model accuracy Vadera & Ameen (2020). Multi-objective sensitivity pruning considers hardware-based objectives such as latency and energy consumption along with model accuracy to optimize the pruning process Sabih et al. (2022). Deep Q-learning-based methods like QLP intelligently determine favorable layer-wise sparsity ratios, implementing them via unstructured, magnitude-based, weight pruning Camci et al. (2022).

Pruning Methods for SNN

Pruning methods in Spiking Neural Networks (SNNs) are essential for optimizing models for deployment in resource-constrained environments. Gradient Rewiring is a method inspired by synaptogenesis and synapse elimination in the neural system, allowing for the optimization of network structure without retraining, maintaining minimal loss of SNNs' performance on various datasets. Spatio-temporal pruning is another method that utilizes principal component analysis to perform structured spatial pruning, leading to significant model compression and latency reduction Chowdhury et al. (2021). DynSNN proposes a dynamic pruning framework that optimizes network topology on the fly, achieving almost lossless performance for SNNs on multiple datasets Liu et al. (2022a). u-Ticket addresses the workload imbalance problem in sparse SNNs by adjusting the weight connections during Lottery Ticket Hypothesis-based pruning, ensuring optimal hardware utilization Yin et al. (2023). Developmental Plasticity-inspired Adaptive Pruning (DPAP) draws inspiration from the brain's developmental plasticity mechanisms to dynamically optimize network structure during learning without pre-training and retraining, achieving efficient network architectures Han et al. (2022). Multi-strength SNNs employ an innovative deep structure that allows for aggressive pruning strategies, reducing computational operations significantly while maintaining accuracy.

Spiking Neural Networks

Spiking neural networks (SNNs) Ponulak & Kasinski (2011) use bio-inspired neurons and synaptic connections, trainable with either unsupervised localized learning rules such as spike-timing-dependent plasticity (STDP) Gerstner & Kistler (2002); Chakraborty et al. (2023) or supervised backpropagation-based learning algorithms such as surrogate gradient Neftci et al. (2019). SNNs are gaining popularity as a powerful low-power alternative to deep neural networks for various machine learning tasks. SNNs process information using binary spike representation, eliminating the need for multiplication operations during inference. Recent advances in neuromorphic hardware Akopyan et al. (2015), Davies et al. (2018), Kim et al. (2022) have shown that SNNs can save energy by orders of magnitude compared to artificial neural networks (ANNs), maintaining similar performance levels. Since these networks are increasingly crucial as efficient learning methods, understanding and comparing the representations learned by different supervised and unsupervised learning models become essential. Empirical results on standard SNNs also show good performance for various tasks, including spatiotemporal data classification, Lee et al. (2017); Khoei et al. (2020), sequence-to-sequence mapping Chakraborty & Mukhopadhyay (2023a),Zhang & Li (2020), object detection Chakraborty et al. (2021); Kim et al. (2020), and universal function approximation Gelenbe et al. (1999); Iannella & Back (2001). Furthermore, recent research has demonstrated that introducing heterogeneity in the neuronal dynamics Perez-Nieves et al. (2021); Chakraborty & Mukhopadhyay (2023b; 2022); She et al. (2021) can enhance the model's performance to levels akin to supervised learning models.

STDP-based learning in Recurrent Spiking Neural Network Spike-Timing-Dependent Plasticity (STDP) Gerstner et al. (1993),Chakraborty & Mukhopadhyay (2021) based learning in recurrent Spiking Neural Networks (SNNs) is a biologically inspired learning mechanism that relies on the precise timing of spikes for synaptic weight adjustment. STDP enables the network to learn and generate sequences and abstract hidden states from sensory inputs, making it crucial for tasks like pattern recognition and sequence generation in recurrent SNNs. For instance, a study by Guo et al. Guo et al. (2021) proposed a supervised learning algorithm for recurrent SNNs based on BP-STDP, focusing on optimizing learning in a structured manner. Another research by van der Veen van der Veen (2022) explored the incorporation of STDP-like behavior in eligibility propagation within multi-layer recurrent SNNs, demonstrating improved classification performance in certain neuron models. Chakraborty et al. Chakraborty & Mukhopadhyay (2022) presented a heterogeneous recurrent SNN for spatio-temporal classification, utilizing heterogeneous STDP with varying learning dynamics for each synapse, showing comparable performance to backpropagation trained supervised SNNs with less computation and training data. Panda et al. Panda & Roy (2017) combined Hebbian plasticity with a non-Hebbian synaptic decay mechanism in a recurrent spiking model to learn stable contextual dependencies between temporal sequences, suppressing chaotic activity and enhancing the model's ability to generate sequences consistently.

Spectral Graph Sparsification Algorithm

Spectral graph sparsification algorithms aim to reduce the complexity of graphs while preserving their spectral properties, particularly the eigenvalues of the Laplacian matrix. One such algorithm is the unweighted spectral graph sparsification algorithm, which constructs a sparsifier with fewer edges but maintains comparable graph Laplacian matrices Anderson et al. (2014). Another approach, feGRASS, focuses on scalable power grid analysis, utilizing effective edge weight and spectral edge similarity to construct low-stretch spanning trees and recover spectrally critical off-tree edges, producing spectrally similar subgraphs efficiently Liu et al. (2022b). The MAC algorithm maximizes the algebraic connectivity of sparsified measurement graphs, providing high-quality sparsification results that retain the connectivity of the graph and the quality of corresponding SLAM solutions Doherty et al. (2022). LGRASS is a linear graph spectral sparsification algorithm designed to run in strictly linear time, optimizing bottleneck subroutines and leveraging spanning tree properties Chen et al. (2022). These algorithms are crucial for various applications, including power grid analysis, autonomous navigation, and other domains where graph analysis is essential.

D Supplementary Section D

D.1 Bayesian Optimization for Optimizing Timescales

The majority of contemporary studies involving Bayesian Optimization (BO) predominantly focus on problems of lower dimensions, given that BO tends to perform poorly when applied to high-dimensional issues (Frazier, 2018). Nevertheless, this study endeavors to leverage BO to fine-tune the neuronal and synaptic parameters within a diverse RSNN model. This application of BO implies the optimization of a substantial array of hyperparameters, making the use of conventional BO algorithms challenging. To address this, we employ an innovative BO algorithm, predicated on the notion that the hyperparameters under optimization are correlated and follow a specific probability distribution, as indicated by Perez et al.Perez-Nieves et al. (2021). Therefore, rather than targeting individual parameters, we modify BO to approximate *parameter distributions* for LIF neurons and STDP dynamics. Once optimal distributions are identified, samples are drawn to determine the model's hyperparameter distribution. To ascertain the data's probability distribution, alterations are made to BO's surrogate model and acquisition function to focus on parameter distributions over individual variables, enhancing scalability across all dimensions. The loss for updates to the surrogate model is determined using the Wasserstein distance between parameter distributions.

BO employs a Gaussian process to represent the objective function's distribution and utilizes an acquisition function to select points for evaluation. For target dataset data points $x \in X$ and corresponding labels $y \in Y$, an SNN with network structure $\mathcal{V}$ and neuron parameters $\mathcal{W}$ serves as a function $f_{\mathcal{V},\mathcal{W}}(x)$, mapping input data x to predicted label $\tilde{y}$. The optimization problem in this study is articulated as

$$\min_{\mathcal{V},\mathcal{W}} \sum_{x \in X, y \in Y} \mathcal{L}\left(y, f_{\mathcal{V},\mathcal{W}}(x)\right) \tag{6}$$

where $\mathcal{V}$ represents the neuron's hyperparameter set in $\mathcal{R}$ (Hyperparameter details are provided in the Supplementary), and $\mathcal{W}$ represents the multi-variate distribution comprising the distributions of various parameters, including the membrane time constants and the scaling function constants for the STDP learning rule in $\mathcal{S}_{\mathcal{R}\mathcal{R}}$.

Furthermore, BO requires a prior distribution of the objective function $f(\vec{x})$ based on the provided data $\mathcal{D}_{1:k} = \{\vec{x}_{1:k}, f(\vec{x}_{1:k})\}$. In GP-based BO, it is presumed that the prior distribution of $f(\vec{x}_{1:k})$ adheres to the multivariate Gaussian distribution, following a GP with mean $\vec{\mu}_{\mathcal{D}_{1:k}}$ and covariance $\vec{\Sigma}_{\mathcal{D}_{1:k}}$. We, therefore, calculate $\vec{\Sigma}_{\mathcal{D}_{1:k}}$ using a modified Matern kernel function, with the loss function being the Wasserstein distance between the multivariate distributions of different parameters. For higher-dimensional spaces, the Sinkhorn distance is utilized as a regularized variant of the Wasserstein distance to approximate it (Feydy et al., 2019).

$\mathcal{D}_{1:k}$ represents the points evaluated by the objective function. The GP will predict the mean $\vec{\mu}_{\mathcal{D}_{k:n}}$ and variance $\vec{\sigma}_{\mathcal{D}_{k:n}}$ for the remaining unevaluated data $\mathcal{D}_{k:n}$. The acquisition function in this study is the expected improvement (EI) of the prediction fitness as:

$$EI\left(\vec{x}_{k:n}\right) = \left(\vec{\mu}_{\mathcal{D}_{k:n}} - f\left(x_{\text{best}}\right)\right)\Phi(\vec{Z}) + \vec{\sigma}_{\mathcal{D}_{k:n}}\phi(\vec{Z}) \tag{7}$$

where $\Phi(\cdot)$ and $\phi(\cdot)$ represent the probability distribution function and the cumulative distribution function of the prior distributions, respectively. BO will select the data $x_j = \operatorname{argmax}\left\{EI\left(\vec{x}_{k:n}\right); x_j \subseteq \vec{x}_{k:n}\right\}$ as the next point for evaluation using the original objective function.

E Supplementary Section E

E.1 Algorithms

Algorithm 1 Compute Lyapunov Exponents

```
1:  **Input:** Batch of input sequences, number of time steps, $n$ different input sequences.
2:  **Output:** Computed Lyapunov Exponents.
3:  **Initialize:** Choose a set of sequences as the validation set.
4:  **for** each input sequence in the batch **do**
5:      Initialize matrix $\mathbf{Q}$ as identity matrix.
6:      Initialize hidden states $h_t$ as zeros.
7:  **end for**
8:  **for** each time step $t$ **do**
9:      Compute Jacobian $\mathbf{J}_t$ as:
            $[\mathbf{J}_t]_{ij} = \frac{\partial \mathbf{h}_t^j}{\partial \mathbf{h}_{t-1}^i}$
10:     Update $\mathbf{Q}$ and Compute $QR$ decomposition:
            $\mathbf{Q}_{t+1}, \mathbf{R}_{t+1} = QR(\mathbf{J}_t \mathbf{Q}_t)$
11: **end for**
12: **for** each $i^{th}$ vector at time step $t$ **do**
13:     Compute the $i^{th}$ LE $\lambda_i$ as:
            $\lambda_k = \frac{1}{T} \sum_{t=1}^{T} \log(r_t^k)$
14: **end for**
15: Compute the average LE for each input in the batch.
16: Normalize and interpolate the LE spectrum to retain the shape of the largest network size.
17: **return**Lyapunov Exponents.
```

Algorithm 2 Pruning Nodes with Lowest Betweenness Centrality

```
1: **Initialize:** $BC(v), T$
2: **Compute:** $BC(v) \forall v \in V$ using $BC(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$
3: **Identify:** $P = \{v \in V : BC(v) < T\}$
4: **Prune:** $G'(V', E')$ by removing $v \in P$ and associated edges
5: **Return:** $G'(V', E')$
```

References

Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, et al. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE transactions on computer-aided design of integrated circuits and systems*, 34(10):1537–1557, 2015.

Algorithm 3 Iterative Activity Pruning

```
1: Input: Recurrent Spiking Neural Network (RSNN) model $M$
2: Parameters: Pruning rate $r$, maximum number of iterations $T$
3: Output: Pruned and retrained RSNN model $M'$
4: Initialize iteration counter: $t = 0$
5: Evaluate the initial performance of the unpruned model $P_{\text{initial}}$
6: while $t < T$ and $P_{\text{current}} \geq 0.1 \cdot P_{\text{initial}}$ do
7:     Evaluate the activity of each neuron in model $M$ to obtain activity list $A$
8:     Sort the neurons in $A$ in ascending order based on activity
9:     Calculate the number of neurons to prune: $n_{\text{prune}} = \text{ceil}(r \cdot \text{number of neurons in } M)$
10:    Prune the $n_{\text{prune}}$ least active neurons from model $M$
11:    Retrain the pruned model $M$ to obtain the new model with updated parameters $M'$
12:    Evaluate the performance $P_{\text{current}}$ of the model $M'$
13:    if $P_{\text{current}} < 0.1 \cdot P_{\text{initial}}$ then
14:        Break the loop to prevent further degradation in performance
15:    end if
16:    Update model: $M = M'$
17:    Increment iteration counter: $t = t + 1$
18: end while
19: return Pruned and retrained model $M'$
```

David G. Anderson, M. Gu, and Christopher Melgaard. An efficient algorithm for unweighted spectral graph sparsification. 2014. URL `https://arxiv.org/abs/1410.4273`.

Anonymous. Towards energy efficient spiking neural networks: An unstructured pruning framework. In *Submitted to The Twelfth International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=eoSeaK4QJo`. under review.

Efe Camci, Manas Gupta, Min Wu, and Jie Lin. Qlp: Deep q-learning for pruning deep neural networks. 2022. doi: 10.1109/TCSVT.2022.3167951. URL `https://doi.org/10.1109/TCSVT.2022.3167951`.

Biswadeep Chakraborty and Saibal Mukhopadhyay. Characterization of generalizability of spike timing dependent plasticity trained spiking neural networks. *Frontiers in Neuroscience*, 15:695357, 2021.

Biswadeep Chakraborty and Saibal Mukhopadhyay. Heterogeneous recurrent spiking neural network for spatio-temporal classification. *arXiv preprint arXiv:2211.04297*, 2022. doi: 10.3389/fnins.2023.994517. URL `https://doi.org/10.3389/fnins.2023.994517`.

Biswadeep Chakraborty and Saibal Mukhopadhyay. Brain-inspired spiking neural network for online unsupervised time series prediction. *arXiv preprint arXiv:2304.04697*, 2023a.

Biswadeep Chakraborty and Saibal Mukhopadhyay. Heterogeneous neuronal and synaptic dynamics for spike-efficient unsupervised learning: Theory and design principles. In *Submitted to The Eleventh International Conference on Learning Representations*, volume 17, pp. 994517. Frontiers Media SA, 2023b. URL `https://openreview.net/forum?id=QIRtAqoXwj`. Accepted.

Biswadeep Chakraborty, Xueyuan She, and Saibal Mukhopadhyay. A fully spiking hybrid neural network for energy-efficient object detection. *IEEE Transactions on Image Processing*, 30:9014–9029, 2021.

Biswadeep Chakraborty, Uday Kamal, Xueyuan She, Saurabh Dash, and Saibal Mukhopadhyay. Brain-inspired spatiotemporal processing algorithms for efficient event-based perception. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6. IEEE, Publisher Name, 2023.

Yuxuan Chen, Jiyan Qiu, Zidong Han, and Chenhan Bai. Lgrass: Linear graph spectral sparsification for final task of the 3rd acm-china international parallel computing challenge. 2022. doi: 10.48550/arXiv.2212.07297. URL `https://doi.org/10.48550/arXiv.2212.07297`.

Sayeed Shafayet Chowdhury, Isha Garg, and K. Roy. Spatio-temporal pruning and quantization for low-latency spiking neural networks. 2021. doi: 10.1109/IJCNN52387.2021.9534111. URL `https://doi.org/10.1109/IJCNN52387.2021.9534111`.

Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99, 2018.

K. Doherty, David M. Rosen, and J. Leonard. Spectral measurement sparsification for pose-graph slam. 2022. doi: 10.1109/IROS47612.2022.9981584. URL `https://doi.org/10.1109/IROS47612.2022.9981584`.

Rainer Engelken, Fred Wolf, and Larry F Abbott. Lyapunov spectra of chaotic recurrent neural networks. *Physical Review Research*, 5(4):043044, 2023.

Jean Feydy, Thibault Séjourné, François-Xavier Vialard, Shun-ichi Amari, Alain Trouvé, and Gabriel Peyré. Interpolating between optimal transport and mmd using sinkhorn divergences. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pp. 2681–2690. PMLR, 2019.

Peter I Frazier. A tutorial on bayesian optimization. *arXiv preprint arXiv:1807.02811*, 2018.

Steve Furber. Large-scale neuromorphic computing systems. *Journal of neural engineering*, 13(5): 051001, 2016.

Erol Gelenbe, Zhi-Hong Mao, and Yan-Da Li. Function approximation with spiked random networks. *IEEE Transactions on Neural Networks*, 10(1):3–9, 1999.

Wulfram Gerstner and Werner M Kistler. Mathematical formulations of hebbian learning. *Biological cybernetics*, 87(5):404–415, 2002.

Wulfram Gerstner, Raphael Ritz, and J Leo Van Hemmen. Why spikes? hebbian learning and retrieval of time-resolved excitation patterns. *Biological cybernetics*, 69(5):503–515, 1993.

Wenjun Guo, Xianghong Lin, and Xiaofei Yang. A supervised learning algorithm for recurrent spiking neural networks based on bp-stdp. In *International Conference on Neural Information Processing*, pp. 583–590. Springer, 2021. doi: 10.1007/978-3-030-92307-5_68. URL `https://doi.org/10.1007/978-3-030-92307-5_68`.

Bing Han, Feifei Zhao, Yi Zeng, and Guobin Shen. Developmental plasticity-inspired adaptive pruning for deep spiking and artificial neural networks. 2022. doi: 10.48550/arXiv.2211.12714. URL `https://doi.org/10.48550/arXiv.2211.12714`.

Zhiqiang He, Yaguan Qian, Yuqi Wang, Bin Wang, Xiaohui Guan, Zhaoquan Gu, Xiang Ling, Shaoning Zeng, Haijiang Wang, and Wujie Zhou. Filter pruning via feature discrimination in deep neural networks. 2022. doi: 10.1007/978-3-031-19803-8_15. URL `https://doi.org/10.1007/978-3-031-19803-8_15`.

T. Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. 2021. URL `https://arxiv.org/abs/2102.00554`.

Nicolangelo Iannella and Andrew D. Back. A spiking neural network architecture for nonlinear function approximation. *Neural Networks*, 14(6):933–939, 2001. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(01)00080-6. URL `https://www.sciencedirect.com/science/article/pii/S0893608001000806`.

Mina A Khoei, Amirreza Yousefzadeh, Arash Pourtaherian, Orlando Moreira, and Jonathan Tapson. Sparnet: Sparse asynchronous neural network execution for energy efficient inference. In *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 256–260. IEEE, 2020.

Daehyun Kim, Biswadeep Chakraborty, Xueyuan She, Edward Lee, Beomseok Kang, and Saibal Mukhopadhyay. Moneta: A processing-in-memory-based hardware platform for the hybrid convolutional spiking neural network with online learning. *Frontiers in Neuroscience*, 16:775457, 2022.

Seijoon Kim, Seongsik Park, Byunggook Na, and Sungroh Yoon. Spiking-yolo: Spiking neural network for energy-efficient object detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 11270–11277, 2020.

Uday Kulkarni, Sitanshu S Hallad, A. Patil, Tanvi Bhujannavar, Satwik Kulkarni, and S. Meena. A survey on filter pruning techniques for optimization of deep neural networks. 2022. doi: 10.1109/I-SMAC55078.2022.9987264. URL `https://doi.org/10.1109/I-SMAC55078.2022.9987264`.

Jaehoon Lee, Yasaman Bahri, Roman Novak, Samuel S Schoenholz, Jeffrey Pennington, and Jascha Sohl-Dickstein. Deep neural networks as gaussian processes. *arXiv preprint arXiv:1711.00165*, 2017.

Fangxin Liu, Wenbo Zhao, Yongbiao Chen, Zongwu Wang, and Fei Dai. Dynsnn: A dynamic approach to reduce redundancy in spiking neural networks. 2022a. doi: 10.1109/icassp43922.2022.9746566. URL `https://doi.org/10.1109/icassp43922.2022.9746566`.

Zhiqiang Liu, Wenjian Yu, and Zhuo Feng. fegrass: Fast and effective graph spectral sparsification for scalable power grid analysis. 2022b. doi: 10.1109/TCAD.2021.3060647. URL `https://doi.org/10.1109/TCAD.2021.3060647`.

Emre O Neftci, Hesham Mostafa, and Friedemann Zenke. Surrogate gradient learning in spiking neural networks. *IEEE Signal Processing Magazine*, 36:61–63, 2019.

P. Panda and K. Roy. Learning to generate sequences with combination of hebbian and non-hebbian plasticity in recurrent spiking neural networks. 2017. doi: 10.3389/fnins.2017.00693. URL `https://doi.org/10.3389/fnins.2017.00693`.

Nicolas Perez-Nieves, Vincent CH Leung, Pier Luigi Dragotti, and Dan FM Goodman. Neural heterogeneity promotes robust learning. *bioRxiv*, 12(1):2020–12, 2021.

Filip Ponulak and Andrzej Kasinski. Introduction to spiking neural networks: Information processing, learning and applications. *Acta neurobiologiae experimentalis*, 71(4):409–433, 2011.

Muhammad Sabih, Ashutosh Mishra, Frank Hannig, and Jürgen Teich. Mosp: Multi-objective sensitivity pruning of deep neural networks. 2022. doi: 10.1109/IGSC55832.2022.9969374. URL `https://doi.org/10.1109/IGSC55832.2022.9969374`.

Sadra Sadeh and Stefan Rotter. Distribution of orientation selectivity in recurrent networks of spiking neurons with different random topologies. *PloS one*, 9(12):e114237, 2014.

Yasufumi Sakai, Yusuke Eto, and Yuta Teranishi. Structured pruning for deep neural networks with adaptive pruning rate derivation based on connection sensitivity and loss function. 2022. doi: 10.12720/jait.13.3.295-300. URL `https://doi.org/10.12720/jait.13.3.295-300`.

Subhrajit Samanta, Mahardhika Pratama, and Suresh Sundaram. Bayesian neuro-fuzzy inference system for temporal dependence estimation. *IEEE Transactions on Fuzzy Systems*, 29(9):2479–2490, 2020.

Xueyuan She, Saurabh Dash, Daehyun Kim, and Saibal Mukhopadhyay. A heterogeneous spiking neural network for unsupervised learning of spatiotemporal patterns. *Frontiers in Neuroscience*, 14:1406, 2021.

David Sussillo and Omri Barak. Opening the black box: low-dimensional dynamics in high-dimensional recurrent neural networks. *Neural computation*, 25(3):626–649, 2013.

S. Vadera and Salem Ameen. Methods for pruning deep neural networks. 2020. doi: 10.1109/ACCESS.2022.3182659. URL `https://doi.org/10.1109/ACCESS.2022.3182659`.

Werner van der Veen. Including stdp to eligibility propagation in multi-layer recurrent spiking neural networks. 2022. URL `https://arxiv.org/abs/2201.07602`.

Ryan Vogt, Maximilian Puelma Touzel, Eli Shlizerman, and Guillaume Lajoie. On lyapunov exponents for rnns: Understanding information propagation using dynamical systems tools. *arXiv preprint arXiv:2006.14123*, 8:818799, 2020.

Meiling Xu, Min Han, CL Philip Chen, and Tie Qiu. Recurrent broad learning systems for time series prediction. *IEEE transactions on cybernetics*, 50(4):1405–1417, 2018a.

Meiling Xu, Min Han, Tie Qiu, and Hongfei Lin. Hybrid regularized echo state network for multivariate chaotic time series prediction. *IEEE transactions on cybernetics*, 49(6):2305–2315, 2018b.

Ruokai Yin, Youngeun Kim, Yuhang Li, Abhishek Moitra, N. Satpute, Anna Hambitzer, and P. Panda. Workload-balanced pruning for sparse spiking neural networks. 2023. doi: 10.48550/arXiv.2302.06746. URL `https://doi.org/10.48550/arXiv.2302.06746`.

Wenrui Zhang and Peng Li. Temporal spike sequence learning via backpropagation for deep spiking neural networks. *Advances in Neural Information Processing Systems*, 33:12022–12033, 2020.