Acknowledgements

I would first like to thank my father and my mother for their sacrifices and encouragements throughout all the successes and failures of my life. I am forever grateful for their love and unconditional support which has lead me to accomplish far more than I could have ever imagined. I would also like to thank my sister Sofia for the unique "sisterly" support she has provided me over the years, I hope to always be a big brother you can be proud of. Special thanks to my grandparent: Antonio, Luciana, Ugo and Vittoria, who have always been there for me with love and affection.

I am deeply grateful to my supervisor, Prof. Riccardo Zecchina, for instilling in me a passion for research and science, and for his invaluable guidance and support throughout the development of this work.
I also wish to extend a special thank you to my high school professor, Dr. Emanuele Ciancio, whose teaching first introduced me to the beauty of mathematics, which I continue to pursue with enthusiasm ever since.

Finally, I am thankful to my friends:Alberto, Alessandro, Ali, Federico, Lorenzo, Luca, Mariano, Sofia, Stefano, Teodoro and especially to my girlfriend, Chiara, for filling this journey with laughter, encouragement, and unwavering support. Their presence has made even the most challenging times manageable and the successes all the more meaningful, and I am deeply grateful to have shared this experience with them.

Contents

Introduction

Many of the foundational innovations in machine learning have their roots in attempts to model aspects of the brain. Early artificial neural networks were explicitly inspired by biological neural circuitry: McCulloch and Pitts' formalization of a "neuron" in 1943 and Rosenblatt's perceptron in 1958 directly borrowed the idea of interconnected neurons and synapses.
Subsequent developments, such as convolutional architectures and Hopfield networks, each drew on biological analogues to improve learning and memory. While the field of machine learning has already made significant advances, with remarkable achievements in computer vision, natural language processing, and strategic game playing, implementing more biologically plausible models may prove instrumental in solving key issues with today's modern AI architectures.

One significant motivation for continuing to study and experiment with biologically-inspired approaches is the stark contrast in energy efficiency between biological and current artificial systems. Following the pioneering work of Kaplan et al. [23], which showed that the performance of large language models scales predictably with the number of parameters, the community has largely pursued ever-larger architectures as the primary path to improved accuracy.
This strategy has indeed been wildly successful; however the exponential increase in complexity of the models has also led to an exponential increase in the total Floating Point Operations (FLOPs) required to train them. FLOPs are directly related to the energy consumption of the models, and as such can be used as a proxy for the energy efficiency of the models. According to current estimates [18, 23] the total FLOPs scale supra-linearly with the number of parameters and can vary from $O(N^{1.37})$ to $O(N^2)$, where N is the number of parameters in the model. The simultaneaous increasing of model size paired

with the supra-linear relationship between FLOPs and size has lead to massive energy requirements to train the latest generations of models.
This stands in stark contrast with the brain, whose reported energy consumption grows as $O(N)$ with the number of neurons [14]; Moreover, under normal conditions a human brain-composed of roughly 9×10^{10} neurons and 10^{14} synapses — operates only on approximately 20 W [5].

There are multiple factors underlying this efficiency gap, of which two are particularly noteworthy:
First, biological learning relies on local update rules, synaptic modifications that depend only on signals available at the level of individual neurons, whereas backpropagation requires global error signals to be propagated through every layer of a network [38].
Second, the brain employs extreme sparsity both in its connectivity and in neuronal firing: at any given moment only a small fraction of neurons and synapses are active, dramatically reducing energy expenditure. These observations suggest that incorporating local learning mechanisms and sparse event-driven processing into artificial models could yield substantial gains in efficiency without sacrificing performance.

The content of the thesis will be divided into two parts:
In Part I, we review the biological foundations of learning and memory, presenting a systematic outline of key models, in particular associative networks, random recurrent networks and reservoir computing, and the neurophysiological principles that underlie them.
In Part II, we introduce Multi-Layer Chain Models (MLCMs), a novel framework that structures learning as a sequence of locally updated layers inspired by biological principles. We begin by motivating the architecture through recent findings on the stabilizing effects of random recurrent networks. The mathematical formulation of MLCMs is then developed, including the dynamics of each layer and the mechanisms that allow for local, biologically plausible learning. We further discuss the role of binary and sparse activations in enhancing both biological realism and computational efficiency. Finally, we present extensive empirical results, comparing MLCMs to conventional architectures and conclude by outlining future research directions for biologically-inspired machine learning.

Part I: Foundations of Biological Learning Models

Chapter 1

Neuron Models and Classical Learning Rules

1.1 Artificial Neuron Model

The standard perceptron model, originally proposed by McCulloch and Pitts in 1943 [29] and later reformulated by Rosenblatt in 1958 [33], abstracts the neuron as a device that receives real-valued inputs $\mathbf{x} \in \mathbb{R}^N$. Each input x_i is modulated by a synaptic weight w_i, and the neuron computes the weighted sum:

$$s = \sum_{i=1}^{N} w_i \, x_i = \mathbf{w} \cdot \mathbf{x}.$$

This sum is compared to a threshold $-b$, yielding a binary output in $\{-1, 1\}$ via the sign activation:

$$\hat{y}(\mathbf{x}) = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b).$$

This methods can also be modified to produce an output in $\{0, 1\}$ by using the Heaviside function.While mathematically tractable, this model omits many biological details (e.g. temporal dynamics, nonnegative constraints, nonlinear integration), focusing instead on a minimal computational abstraction.

The Perceptron as a Classifier

Fixing the parameters $(\mathbf{w}, b)$, the perceptron implements a function:

$$\hat{y} : \mathbb{R}^N \to \{-1, 1\},$$

which can be interpreted as a binary classifier. Input data of arbitrary form is vectorized into $\mathbf{x} \in \mathbb{R}^N$, and classified by the sign of the affine form.

Geometric Interpretation

Geometrically, the perceptron defines a hyperplane in $\mathbb{R}^N$ given by

$$\{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} + b = 0\}$$

which partitions space into two half-spaces corresponding to the two labels. The normal vector $\mathbf{w}$ determines the orientation of the hyperplane, while the bias b controls its offset from the origin. Notably, scaling $(\mathbf{w}, b)$ by any positive constant leaves the classification boundary invariant, though the norm of $\mathbf{w}$ can affect learning dynamics.

1.2 Classical Learning Rules

Learning rules define how synaptic weights adapt to a task through iterative adjustments based on observed data. They can be broadly categorized into two classes:

- **Local Learning Rules:** In a physical neural system, the update of each synapse depends only on variables accessible locally at that synapse, both in space and time [6, Chapter 7]. Local rules are biologically plausible, requiring no global coordination or knowledge of distant network states.
- **Global Learning Rules:** These updates rely on information that may not be available at individual synaptic sites, such as error signals computed from a network-wide objective. They have become the default way to train deep artificial neural networks due to their ability to better propagate errors between layers, though they are less biologically plausible because they depend on nonlocal information.

We now quickly review some of the most common local and global learning rules.

Perceptron Learning Rule

For a binary neuron with output $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x} + b)$ and target $y \in \{\pm 1\}$, the perceptron rule updates only on errors:

$$\Delta w_i = \eta \ (y - \hat{y}) \ x_i$$

where η is the learning rate. The update is proportional to the input x_i and the error δ. Through the *Perceptron Convergence Theorem* we can prove that this rule converges in finite steps for linearly separable data.

Hebbian Rule

This rule is based on the principle of Hebbian learning, which states that "cells that fire together, wire together". It captures the correlation between presynaptic and postsynaptic activity:

$$\Delta w_i = \eta \, x_i \, y$$

where x_i is presynaptic activity of the i-th neuron and y postsynaptic activity. It captures correlations but can lead to unbounded weight growth.

Oja's Rule

A normalized Hebbian variant that limits weight magnitude:

$$\Delta w_i = \eta \left(x_i \, y - \frac{y^2}{\lambda} \, w_i \right), \quad \lambda > 0$$

with λ controlling the weight decay. This rule converges to the first principal component of the input data.

Delta Rule

Also known as the Widrow-Hoff or least-mean-squares rule, for a linear neuron $y = \mathbf{w} \cdot \mathbf{x}$ with target t:

$$\Delta w_i = \eta \ (t - y) \ x_i$$

it minimizes the mean squared error between the target and output. This rule is equivalent to gradient descent on on the squared error $\frac{1}{2}\delta^2$.

Backpropagation

A global rule for multi-layer networks. It's the most used learning algorithm in deep learning, and it is based on the chain rule of calculus. For weight w_{ij}^l in layer l:

$$\Delta w_{ij}^l = \eta \, \delta_j^l \, y_i^{l-1}, \quad \delta_j^l = \frac{\partial E}{\partial y_j^l} \, \sigma'(y_j^l)$$

where E is the error function, y_i^{l-1} is the output of the previous layer, and σ is the activation function. By propagating δ backward, it minimizes a global loss via gradient descent.

Chapter 2

The Hopfield Network

Associative memory models constitute a class of Recurrent Neural Networks designed to store and retrieve patterns by means of content-addressable recall. In contrast to feedforward architectures, these models exploit rich feedback connections to compare an input cue with a set of stored prototypes and dynamically converge to the closest match. The principal aim of associative memory is twofold: (i) to robustly recover complete patterns from partial or noisy inputs, and (ii) to continuously incorporate novel experiences without catastrophic interference.

In this chapter, we provide an overview of one of the most prominent models of associative memory: the Hopfield network. We begin by exploring its theoretical foundations, followed by an analysis of its key properties and inherent limitations.

2.1 Framework

The Hopfield network is one of the first models of associative memory [19, 20], it's a recurrent neural network of binary states which is capable of storing a set of patterns by encoding them into the weights through the use of the Hebb rule.

Mathematically, it is defined by a set of N binary neurons whose states at time t are denoted:

$$s_i(t) \in \{-1, +1\}, \qquad i = 1, \ldots, N.$$

Each unordered pair of neurons (i, j) is connected by symmetric weights $J_{ij} = J_{ji}$, and we

enforce $J_{ii} = 0, \ \ \forall i$. A canonical choice of the synaptic matrix is given by the Hebbian learning rule:

$$J_{ij} \ = \ \frac{1}{N} \sum_{\mu=1}^{p} \xi_i^\mu \, \xi_j^\mu, \qquad i \neq j \tag{2.1}$$

where $\{\xi^\mu\}_{\mu=1}^{P}$ are the P prototype patterns to be stored (each $\xi_i^\mu \in \{-1, +1\}$). As we will see later, this prescription guarantees that each stored pattern is a stable fixed point of the network dynamics[1].

Updates are performed asynchronously: at each time step one neuron i is selected at random and its state is updated according via the rule:

$$s_i(t+1) \ = \ \text{sign}\big(\sum_{j=1}^{N} J_{ij} \, s_j(t) \big) \tag{2.2}$$

Under this update scheme, we can treat the Hopfield networks as an Ising model at zero temperature governed by the Lyapunov (energy) function:

$$E(\mathbf{s}\,(t)) \ = \ -\frac{1}{2} \sum_{i,j} J_{ij} \, s_i\,(t) \ s_j\,(t) \ , \tag{2.3}$$

which can be easily shown to monotonically decreases (or remains constant) with each asynchronous update.

Biological inconsistencies

One of the strengths of the Hopfield model is that it's simple enough to be analytically tractable. However this simplification of also gives rise to some notable biological inconsistencies [16]:

1. *Symmetric interaction:* The symmetric interaction between states is a very unrealistic assumption due to Dale's Rule, which states that neurons can either have a purely excitatory or inhibitory role. However, it was judged a necessary sacrifice by Hopfield as it allows us to treat the network as an Ising model with a well defined energy function.

2. *Spurious potentiation during joint silence:* The current formulation of the weights

[1] in the limit of low loading p/N

erroneously strengthens synapses even when neither neuron is active, a phenomenon not supported by experimental neurophysiology.

3. *Excitatory-inhibitory synaptic flips:* As more patterns are stored (p increases), a synapse that was initially excitatory has the potential of becoming inhibitory, once again due to Dale's law this is not a realistic phenomenon.

2.2 Attractor states

The main property of the Hopfield Network is its ability to store and retrieve patterns. In practice this happens because the Hebb rule ensures that the stored patterns are not only fixed points of the dynamics, but also function as attractors; meaning that initializing the network with a states that is close in hamming distance to one of the stored patterns will cause the Hopfield network to converge towards that pattern.

Although true also in the general case (up to certain capacity constraints that we will explore later), let's examine the stability claim of a particular stored pattern ξ^ν in the case of P randomly drawn i.i.d. patterns $\{\xi^\mu\}_{\mu=1}^P$ and $N \to \infty$.
Being a fixed point corresponds to:

$$\xi_i^\nu = \text{sign}\left(\sum_j J_{ij}\xi_j^\nu\right) = \text{sign}\left(\frac{1}{N}\sum_{j,\,\mu} \xi_i^\mu\, \xi_j^\mu\, \xi_j^\nu\right) \quad \forall i. \tag{2.4}$$

Separating the contribution of ν from the rest of the patterns:

$$\xi_i^\nu = \text{sign}\left(\xi_i^\nu + \frac{1}{N}\sum_{j,\,\mu\neq\nu} \xi_i^\mu\, \xi_j^\mu\, \xi_j^\nu\right) \tag{2.5}$$

But we can see that $X_j := \xi_j^\mu \sum_{\mu\neq\nu}^P \xi_i^\mu \xi_j^\mu$ is a random variable with: $\mathbb{E}[X_j] = 0$ and $V(X_j) = P - 1$. Thus we can apply the central limit theorem to get:

$$\frac{1}{N}\sum_{j,\,\mu\neq\nu} \xi_i^\mu\, \xi_j^\mu\, \xi_j^\nu = \frac{1}{N}\sum_j X_j \sim N\left(0, \frac{P-1}{N}\right) \tag{2.6}$$

We conclude that, in the thermodynamic limit $N \to \infty$, the noise term becomes negligible and thus we verify with probability one that the pattern is indeed a fixed point.

The stable configurations of the network dynamics correspond to local minima of the energy function defined in (2.3). Geometrically, one may view the energy as a real-valued surface over the 2^N vertices of an N-dimensional hypercube, each vertex representing a possible network state $\mathbf{s} \in \{-1, +1\}^N$. Under the asynchronous update rule, each individual spin flip can only decrease or leave unchanged the network's energy [16]. Analogously to a ball rolling downhill on a rugged landscape and settling in the nearest valley, the network, starting from any initial vertex, successively moves (or remains) to neighboring vertices of decreasing energy until it becomes trapped in a local minimum of the energy surface.

Spurious states

It has been observed that, in addition to the P desired patterns, the Hopfield network admits 3 other types of attractors arising from the Hebbian connectivity.

1. *Complementary Patterns:* Trivially, each stored pattern ξ^μ has its exact negation $-\xi^\mu$ as an attractor, since the energy function (2.3) is invariant under the transformation $\mathbf{s} \mapsto -\mathbf{s}$. These complements are generally of little concern, since they lie at the same energy level as the intended memories and can be distinguished by an overall sign flip.

2. *Mixture States:* Linear combinations of an odd number of stored patterns can themselves form fixed points [2]. For example, given three patterns ξ^1, ξ^2, ξ^3, one can construct:

$$\xi_i^{\text{mix}} \;=\; \operatorname{sgn}\!\left(\xi_i^1 \pm \xi_i^2 \pm \xi_i^3\right),$$

   These mixture states lie at intermediate Hamming distance from each constituent pattern and satisfy the fixed point's stability conditions for moderate loadings.

3. *Spin-Glass States:* When the number of stored patterns p becomes large, the random superposition of many weakly correlated patterns creates a highly rugged energy landscape populated by *spin-glass* minima [3]. These states bear no significant overlap with any single stored pattern and arise purely from interference among many stored memories.

The second and third type are called *spurious states* and, even though they usually have

small basins of attractions compared to the retrieval states, they tend to be problematic and degrade retrieval performance.

To mitigate these effects, researchers have introduced stochasticity into the update rule, effectively turning the network into a finite temperature Ising model. This system can escape shallow local minima and more thoroughly explore the state space, thereby reducing the prevalence of spurious states.

The seminal work carried by Little, Peretto, Hinton and others [17,24,32] implement this idea by replacing the deterministic sign update with a probabilistic rule:

$$P\big(S_i(t+1) = \pm 1\big) \;=\; \frac{\exp\big(\pm\beta\, h_i(t)\big)}{\exp\big(\beta\, h_i(t)\big) + \exp\big(-\beta\, h_i(t)\big)},$$

where:

$$h_i(t) \;=\; \sum_j J_{ij}\, S_j(t)$$

is the local field, and the inverse temperature β controls the noise level. In the long-time limit, the network converges to the Boltzmann distribution:

$$P(S_1, \ldots, S_N) \;=\; \frac{1}{Z}\, \exp\big(-\beta\, E(S)\big),$$

with energy 2.3 and partition function Z.

2.3 Capacity considerationss

In the thermodynamic limit $N \to \infty$, two complementary notions of capacity arise: one concerning exact, error-free recall (sublinear regime) and one concerning extensive recall (linear regime).

1. Perfect Recall: sublinear pattern storage

The maximal number $P_{\max}$ of patterns that can be stored and retrieved with zero errors grows only sublinearly, with the well-known bound [16]:

$$P_{\max} \;=\; \frac{N}{2\, \log_2 N}\,, \tag{2.7}$$

Which is determined from combinatorial and information-theoretic arguments. Indeed if P grows faster than $O(N/\log N)$, then exact retrieval fails almost surely. Thus, in the zero-error regime one must restrict $P \ll N$ to guarantee that all stored patterns are stable attractors.

2. Extensive Recall: linear loading ratio

If one instead allows for a small error in the recollection, we can consider a more extensive number of patterns $P = \alpha N$, with $\alpha > 0$ fixed, then there exists a critical loading:

$$\alpha_c \approx 0.138\,, \tag{2.8}$$

Above which pattern retrieval catastrophically breaks down. Amit, Gutfreund and Sompolinsky [1,3,4] first derived α_c via replica-symmetric mean-field theory, identifying it as the boundary between the retrieval and spin-glass phases at zero temperature. Experimental and numerical studies corroborate these findings as well.

Chapter 3

Random Recurrent Neural Networks

Particular effort has been devoted by researchers to study Hopfield-type models that allow for random and asymmetric synaptic connections [10, 12, 36, 39]. This interest stems not only from a desire of greater biological plausibility, but also from results claiming that the inclusion of asymmetric part to the Hopfield model can lead to a reduction in the number of spurious states while largely preserving the retrieval capacity of the network [8, 15, 31]. These approaches usually focus on perturbing the hebbian weights obtained in the classic Hopfield framework. This is accomplished either through random dilution the network (ie. setting some weights to zero) or by the direct inclusion random noise to the weights. Thus while the network is made more robust through the inclusion of noise, the core idea of Hebbian connectivity is maintaned.

However, despite these encouraging results, all of the above asymmetric-weight schemes remain fundamentally tethered to the Hopfield paradigm: the underlying dynamics and Hebbian backbone are unchanged, and the random or diluted terms enter only as small perturbations. Consequently, they inherit the same physiological inconsistencies as the original model, such as joint-silence potentiation, since the core conceptual framework is still that of the Hopfield network.

Recent works [11,34] have abandoned the Hopfield ansatz altogether and examined *purely random* associative networks, in which each s ynaptic weight J_{ij} is drawn independently from a prescribed distribution (e.g. Gaussian).
However, purely random associative networks also present serious limitations, chief among

them the lack of any guaranteed convergence mechanism. In the absence of symmetric couplings or a well-defined energy function, their update dynamics can exhibit oscillations or even chaotic trajectories, rendering memory retrieval unreliable.
In the following chapter, we extend the investigations of these works by introducing and evaluating two stabilization methods that restore convergence to fixed-point attractors.

3.1 Stabilization through Self-couplings

We first investigate the role of self-couplings in the dynamics of random networks. Previous works have shown that the inclusion of self-couplings in a neuron as a method to prevent excessive oscillations has shown promising stabilizing effects [11].
Mathematically, we model the self-coupling term J_D as a weight from neuron i to itself. Thus the updated dynamics for this type of network become:

$$s_i(t+1) \;=\; \text{sign}\left(J_D\, s_i(t) + \sum_{j\neq i} J_{ij}\, s_j(t)\right) \tag{3.1}$$

In this case the value of J_D is set to be positive and is taken to be the same for all neurons.

It's clear to see that the self-coupling term acts as a stabilizing force, as it works to oppose changes in the state of the neuron.
However, $J_D \gg 1$ leads to a trivial model as the self-coupling term overpowers completely the other terms of the dynamics and each configuration becomes a fixed point of the network.
The self-coupling term is thus a double-edged sword: it can be used to successfully stabilize the network and induce convergences, but if used excessively it can lead to trivial dynamics.
The objective then becomes to find reasonable values of J_D that allow for non-trivial but stable dynamic. To this end we introduce the concept of *critical self-coupling* J_D^*, which we define to be the infimum of the values of J_D such that the network is able to fully converge for any combination of weights and states.

Earlier works have taken both theoretical approaches [34], showing through stastical

physics methods that the logarithm of the (normalized) number of fixed points of the network as a function of J_D has form:

$$\log\left[1+\operatorname{erf}\left(\frac{J_D}{\sqrt{2}}\right)\right], \quad \text{when } \beta\to\infty \text{ and } N\to\infty; \tag{3.2}$$

as well as a numerical approaches [11], where the focus was on implementing simulations of the network dynamics to measure the number of fixed points for different values of J_D and different network sizes.

Our contribution to this line of work has been to massively scale up the numerical experiments by developing a highly optimized C implementation of the model, allowing us to expand from the networks of 8,000 neurons found in previous studies to over 100,000 neurons; enabling for robust measurement of J_D^* across large variations in system size.

We present two main results: the first is the behaviour of the critical self-coupling as a function of the network size, and the second is the convergence time (in iterations) of the states in the network as a function of the self-coupling.

Experimental Setup

We implement a fully-connected network of size N with:

- Binary states $s_i(t) \in \{-1, +1\}$
- Random i.i.d binary weights $\mathbb{P}\left(J_{ij} = \frac{1}{\sqrt{N}}\right) = \mathbb{P}\left(J_{ij} = -\frac{1}{\sqrt{N}}\right) = 0.5 \quad \forall i \neq j$
- Self-coupling term J_D s.t. $J_{ii} = J_D \quad \forall i$

The update rule is given by (3.1) and we perform synchronous updates, as it provides greater computational efficiency without altering the behaviour of the network.

The estimation of the J_D^* is done by randomly drawing the weights for a network of size N and then sampling various initial states to verify their convergence for a given range of self-coupling values J_D. This procedure is repeated multiple times to ensure that the results are not biased by the choice of weights.

The final estimate of J_D^* is taken to be maximum value of J_D such that all the initialization of all the weight configurations sampled converged.

The exact number of times that the network is re-initialized and the number of samples

from each initialization varies and depends on the size of the network since larger networks are more computationally expensive to simulate and vary less compared to smaller networks.

Critical Self-Coupling

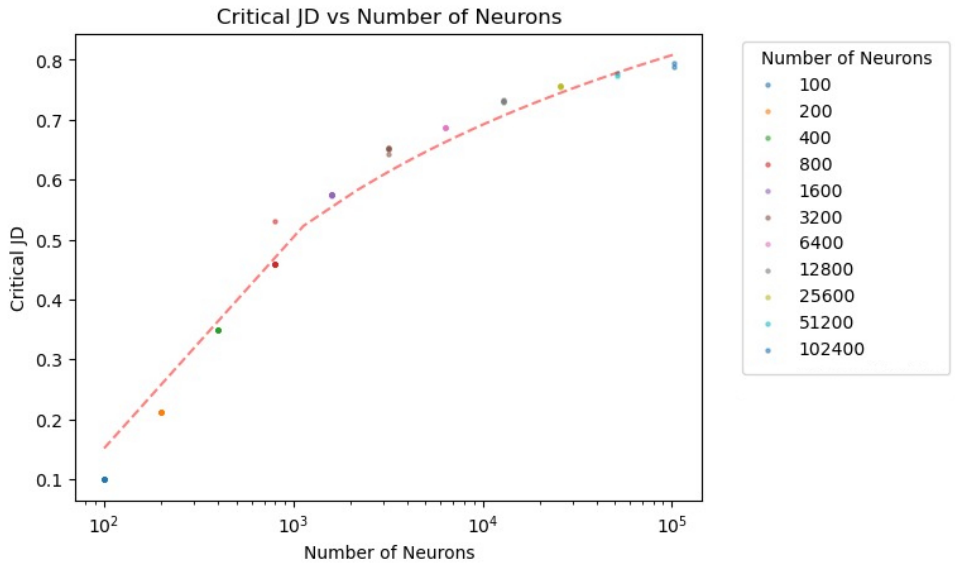


Figure 3.1: Behavior of the critical self-coupling J_D^* as a function of the network size N

We report the results of our numerical experiments in figure 3.1 where we plot the critical self-coupling J_D^* on the y-axis and the network size N on the x-axis in log scale.
Additionally, we also plot an interpolation of the data to surmise a functional form for the behaviour of J_D^*.

What is immediately noticeable is that the critical self-coupling J_D^* increases with the network size N. However this growth does not appear to be linear even in the log scale, indicating that the relationship between the two variables has an upper bound lower than $O(\log(N))$.
Determining the exact functional form of the relationship though is not trivial, and further work is needed to establish a more precise relationship.

Convergence Times

Figure 3.2 shows the number of asynchronous update iterations required for full convergence as a function of the self-coupling J_D, each colour indicating a different network

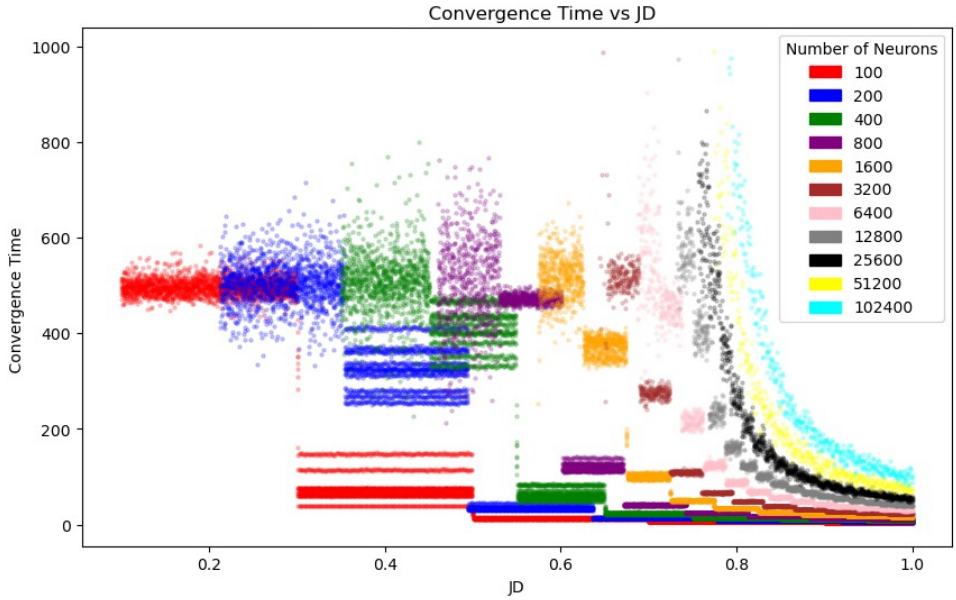


Figure 3.2: Convergence times as a function of the self-coupling J_D

size.

As a confirmation of our previous results, we see that larger networks only start converging for larger values of J_D. Indeed we notice a progressive shift to the right of the curves as N increases.

But by far the most interesting feature of this graph is the step-like plateaus: for smaller networks ($N \leq$ 12,800), the data align on discrete horizontal steps each corresponding to a fixed integer number of sweeps through the neurons before stabilization. These plateaus arise from the granularity of the fields each neuron receives. In smaller networks the values that the field $h_i(t) = \sum_{j \neq i} J_{ij}\, s_j(t)$ can take are relatively far apart so that small changes in the self-coupling J_D do not contribute to changing the sign in 3.1.

Indeed, as N increases, the width of each plateau in J_D-space shrinks, reflecting a greater sensitivity to small changes of J_D. Beyond roughly N = 25,600, the step structure becomes imperceptible and the convergence time $T_c(J_D)$ is seemingly well described by a power law:

$$T_c(J_D) \;\propto\; (J_D - J_D^*)^{-\gamma},$$

Though this is currently speculation and should be the object of further analyses.

3.2 Stabilization through Sparse Positive Reinforcement

We now investigate a different approach to stabilizing the dynamics of random networks. In biological neural circuits, a small subset of synapses often exhibits disproportionately large, stable strengths, forming a structural backbone that coexists with a much larger population of weaker, plastic connections [13, 35]. Inspired by this heterogeneity, we propose a simple stabilization scheme for random associative networks based on *sparse positive reinforcement*. The main idea is to overlay to the random synaptc matrix J a sparse matrix containing positive weights, which we will refer to as the *reinforcement matrix* A.

The network dynamics are then defined by the update rule:

$$s_i(t+1) \;=\; \text{sign}\left(\sum_{j\neq i}\left(J_{ij}+A_{ij}\right)\, s_j(t)\right) \tag{3.3}$$

In the following, we will examine how the parameters ρ (sparsity level) and a (reinforcement strength) jointly affect the emergence of robust attractors and the suppression of oscillatory or chaotic regimes.

3.2.1 Experimental Setup

We implement a network of size N with:

- Binary states $s_i(t) \in \{-1, +1\}$
- Random i.i.d weights $J_{ij} \sim N(0, \frac{1}{\sqrt{N}}) \quad \forall i \neq j, \quad J_{ii} = 0 \quad \forall i$
- Sparse reinforcement matrix A s.t. $A_{ij} = a > 0$ non-zero for entries of order $O\left(N^d\right)$ with $d \in (1, 2)$.

The update rule is given by (3.3) and we once again perform synchronous updates.

We generate the reinforcement matrix A by first fixing the strenght a and an order for the density $O\left(N^d\right)$ and then randomly selecting a density $\rho \sim N\left(N^d, \frac{1}{\sqrt{N^d}}\right)$ of non-zero entries. The position of the non-zero entries is chosen from a uniform distribution over the N^2 possible positions in the matrix.

3.2.2 Results

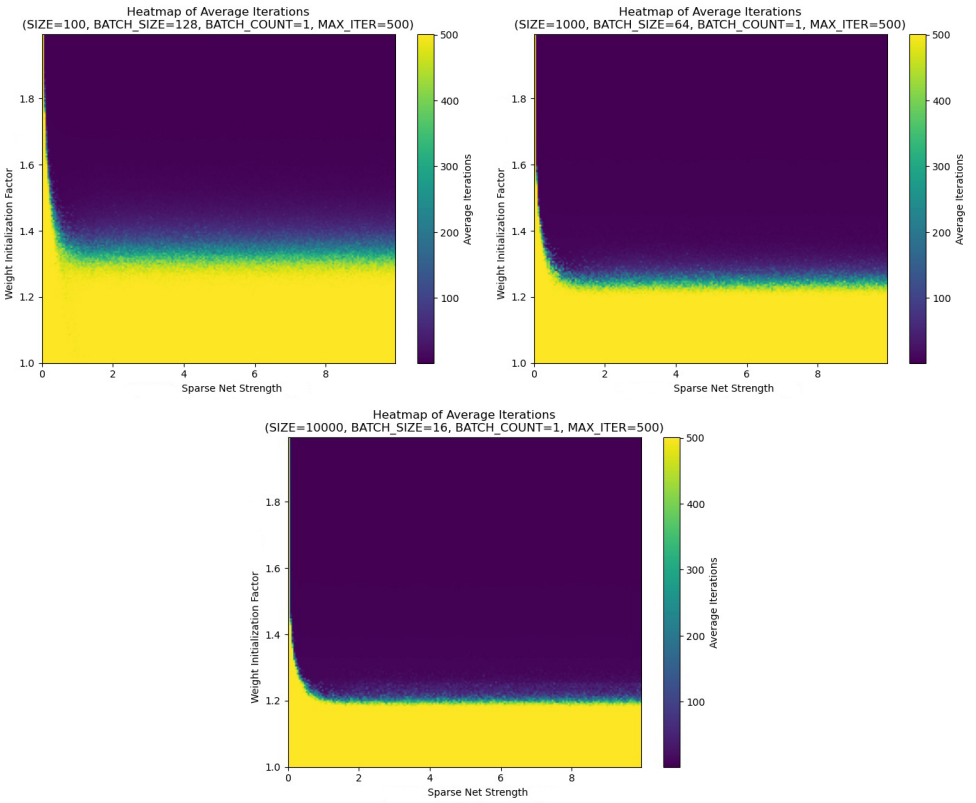


Figure 3.3: Comparison of average iterations for networks of size 100, 1000, and 10000. On the x-axis we have the reinforcement strength a and on the y-axis the exponential d determing the order $O\left(N^d\right)$ of the sparsity

Figure 3.3 shows the average number of iterations taken to converge for networks of different sizes N (100, 1000, and 10000). Since it is not possible to numerically verify whether a state does not converge, we set a maximum number of iterations $T_{\max} = 500$ and make the assumption that if the network does not converge within this time, then it is likely to be in a chaotic regime.

We can distinguish three distinct dynamical regimes in the sparsity-density plane (d, a) plane, whose extents depend markedly on network size N:

- *Chaotic regime (yellow):* For sparsity exponents d in approximately the range $1.0 \lesssim d \lesssim 1.2$, no value of reinforcement strength a yields convergence within the maximum iterations. This disorder-dominated region persists across all N, indicating that below a critical sparsity order the random background overwhelms the sparse foundation and the dynamics remain non-attracting.

- *Trivial regime (purple):* At very high d and/or very large a, almost every trial

converges in just a handful of iterations. Here the sparse reinforcement saturates the dynamics producing a trivial fixed point that carries little information about the initial condition. Further inspection indeed reveals that all states in this region converge to either the all $+1$ or all -1 states, which composed the only two fixed points of the dynamics.

- *Intermediate convergence regime (green/blue):* Sandwiched between the two extremes lies a band of parameter combinations for which convergence occurs after a nontrivial number of updates. This "sweet spot" represents the most interesting operational regime, where the sparse positive backbone and the random substrate jointly give rise to genuine attractor dynamics.

The width of the intermediate (green/blue) band shrinks with increasing N. Thus, although sparse positive reinforcement can reliably induce convergence, the "parameter margin" for nontrivial dynamics becomes more stringent in very large systems.

Another interesting remark is that the lower boundary of the chaotic region, the minimal exponent $d_{\min}(N)$ beyond which convergence becomes possible decreases as N grows. In other words, larger networks allow for stable dynamics even in progressively sparser reinforcement (d) regimes, suggesting a scaling law $d_{\min}(N) \to d_\infty < 2$ that merits analytic characterization.

Overall, these heatmaps demonstrate that sparse positive reinforcement can stabilize large random networks, but only within a narrowing window of sparsity and strength. Highlighting the need for further theoretical insight into the asymptotic limits of d and a as $N \to \infty$.

Chapter 4

Reservoir Computing

Reservoir Computing (RC) is a computational framework particularly well-suited for processing temporal and sequential data. The fundamental idea behind RC is the use of a fixed, randomly connected recurrent neural network, known as the "reservoir", to project input data into a high-dimensional space.

Unlike traditional RNNs, the weights within the reservoir are not trained. Instead, only the weights of a simple linear "readout" layer connected to the reservoir's states are trained to perform the specific task.

The reservoir's complex, non-linear dynamics, driven by sequential input, create rich spatio-temporal patterns that capture the input's history and context. While not adapted to a specific task, the reservoir provides the computational preprocessing for a large range fo possible tasks; this can then be leveraged by a simple linear readout layer to perform complex computations. Essential properties for a functional reservoir include high dimensionality, nonlinearity, and fading memory, which ensures sensitivity to recent inputs [37].

In this chapter we explore the two main types of RC models: Liquid State Machines [27] and Echo State Networks [21].

4.1 Liquid State Machines

Liquid State Machines (LSMs) represent a specific type of Reservoir Computing model that was developed with a strong emphasis on the computational capabilities of biological

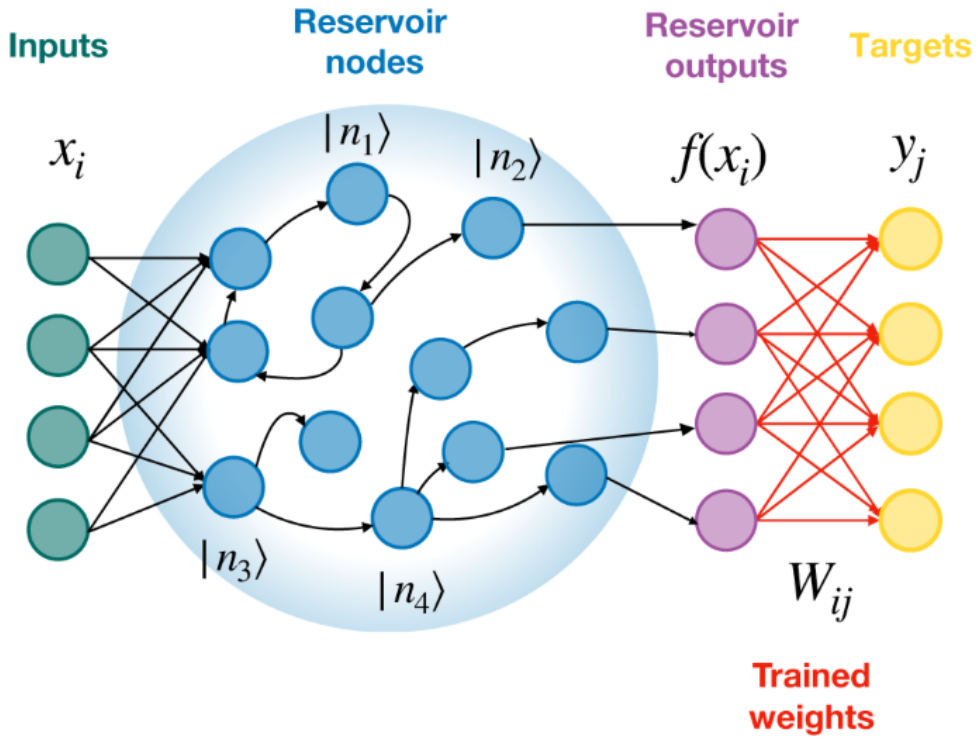


Figure 4.1: Reservoir Computing framework. The reservoir is a fixed recurrent neural network that transforms input data into a high-dimensional space. The readout layer is trained to perform the specific task.

neural microcircuits within the brain [25–27].

The primary motivation behind LSMs was to create biologically relevant learning models using spiking neural networks (SNNs) with recurrent connectivity.

The computational framework of an LSM is similar to the general RC architecture, consisting of a reservoir (usually known as the Liquid) and a readout layer. In LSMs, the liquid is typically composed of excitatory and inhibitory spiking neurons. The input $\mathbf{u}(t)$ to the system is often encoded as a spike sequence. This input drives the dynamics of the liquid, typically represented as L^M, transforming the input into high-dimensional spatio-temporal patterns of neuronal activity, referred to as the reservoir/liquid state:

$$\mathbf{x}^{\mathbf{M}}(t) := (L^M \mathbf{u})(t)$$

The output $y(t) := W\mathbf{x}^{\mathbf{M}}(t)$ is then generated by a memory-less readout map W that processes the reservoir state. A simple machine learning algorithm or a biologically plausible learning rule like the ones seen in section 1.2 can be used to train the readout map.

Biological Characteristics of the Liquid

As birefly mentioned above, the liquid is designed to employ several biological principles, including:

- *Spiking Neurons:* The units within the LSM reservoir are typically modeled using biologically plausible spiking neuron models, such as the continuous-time leaky integrate-and-fire (LIF) neuron model [37]. This model is particularly common in theoretical neuroscience due its simplicity and ability to capture the essential feature of neurons (i.e. the integration of incoming signals and the generation of output spikes when a threshold is reached).
- *Excitatory and Inhibitory Neurons:* The LSM reservoir incorporates both excitatory and inhibitory neurons, reflecting the fundamental organization of biological neural networks according to Dale's law. Typically the ratio of excitatory to inhibitory neurons is set to 4:1, which is a common ratio found in biological neural networks.
- *Biologically Constrained Connectivity* The topology and connectivity in the LSM reservoir are designed to adhere to constraints observed in biological neural networks. For instance, the probability of connection between two neurons may depend on their spatial distance [37].

4.2 Echo State Networks

Echo State Networks (ESNs) are another prominent type of Reservoir Computing model that emerged independently of Liquid State Machines (LSMs) around the same time. While both ESNs and LSMs share the core principles of using a fixed RNN as a reservoir and training only a readout layer, they differ in their specific reservoir implementation.

In contrast to the spiking neural networks typically used in LSMs which operate in continuous time and use spike sequences as inputs, ESNs employ reservoirs composed of discrete-time artificial neurons, often with sigmoid-type activation functions [30]. The dynamics of the neuronal states in an ESN reservoir are typically described as:

$$\mathbf{x}(n+1) = f(W^{in}\mathbf{u}(n+1) + W\mathbf{x}(n) + W^{back}y(n)) \tag{4.1}$$

where n is the (discrete) time, $x(n)$ is the state vector of the reservoir units, $u(n)$ is the input vector, W^{in} is the input-to-reservoir weight matrix, W is the fixed recurrent weight matrix within the reservoir, W^{back} is the output-to-reservoir weight matrix, and f is the activation function. The output $y(n)$ is often a linear combination of the reservoir states:

$$y(n) = W^{out}\mathbf{x}(n) \tag{4.2}$$

where W^{out} is the trained readout weight matrix.

Part II: Multi-Layer Chain Models

Chapter 5

Introduction and Motivation

5.1 Background

Before describing the multi-layer chain architecture, we first provide a brief overview of two key results crucial to the development of the model.

Extension of Perceptron capacity

The capacity of the perceptron has been extensively studied, with the seminal work of Cover [9] establishing the conditions under which a perceptron can separate data.
In this context the measure of complexity is taken to be the maximum number of patterns which can be classified correctly by a perceptron. Through the use of Cover's Counting Theorem, it can be shown that for a perceptron with N inputs, the probability of being able to separate P with randomly assigned labels is given by[1]:

$$\mathbb{P}\left(\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_P, y_P)\} \text{ linearly separable}\right) = \frac{1}{2^{P-1}} \sum_{k=0}^{N-1} \binom{P-1}{k} \tag{5.1}$$

In the large N limit, this probability approaches 1 when $P < 2N$ and approaches 0 when $P > 2N$. We thus conclude the capacity of the perceptron to be $2N$.

Surprisingly, empirical observations [11] have shown that passing the data through a

[1] It's worth remarking that we must also assume the P points to be in general position, meaning that no subset of size less than N is linearly dependent. In practice this is a very mild condition and almost never happens.

Random RNN (properly constructed so as to make the dynamics stable and non-trivial) as a preprocessing step can increase the capacity of the perceptron. This is due to the fact that the network creates an internal representation of the data which better reveal the underlying structure of the data, thus allowing for the perceptron to separate the data more easily. In practice this preprocessing step is executed by setting the initial state of the RNN to be the input data, and then running the dynamics until convergence. Through this process, the capacity of the perceptron increases from $2N$ to $\sim 6N$.

Stabilization effects of Ferromagnetic Couplings

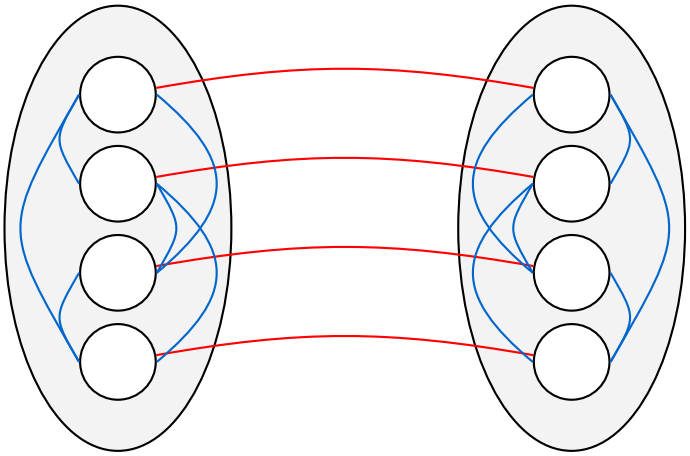


Figure 5.1: RNNs in parallel with spatial ferromagnetic connections

In section 3.1 we have seen that some methods that can be used to stabilize the dynamics of a random RNN. In particular, those methods relied on the presence of a self-coupling term J_D which acted as an inertial force that prevented each neuron from changing its state too frequently.

Borrowing a term from statistical physics, we call a "ferromagnetic" interaction a positive interaction between neurons which encourages them to have the same sign of activity. In this case, the self-coupling can be seen as a ferromagnetic connection in time between a neuron at time t and itself at time $t-1$.

The positive effect that ferromagnetic coupling have on the dynamics is not only restricted to temporal connections, but can also be extended to spatial connections. Indeed research indicates that having multiple RNNs connected in parallel, with each neuron in each RNN not only being fully connected to all the other neurons in the same RNN but also being connected to other neurons with the same index in other RNNs, leads to a stabilization of the dynamics comparable to that of the self-coupling term.

5.2 Overview of Multi-Layer Chain Architecture

Following the above remarks, we can construct a multi-layered architecture composed of multiple Random RNNs connected to each other sequentially through ferromagnetic connections. The idea would be to connect the input data to the first network layer, which would then pass along information to the next layers and so on, until the final layer which would then be connected to a classifier/regressor head that would output the prediction. We call such an architecture a Multi-Layer Chain Model (MLCM).

Just as Multi-Layer Perceptron networks extract more abstract features from the data the deeper into the network we go, the internal representation of the data in the MLCM is expected to become more and more instructive as we go deeper into the network.

Just like other architectures, MLCMs can also be trained to allow them to attune better to the data and specialize to the current task; crucially, this can be accomplished via local learning rules. Indeed, we will see in later sections that each neuron in each RNN layer can be thought of as a separate instance where to run a modified version of the perceptron learning algorithm.

During inference, the input data is passed to the first layer as an external input of a specified strength, then the dynamics of all the networks are made to run until convergence. The fixed point of the final layer is taken to be the internal representation of the data, which is then passed to the classifier head for the prediction.

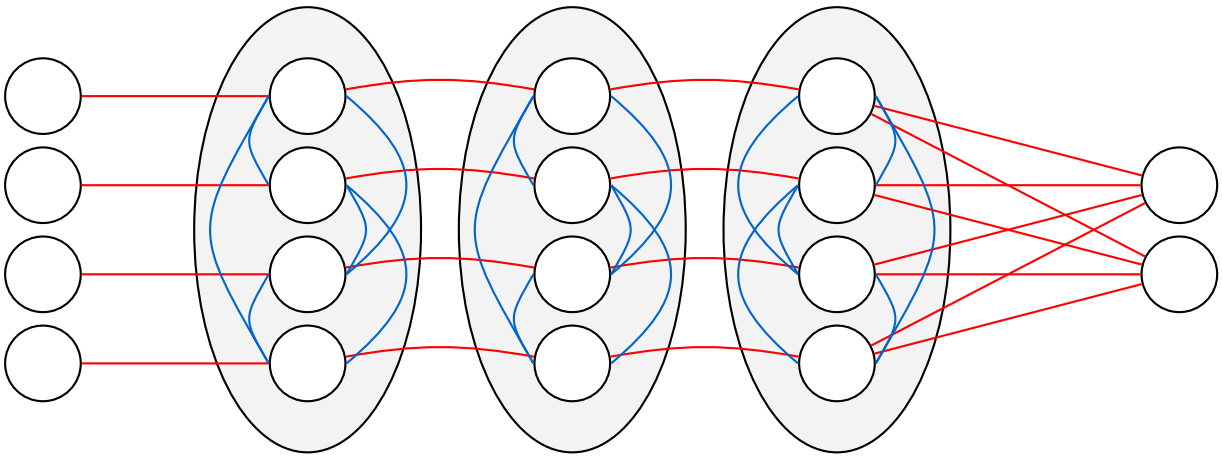


Figure 5.2: Diagram of MLCM architecture. The input data is passed through the network and ends up at the output layer, which is connected to a classifier head

Chapter 6

Mathematical Formulation

6.1 Model Framework

In the last chapter we introduced the general architecture behind MLCMs, we now proceed with providing a more formal definition of the model and its components.
Let's consider the simple case in which an input of size N is passed through L layers of RNNs, each of size N, and then through a classifier head of size M.

The model is composed of:

- *Layer Weights:* A a set of trainable randomly-initialized weights for each RNN: $\{J^l \mid J^l \in \mathbb{R}^{N\times N}\}_{l=1}^{L}$
- *Self Couplings:* A set of self-coupling weights: $\{J_D^l \mid J_D^l \in \mathbb{R}_+\}_{l=1}^{L}$
- *Left Ferromagnetic Couplings:* A set of ferromagnetic connections connecting the previous layer $l-1$ to the current layer l: $\{\lambda_{\mathrm{L}}^l \mid \lambda_{\mathrm{L}}^1 \in \mathbb{R}_+\}_{l=2}^{L}$
- *Right Ferromagnetic Couplings:* A set of ferromagnetic connections connecting the next layer $l+1$ to the current layer l: $\{\lambda_{\mathrm{R}}^l \mid \lambda_{\mathrm{R}}^L \in \mathbb{R}_+\}_{l=1}^{L-1}$
- *Input Coupling:* A value determining the strength that the input data applies to the first layer in the form of an external field: $\lambda_{\mathrm{IN}} \in \mathbb{R}$
- *Output Coupling:* A value determining the strength that the output applies to the last layer in the form of an external field: $\lambda_{\mathrm{OUT}} \in \mathbb{R}$
- *Forward Output Weight:* The weights of the perceptron predicting the ouput from the final RNN layer: $W^{\mathrm{fwd}} \in \mathbb{R}^{M\times N}$.

- *Backward Output Weight:* A weight matrix connecting the output layer to the last RNN layer and determining the influence it has on it: $W^{\text{back}} \in \mathbb{R}^{N\times M}$.

We want the model to be able to take an input data vector $\mathbf{x} \in \mathbb{R}^N$ and pass it through the network to obtain an output vector $\mathbf{y} \in \mathbb{R}^M$. It stands to reason that also $\mathbf{y}$, not only $\mathbf{x}$, should influence the learning of the weights in the network during training. We implement this by projecting the output layer up to N dimensions through W^{back} and then allowing it to influence the last RNN layer in a similar way to how the input data influences the first RNN layer. The flow of information in MLCMs is thus bidirectional, with the information about the input flowing forward and the information about the output flowing backward simultaneously.
Mathematically, we update the state of a neurons in layer l at time t according to the following dynamics:

$$s_i^l(t{+}1) = \begin{cases} \text{sign}\left(\sum_{j\neq i} J_{ij}^l s_j^l(t) + J_D^l\, s_i^l(t) + \lambda_{\text{IN}}\, x_i + \lambda_{\text{R}}^l\, s_i^{l+1}(t)\right) & \text{if } l = 1 \\ \text{sign}\left(\sum_{j\neq i} J_{ij}^l s_j^l(t) + J_D^l\, s_i^l(t) + \lambda_{\text{L}}^l\, s_i^{l-1}(t) + \lambda_{\text{R}}^l\, s_i^{l+1}(t)\right) & \text{if } 1 < l < L \\ \text{sign}\left(\sum_{j\neq i} J_{ij}^l s_j^l(t) + J_D^l\, s_i^l(t) + \lambda_{\text{L}}^l\, s_i^{l-1}(t) + \lambda_{\text{OUT}} \sum_{k=1}^{M} W_{ik}^{\text{back}} y_k\right) & \text{if } l = L \end{cases} \tag{6.1}$$

While a prediction is made by the readout layer as:

$$\hat{\mathbf{y}} = \text{softmax}\left(W^{\text{fwd}} \mathbf{s}_\infty^L\right) \tag{6.2}$$

Where $\mathbf{s}_\infty^L := \lim_{t\to\infty} \mathbf{s}^L(t)$ is the fixed point the last RNN layer converges to.

6.2 Learning Dynamics

Learning in MLCMs concerns updating the layer weights J^l and the output weights W^{fwd}, this is due to the fact that changing the other parameters either affect the stability of the dynamics or the flow of information upon which learning is based on for the other hyperparameters. They are best thought of as intrinsic properties of the network and, while it is possible to have them vary in time, they are usually treated as constant hyperparameters.

At each iteration, the training protocol is divided in two steps:

1. *Convergence Step:* In this step both $\mathbf{x}$ and $\mathbf{y}$ are passed through the network and the dynamics are left to run until convergence. As long as the ferromagnetic and the self-coupling terms are set to reasonable values, the network dynamics are sure to converge to some fixed points $\{\mathbf{s}_\infty^1, ..., \mathbf{s}_\infty^L\}$.
2. *Update Step:* In this step a single pass of the perceptron training rule updates the layer weights J^l, while the output weights W^{fwd} are updated through the cross-entropy loss.

A full pseudo-code implementation of the training algorithm is present in appendix A.

6.2.1 Convergence Step

The convergence step is computed through the dynamics described in 6.1. This step is crucial to the training of the MLCM, as it allows the network to reach an internal representation containing information about both the data and the label.
Similarly to discussions in chapter 3, there are multiple ways to update the states:

- *Layer-Independent Asynchronous Update:* Select a random neuron in the network and update its state using the current value of t for that neuron. Continue until all neurons in the entire MLCM have been updated.
- *Layer-Sequential Asynchronous Update:* Update each layer using asynchronous updating, but do so going from layer 1 to layer L sequentially.
- *Layer-Sequential Synchronous Update:* Update every neuron in a layer at the same time, using the current value of t for each neuron in that layer. Then proceed to the next layer and repeat until all layers have been updated.
- *Layer-Independent Synchronous Update:* Update all neurons in the network at the same time, using the current value of t for each neuron.

In practice, the last two methods are the most commonly used because they parallelize the computations through matrix multiplication and are thus much faster.

Usually at each iteration of training, the hidden layer states are initialized to a fixed initial state. This is done both to reduce the level of noise in the network as well as to allow for batching during training, which will be discussed later.

6.2.2 Update Step

During inference, the model will not have access to the true label $\mathbf{y}$ of $\mathbf{x}$, thus it would be preferable for the training to be done in such a way so that the outcome does not depend directly on the backward flow of information. To this end, at each iteration of training we can modify the layer weights J^l so as to try to better approximate the results obtained during the convergence step only through the forward flow.

In other words, after reaching the convergence step consider a neuron i in layer l: we can see it as a perceptron which as inputs has the other neurons in the same layer and neuron i in layer $l-1$, as pre-activation signal has: $\sum_{j\neq i} J^l_{ij} \left(\mathbf{s}^l_\infty\right)_j + \lambda^l_{\mathrm{L}} \left(\mathbf{s}^{l-1}_\infty\right)_i$ and as output we want to predict has $\left(\mathbf{s}^l_\infty\right)_i$. We can then apply the perceptron learning rule to update the weights J^l_{ij} with learning rate $\eta > 0$ as follows:

$$J^l_{ij} \leftarrow J^l_{ij} + \eta \left(\left(\mathbf{s}^l_\infty\right)_i - \operatorname{sign}\left(\sum_{j\neq i} J^l_{ij} \left(\mathbf{s}^l_\infty\right)_j + \lambda^l_{\mathrm{L}} \left(\mathbf{s}^{l-1}_\infty\right)_i \right) \right) \left(\mathbf{s}^l_\infty\right)_j \tag{6.3}$$

Unlike equation 6.1, this update rule actively ignores the contributions from both itself and $\left(\mathbf{s}^{l+1}_\infty\right)_i$ precisely becauuse we want we want the model at inference to only depend on the influence of the input.

The output weights W^{fwd} are more straightforward to train, they can either be trained through the perceptron learning rule or using cross-entropy loss. Either way, at each iteration we check whether the output $\mathbf{y}$ is predictable from the final layer states $\mathbf{s}^L_\infty$ and make a single sweep of the updates to the weights.

Both approaches are valid, but the cross-entropy loss has been observed to converge more quickly in practice.

Chapter 7

Empirical Evaluation

In the preceding chapters, we have established the theoretical underpinnings and architectural principles of Multi-Layer Chain Models.

This chapter presents a systematic experimental assessment of MLCMs, focusing on their predictive performance and empirical feasibility on classic datasets such as MNIST. Additionally, we seek to explore whether the MLCM architecture provides meaningful advantages over simple baseline models, such as single-layer perceptrons, in realistic settings.

Given that MLCMs are a novel approach whose behavior is not yet fully characterized, our primary goal is not to achieve state-of-the-art performance, but rather to conduct a series of exploratory experiments that test the basic feasibility and potential of the architecture.

To this end, we outline two main experimental protocols. The first evaluates baseline MLCM performance on two key datasets, MNIST and Fashion MNIST, along with two more challenging variants. Results are compared to those from simple perceptron classifiers trained on the same data, highlighting any practical advantages or limitations. The second experiment explores a modification of the MLCM architecture that enforces neural sparsity by switching to binary $\{0, 1\}$ states, using a fixed threshold and a Winner-Takes-All mechanism. This setup assesses how sparsity and competitive activation affect the model's empirical performance.

We will be focusing specifically on classification tasks with binary inputs, as they lend themselves best to the current MLCM formulation. However, this constraint can easily be relaxed in future work by making minor modifications to the architecture.

Through these exploratory studies, we aim to provide an initial foundation for understanding the empirical properties of MLCMs and to identify promising directions for future research and architectural refinement.

7.1 Baseline Implementation with Classical datasets

7.1.1 Experimental Setup

In this first experiment, we evaluate the basic empirical feasibility of MLCMs on the multiple basic datasets: Binary MNIST (either projected up or projected down) and Binary Fashion MNIST (either projected up and projected down). MNIST is a widely used dataset consisting of 28×28 grayscale images of handwritten digits (0-9), with 60,000 samples for training and 10,000 for testing. The task is to correctly classify each image into one of 10 classes. Fashion MNIST is a similar dataset composed of 28×28 grayscale images of clothing items, also with 10 classes.

When talking about projected up MNIST or Fashion MNIST we mean that we pass them through a fixed linear transformation that scales them up to a larger dimension[1], usually the dimension of the layer N. Since we are projecting into a higher dimension, we are not actually losing any information in the data but rather just rescaling the data to fit our model better. We thus refer to this case simply as MNIST or Fashion MNIST in any future references.

When we refer to projected down MNIST and fashion MNIST, we mean that the images are passed through a fixed linear projection that scales them down to a smaller dimension[2], 100 in our case. This lossy transformation makes the dataset more complicated to learn and acts as a more challenging test for the model. We employ a binarized version of the datasets, where each pixel is either black (denoted -1) or white (denoted 1).

Model Architecture

The MLCM employed in this experiment consists of 5 layers of random recurrent neural networks, each of size $N = 1000$. Since there is a mismatch between the input and the RNN size, we pad the inputs with zeroes so as to get the the input dimension d up to N.

[1] As long as the projection W is fixed for each input it doesn't matter too much what its values actually are

[2] See Footnote 1

To favour the forward flow of information over the backward one, we set the left ferromagnetic strength to be higher than the right one. The main hyperparameters are: $\lambda_L^2, \lambda_L^3, \lambda_L^4, \lambda_L^5 = 4.0, 4.0, 4.0, 4.0$ and $\lambda_R^1, \lambda_R^2, \lambda_R^3, \lambda_R^4 = 2.0, 2.0, 2.0, 2.0$, then $J_D^l = 0.1\ \forall l$, $\lambda_{\text{IN}} = 6.0$ and $\lambda_{\text{OUT}} = 12.0$.

Evaluation

Performance is measured by classification accuracy on the held-out MNIST test set. We report the average accuracy over 10 random initializations of the MLCM weights to assess robustness. Training and evaluation accuracies are recoded at each epoch for the entire training duration of 20 epochs.

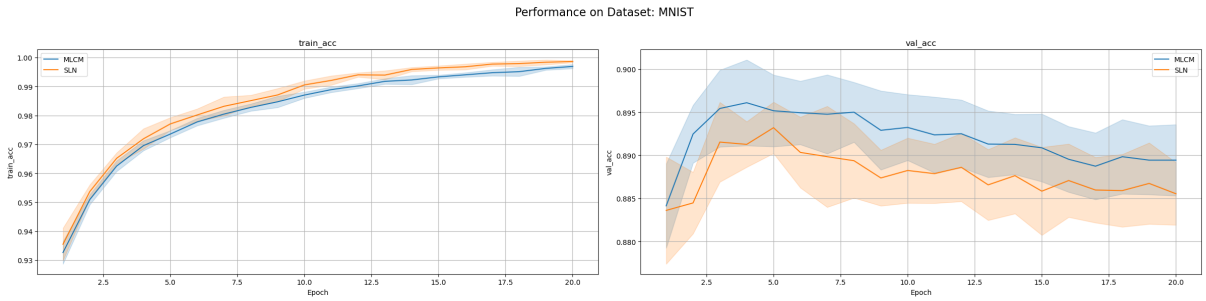
Baseline Comparison

To contextualize the results, we compare the accuracy of the MLCM-based classifier to that of a simple single-layer perceptron trained directly on the raw input vectors, using the same number of training epochs and analogous learning rules.

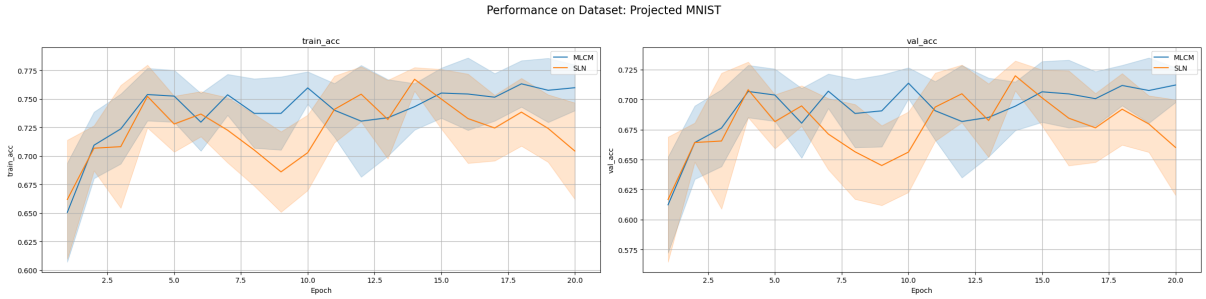
This experimental setup allows us to assess whether MLCMs can successfully learn meaningful internal representations on a standard machine learning benchmark, and provides a foundation for more detailed investigations in subsequent experiments.

7.1.2 Results

We report in Fig. 7.1 the results for the MNIST experiments and in Fig. 7.2 those for Fashion MNIST. Overall, MLCM outperforms the perceptron model, with a notable 5% validation accuracy advantage on projected MNIST, a significant margin for this task. While both models reach similar training accuracy on the simpler base datasets, MLCMs achieve much higher validation accuracy, indicating that their internal representations better highlight important features and reduce overfitting. These initial results suggest that MLCM-based preprocessing positively impacts perceptron performance and that the architecture is capable of learning meaningful internal representations of the data.

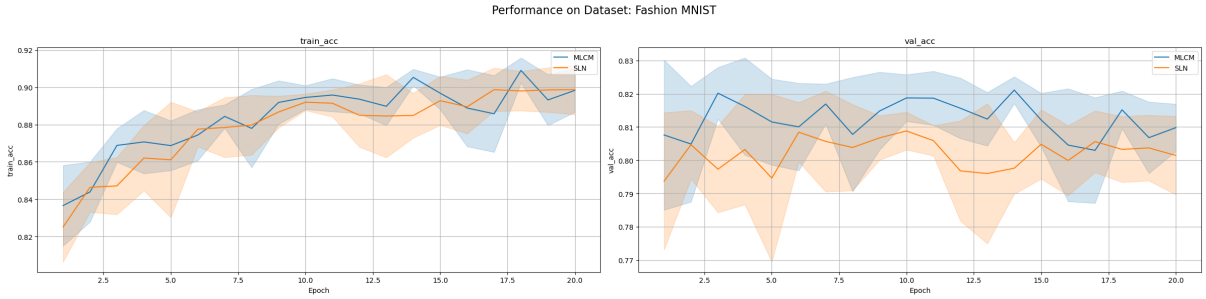


(a)

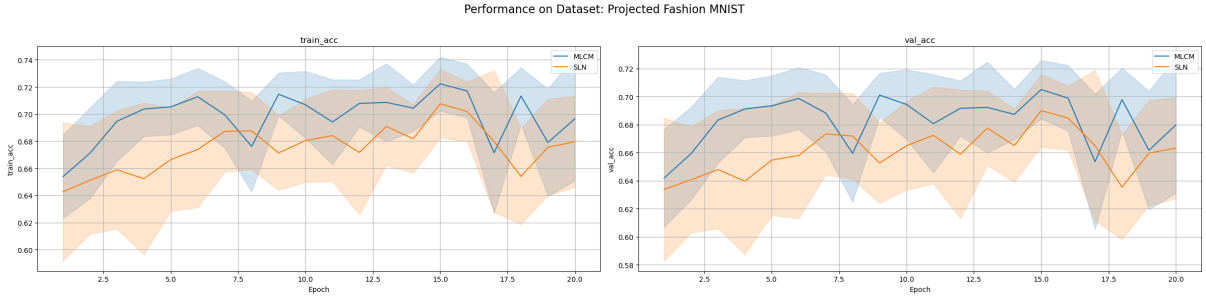


(b)

Figure 7.1: Comparison of testing and validation accuracy per epoch, reported with their 95% confidence interval, for MLCM and the Perceptron (Single Layer Network - SLN) on MNIST (Fig. 7.1a) and Projected MNIST (Fig. 7.1b).



(a)



(b)

Figure 7.2: Comparison of testing and validation accuracy per epoch, reported with their 95% confidence interval, for MLCM and the Perceptron (Single Layer Network - SLN) on Fashion MNIST (Fig. 7.2a) and Projected Fashion MNIST (Fig. 7.2b).

7.2 Sparsity Enforcement in MLCMs

The reliance on $s_i \in \{-1, 1\}$ states in the standard formulations of perceptrons and MLCMs is not biologically realistic, as biological neurons are typically either silent or active. A better framework to model biological neurons is to use binary states which take values in $z_i \in \{0, 1\}$, where 0 represents a silent neuron and 1 an active one.

We can express the $\{-1, 1\}$ states as the $\{0, 1\}$ states through: $s_i^l(t) = 2\, z_i^l(t) - 1$. Then equation 6.1 can be rewritten as:

$$\begin{aligned} z_i^l(t+1) &= \frac{1}{2}\left(\operatorname{sign}\left(\sum_{j\neq i} J_{ij}^l (2\, z_j^l(t) - 1) + J_D\,(2\, z_i^l(t) - 1) + \lambda_{\mathrm{L}}^l\,(2\, z_i^{l-1}(t) - 1) + \lambda_{\mathrm{R}}^l\,(2\, z_i^{l+1}(t) - 1) \right) + 1 \right) \\ &= \boldsymbol{\theta}\left(2\left(\sum_{j\neq i} J_{ij}^l z_j^l(t) + J_D\, z_i^l(t) + \lambda_{\mathrm{L}}^l\, z_i^{l-1}(t) + \lambda_{\mathrm{R}}^l\, z_i^{l+1}(t) \right) - \left(\sum_{j\neq i} J_{ij}^l + J_D + \lambda_{\mathrm{L}}^l + \lambda_{\mathrm{R}}^l \right) \right) \\ &= \boldsymbol{\theta}\left(\sum_{j\neq i} J_{ij}^l z_j^l(t) + J_D\, z_i^l(t) + \lambda_{\mathrm{L}}^l\, z_i^{l-1}(t) + \lambda_{\mathrm{R}}^l\, z_i^{l+1}(t) - \frac{1}{2}\left(\sum_{j\neq i} J_{ij}^l + J_D + \lambda_{\mathrm{L}}^l + \lambda_{\mathrm{R}}^l \right) \right) \\ &= \boldsymbol{\theta}\left(\sum_{j\neq i} J_{ij}^l z_j^l(t) + J_D\, z_i^l(t) + \lambda_{\mathrm{L}}^l\, z_i^{l-1}(t) + \lambda_{\mathrm{R}}^l\, z_i^{l+1}(t) - \kappa_i^l \right) \end{aligned}$$

Where $\boldsymbol{\theta}(x) = \mathbb{1}_{x\geq 0} = \frac{1}{2}\,(\operatorname{sign}(x) + 1)$ is the Heaviside function and

$$\kappa_i^l = \frac{1}{2}\left(\sum_{j\neq i} J_{ij}^l + J_D + \lambda_{\mathrm{L}}^l + \lambda_{\mathrm{R}}^l \right)$$

is a constant threshold resulting from the conversion from $\{-1, 1\}$ to $\{0, 1\}$ states. With a similar computation you can obtain similar results in the $l = 1$ and $l = L$ cases, obtaining:

$$\kappa_i^1 = \frac{1}{2}\left(\sum_{j\neq i} J_{ij}^l + J_D + \lambda_{\mathrm{IN}}^l + \lambda_{\mathrm{R}}^l \right), \qquad \kappa_i^L = \frac{1}{2}\left(\sum_{j\neq i} J_{ij}^l + J_D + \lambda_{\mathrm{L}}^l \right)$$

The output $\mathbf{y}$ is not transformed from $\{-1, 1\}$ to $\{0, 1\}$, while the input $\mathbf{x}$ is, thus κ_i^L does not include a $\lambda_{\mathrm{OUT}} \sum_{k=1}^{M} W_{ik}^{\mathrm{back}}$ that would at first glance be expected.

Switching from $z_i^l(t)$ back to $s_i^l(t)$, the dynamics of the MLCM in $\{0, 1\}$ case are expressed

in the as:

$$s_i^l(t{+}1) = \begin{cases} \boldsymbol{\theta}\left(\sum_{j\neq i} J_{ij}^l s_j^l(t) + J_D^l\, s_i^l(t) + \lambda_{\text{IN}}\, x_i + \lambda_{\text{R}}^l\, s_i^{l+1}(t) - \kappa_i^l - \boldsymbol{\gamma}\right) & \text{if } l = 1 \\ \boldsymbol{\theta}\left(\sum_{j\neq i} J_{ij}^l s_j^l(t) + J_D^l\, s_i^l(t) + \lambda_{\text{L}}^l\, s_i^{l-1}(t) + \lambda_{\text{R}}^l\, s_i^{l+1}(t) - \kappa_i^l - \boldsymbol{\gamma}\right) & \text{if } 1 < l < L \\ \boldsymbol{\theta}\left(\sum_{j\neq i} J_{ij}^l s_j^l(t) + J_D^l\, s_i^l(t) + \lambda_{\text{L}}^l\, s_i^{l-1}(t) + \frac{\lambda_{\text{OUT}}}{2}\sum_{k=1}^{M} W_{ik}^{\text{back}} y_k - \kappa_i^l - \boldsymbol{\gamma}\right) & \text{if } l = L \end{cases} \tag{7.1}$$

Where the extra term included, $\boldsymbol{\gamma}$, is a threshold function that regulates the activation of the states.

This change is not only attractive for the above reason, but also because through the inclusion of an inactive state we can work to enforce sparsity on the network. Human brains are known to be highly activation sparse, with only a small fraction of neurons firing at any given time. This sparsity is thought to be crucial for efficient information processing and sparse representations have been hypothesized to be key in encoding high-level features and concepts in the brain [22].
Sparsity also has practical benefits in terms of computational efficiency, as it reduces the number of non-trivial computations required to run the dynamics.

In mathematical terms, we say that layer l is has a sparsity of δ^l at time t if:

$$\frac{1}{N}\sum_{i=0}^{N} s_i^l(t) = \delta^l$$

Through the modification of the threshold function γ we can regulate the sparsity of the network. Bigger threshold values will lead to less neurons being activated, while smaller threshold values will lead to more neurons being activated.

In this section, we examine two strategies for setting the threshold γ: using a fixed threshold value for all neurons, and employing a Winner-Takes-All (WTA) mechanism, where only the most strongly activated neurons become active.

7.2.1 Fixed Threshold Sparsity

Experimental Setup

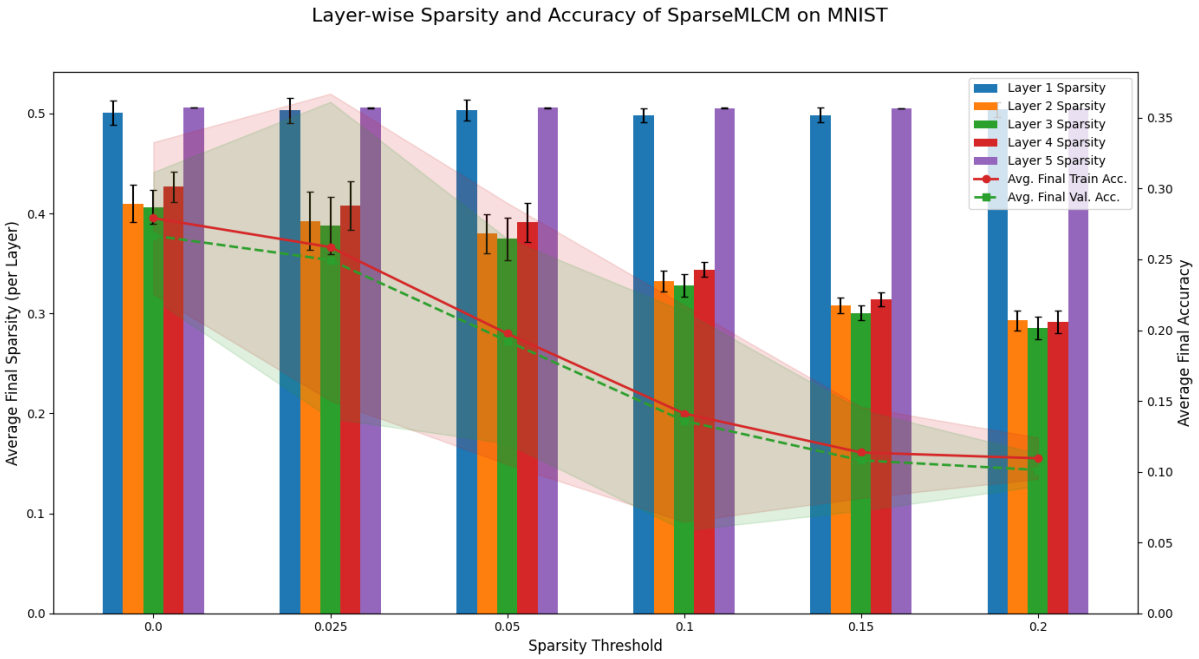
In this first experiment, we explore the impact of a fixed threshold value on the sparsity and the performance of the MLCM. The threshold function is set to a constant value $\boldsymbol{\gamma} = \gamma \in [0, +\infty]$ across all neurons. The focus of this experiment is not necessarily on achieving the best performance, but rather gain a better understanding of the trade-off between compression of the network through sparsity and the predictive capability of the network. For this reason we will be solely working on a single dataset, MNIST.

We will be using the same MLCM architecture as in the previous experiment, constructing one model for each value of γ in the set $\{0.0, 0.025, 0.05, 0.1, 0.15, 0.2\}$. The remaining hyperparameters are kept the same as in the previous experiment.
For each run we measure the training and validation accuracy at each epoch, as well as the sparsity for each layer of the network. Each model is trained for 25 epochs and for robustness of the results, we will be averaging the results over 5 random initializations of the MLCM weights.

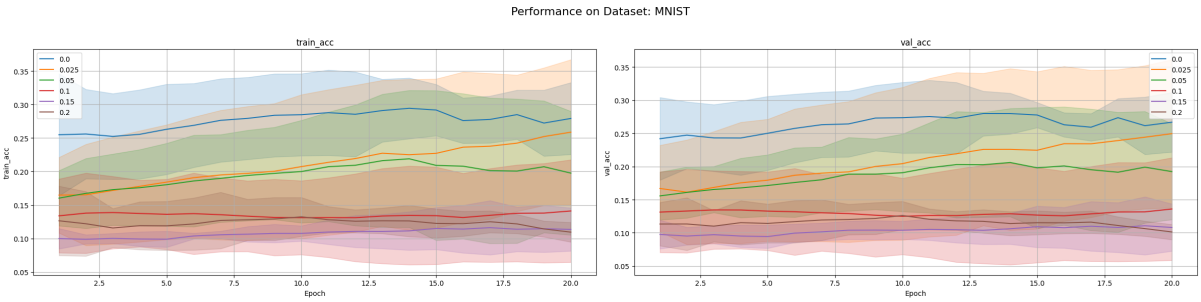
Results

The results of the experiment are presented in Fig. 7.3. As can be seen from 7.3a, the sparsity of the middle layers of the network decreases by $\sim 10\%$ as γ increases from 0 to 0.2. The first and the last layer maintain a constant sparsity level of ~ 0.5, this is due to the strong influence they receive from the non-sparse input and output [3]. The fixed states of these layers thus reflect this influence and are less affected by the sparsity threshold.
Another important observation is the presence of a sharp drop in the model's learning abilities as γ increases, this is due to the fact that forcing too many neurons to be inactive at once leads to a loss of information in the network to the point where it is no longer able to create meaningful representations of the data. After the $\gamma = 0.1$ threshold (corresponding to a sparsity of $\sim 25\%$), the MLCM is not able to be trained and its performance resembles that of random classifer.
These results confirm the presence of a trade-off between sparsity and performance.

[3] Even though $\mathbf{y}$ is one-hot-encoded, the influence it has on the last layer is not sparse since it is passed through the matrix W^{back}



(a)



(b)

Figure 7.3: Fig. 7.3a displays the sparsity of each layer in the final epoch of training and the final training and validation accuracy for each value of γ, reported with a 95% confidence interval. Fig. 7.3b shows the training and validation accuracy per epoch for each value of γ, reported with a 95% confidence interval.

Specifically, while increased sparsity can yield greater biological plausibility and reduced computational cost, it also leads to a measurable decline in the network's accuracy and memory capacity, thus highlighting the importance of careful tuning of the sparsity level to balance these competing objectives.

7.2.2 Winner-Takes-All Sparsity

Experimental Setup

We now turn our attention to another approach for sparsity enforcement, denoted as Winner-Takes-All (WTA). In this method, the threshold function $\boldsymbol{\gamma}$ is set to a dynamic

value that depends on the activation of the neurons in the current layer. Specifically, we define a predetermined sparsity level δ^l for each layer l and then activate the $N\delta^l$ neurons with the highest pre-activation signal, while the rest are set to zero. In other words:

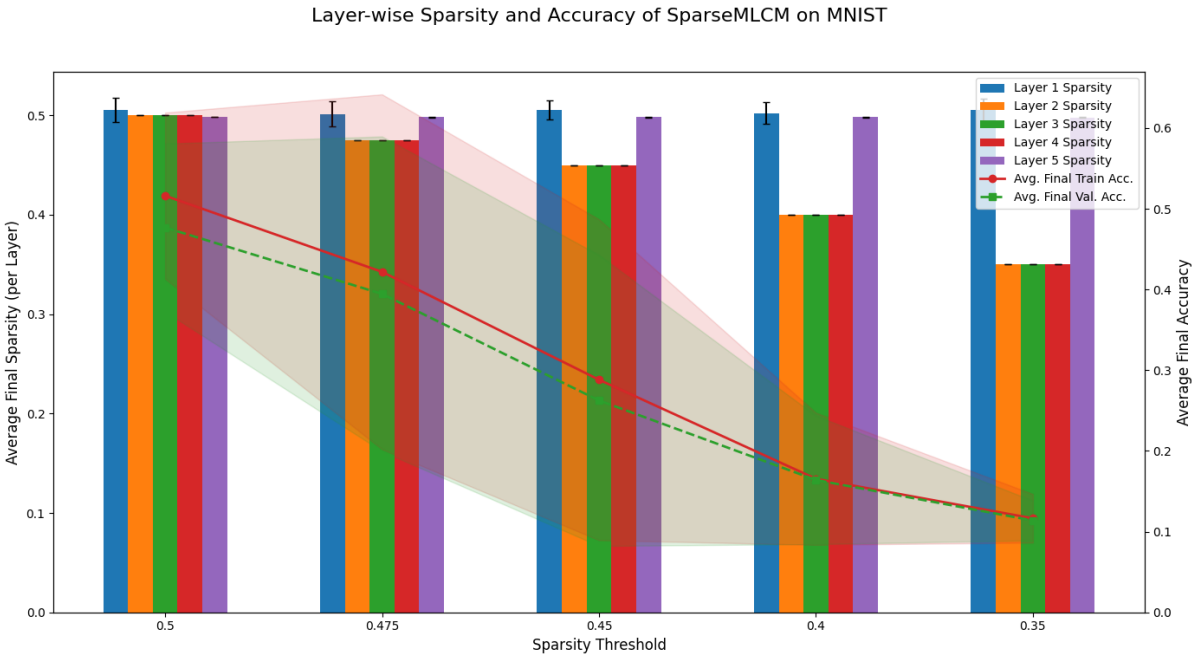
$$\boldsymbol{\gamma}(l) = Q\left(1 - \delta^l\right)$$

Where $Q\left(\delta^l\right)$ is the quantile function of $\left(\sum_{j \neq i} J_{ij}^l s_j^l(t) + J_D\, s_i^l(t) + \lambda_{\mathrm{L}}^l\, s_i^{l-1}(t) + \lambda_{\mathrm{R}}^l\, s_i^{l+1}(t) - \kappa_i^l\right)$. This approach allows us to dynamically adjust the threshold of each layer to maintain a fixed sparsity level, while still allowing for some neurons to be active at each time step. There are several regions of the brain which are thought to work similarly, and in theoretical neuroscience the WTA mechanism is widely used for modelling purposes [7, 28]. For this reason, we chose to experiment with WTA sparsity as a biologically motivated approach to enforcing activity constraints in our networks.

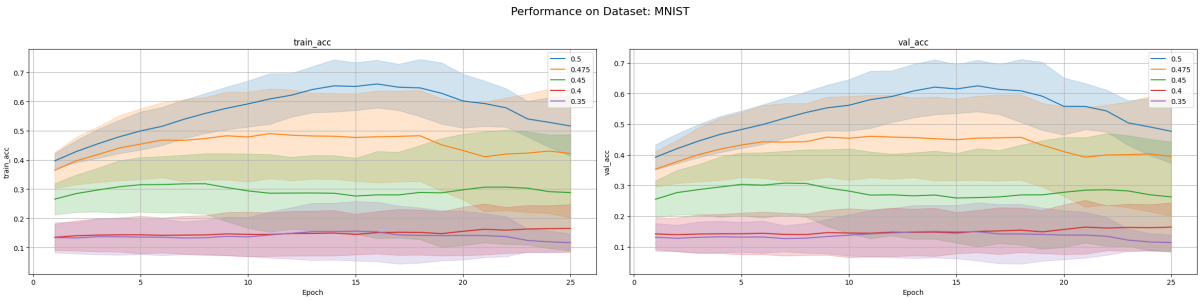
We employ the same experimental setup as in the previous tests: testing on MNIST, maintaining the same architecture and hyperparameters, training for 25 epochs and repeating each result 5 times. We fit one model for each value of δ in the set $\{0.5, 0.475, 0.45, 0.4, 0.35\}$. Due to influence of the input and the output on the first and last layer, we forgo a WTA-based sparsity implementation for them an instead revert back to a fixed threshold of $\gamma = 0$ to ensure that there is no loss of information. Thus the first layer and last layers will have no sparsity enforcement while the middle layers will all be constrained to a sparsity of δ.

Results

We report the results of the experiment in Fig. 7.4. The trade-off between sparsity and performance observed in the previous experiment persists here, with the performance dropping to near random levels as the δ reaches 0.4 sparsity.
Together with the results from the previous experiment, these findings suggest that there is not much room for compression of the network through sparsity without significantly degrading the performance, at least in the case of non-sparse inputs.
In particular, the WTA approach seems to be more effective when close to the baseline sparsity of 50%, however it also decays more quickly than the fixed threshold as the sparsity is increased.



(a)



(b)

Figure 7.4: Fig. 7.3a displays the sparsity of each layer in the final epoch of training and the final training and validation accuracy for each value of δ, reported with a 95% confidence interval. Fig. 7.3b shows the training and validation accuracy per epoch for each value of δ, reported with a 95% confidence interval.

An interesting phenomenon that can be seen in Fig. 7.4b is that after the 15-th epoch both the training and validation accuracies of the best performing models start decreasing. Upon further inspection, this "unlearning" phase coincides with an increase in the average iteration taken for the model to converge.

Further experiments with an increase in the batch size have helped mitigate this effect, though at the cost of a much longer training time, however further research is needed to properly understand the cause of this effect.

Conclusions and Future Directions

In this thesis, we explored a small subset of large interplay between biological and machine learning, with a particular focus on neural architectures and learning rules that reflect principles observed in the brain. Part I provided an overview of foundational models, including the perceptron, Hopfield networks, and random recurrent neural networks, emphasizing both their computational properties and biological relevance.
Part II introduced the Multi-Layer Chain Model (MLCM), a novel architecture inspired by biological mechanisms such as local learning and sparse activity. We presented the mathematical formulation of MLCMs and discussed their potential to generate meaningful internal representations through local, layer-wise interactions. Detailed empirical tests of MLCM's capabilities were performed, demonstrating the practical advantages that MLCMs can learn meaningful representations and achieve competitive performance on standard benchmarks.

Despite these promising results, several limitations remain. The recent nature of the MLCM architecture means there is still no rigorous theoretical framework, and its strengths and weaknesses are not fully known. For instance, the observed difference in performance between models using $\{0, 1\}$ and $\{-1, 1\}$ binary states is not yet well understood and warrants further investigation. Future research should focus on developing a deeper theoretical understanding of MLCMs, exploring more complex variants (such as with convolutional layers), and further investigating the role of sparsity in learning and generalization to further close the gap between biological plausibility and state-of-the-art machine learning.

Bibliography

[1] Daniel J. Amit. *Modeling Brain Function: The World of Attractor Neural Networks*. Cambridge University Press, September 1989.

[2] Daniel J. Amit, Hanoch Gutfreund, and H. Sompolinsky. Spin-glass models of neural networks. *Phys. Rev. A*, 32:1007–1018, Aug 1985.

[3] Daniel J. Amit, Hanoch Gutfreund, and H. Sompolinsky. Storing infinite numbers of patterns in a spin-glass model of neural networks. *Phys. Rev. Lett.*, 55:1530–1533, Sep 1985.

[4] Daniel J. Amit, Hanoch Gutfreund, and H. Sompolinsky. Information storage in neural networks with low levels of activity. *Phys. Rev. A*, 35:2293–2303, Mar 1987.

[5] David Attwell and Simon B. Laughlin. An energy budget for signaling in the grey matter of the brain. *Journal of Cerebral Blood Flow & Metabolism*, 21(10):1133–1145, 2001. PMID: 11598490.

[6] Pierre Baldi. *Deep Learning in Science*. Cambridge University Press, April 2021.

[7] Matteo Carandini and David J. Heeger. Normalization as a canonical neural computation. *Nature Reviews Neuroscience*, 13(1):51–62, November 2011.

[8] Zhang Chengxiang, Chandan Dasgupta, and Manoranjan P. Singh. Retrieval properties of a hopfield model with random asymmetric interactions. *Neural Computation*, 12(4):865–880, April 2000.

[9] Thomas M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Transactions on Electronic Computers*, EC-14(3):326–334, 1965.

[10] A. Crisanti and H. Sompolinsky. Dynamics of spin systems with randomly asymmetric bonds: Langevin dynamics and a spherical model. *Physical Review A*, 36(10):4922–4939, November 1987.

[11] Martina Del Gaudio. Bachelor's thesis based on unpublished work of R. Zecchina, M. Mezard, C. Baldassi, Bocconi University, 2024.

[12] B Derrida, E Gardner, and A Zippelius. An exactly solvable asymmetric neural network model. *Europhysics Letters (EPL)*, 4(2):167–173, July 1987.

[13] Ian D. Forsythe, Chunlai Wu, and J. Gerard G. Borst. Size matters: formation and function of giant synapses. *The Journal of Physiology*, 591(13):3123–3123, July 2013.

[14] Suzana Herculano-Houzel. Scaling of brain metabolism with a fixed energy budget per neuron: Implications for neuronal activity, plasticity and evolution. *PLOS ONE*, 6(3):1–9, 03 2011.

[15] J. A. Hertz, G. Grinstein, and S. A. Solla. Memory networks with asymmetric bonds. In *AIP Conference Proceedings*, volume 151, pages 212–218. AIP, 1986.

[16] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation*. CRC Press, March 2018.

[17] G. E. Hinton and T. Sejnowski. Optimal perceptual inference. pages 448–453, Washington, DC, June 1983. ieeecs.

[18] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.

[19] J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8):2554–2558, April 1982.

[20] J.J. Hopfield. Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences of the United States of America*, 81(10):3088–3092, 1984.

[21] Herbert Jaeger. The" echo state" approach to analysing and training recurrent neural networks-with an erratum note'. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, 148, 01 2001.

[22] Iiro P. Jääskeläinen, Enrico Glerean, Vasily Klucharev, Anna Shestakova, and Jyrki Ahveninen. Do sparse brain activity patterns underlie human cognition? *NeuroImage*, 263:119633, 2022.

[23] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[24] W.A. Little. The existence of persistent states in the brain. *Mathematical Biosciences*, 19(1):101–120, 1974.

[25] Wolfgang Maass. *Liquid State Machines: Motivation, Theory, and Applications*, pages 275–296. 2011.

[26] Wolfgang Maass and Henry Markram. On the computational power of circuits of spiking neurons. *Journal of Computer and System Sciences*, 69(4):593–616, 2004.

[27] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):25312560, November 2002.

[28] Svenja Marx, Gina Gruenhage, Daniel Walper, Ueli Rutishauser, and Wolfgang Einhäuser. Competition with and without priority control: linking rivalry to attention through winner-take-all networks with memory. *Annals of the New York Academy of Sciences*, 1339(1):138–153, January 2015.

[29] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, December 1943.

[30] Luca Melandri. Introduction to reservoir computing methods. Master's thesis, Alma Mater Studiorum - Università di Bologna, 2014.

[31] G Parisi. Asymmetric neural networks and the process of learning. *Journal of Physics A: Mathematical and General*, 19(11):L675, aug 1986.

[32] P. Peretto. Collective properties of neural networks: A statistical physics approach. *Biological Cybernetics*, 50(1):51–62, February 1984.

[33] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[34] Mattia Scardecchia. Statistical physics of learning. Bachelor's thesis based on unpublished work of R. Zecchina, M. Mezard, C. Baldassi, Bocconi University, 2023.

[35] Benjamin Scholl, Connon I. Thomas, Melissa A. Ryan, Naomi Kamasawa, and David Fitzpatrick. Cortical response selectivity derives from strength in numbers of synapses. *Nature*, 590(7844):111–114, December 2020.

[36] H. Sompolinsky, A. Crisanti, and H. J. Sommers. Chaos in random neural networks. *Physical Review Letters*, 61(3):259–262, July 1988.

[37] Gouhei Tanaka, Toshiyuki Yamane, Jean Benoit Héroux, Ryosho Nakane, Naoki Kanazawa, Seiji Takeda, Hidetoshi Numata, Daiju Nakano, and Akira Hirose. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, 2019.

[38] Zhenghua Xu, Miao Yu, and Yuhang Song. Inspires effective alternatives to backpropagation: predictive coding helps understand and build learning. *Neural Regeneration Research*, 20(11):3215–3216, October 2024.

[39] Zong-Ben Xu, Guo-Qing Hu, and Chung-Ping Kwong. Asymmetric hopfield-type networks: Theory and applications. *Neural Networks*, 9(3):483–501, April 1996.

Appendix A

MLCM Training Algorithm

We present an advanced implementation of the MLCM training algorithm, which includes weight decay, robustness, and initialization methods. The algorithm is designed to train a multi-layer chain model (MLCM) using a batched input data and labels.

The algorithm makes use of two other algorithms: `ConversionDynamics` (alg. 2) and `UpdateSweep` (alg. 3). The former is responsible for running the dynamics of the MLCM until convergence, while the latter is responsible for updating the weights of the MLCM based on the converged states of the MLCM and the labels.

Algorithm 1 MLCM Training Algorithm

Require: Batched Input Data $\mathbf{x}_{batch}$, Labels $\mathbf{y}$, Layer Weights $\mathbf{W}$, Self-Coupling J_d, Left Strength λ_L, Right Strength λ_R, Max Iterations T_{max}, Learning Rate η, Weight Decay λ, Robustness ρ, Initialization Method *init_method*

// Initialization Step
$\mathbf{y}_{batch} \leftarrow \mathbf{y} \cdot \mathbf{W}^{back}$
if *init_method* = "input" **then**
    Initialize states $\mathbf{S}$ by repeating $\mathbf{x}_{batch}$ for $L+2$ layers
else if *init_method* = "zeros" **then**
    Initialize states $\mathbf{S}$ as zeros
end if
$\mathbf{S}^0 \leftarrow \mathbf{x}_{batch}$ ▷ Set input as 0-th layer
$\mathbf{S}^{L+1} \leftarrow \mathbf{y}_{batch}$ ▷ Set target's influence as (L+1)-th layer

// Convergence Step
$\mathbf{S}, \mathbf{iters} \leftarrow \text{ConversionDynamics}(\mathbf{S}, \mathbf{W}, J_d, \lambda_L, \lambda_R, T_{max})$

if *train* **then**
// Update Step
    $\mathbf{W}, \mathbf{W}^{fwd} \leftarrow \text{UpdateSweep}(\mathbf{S}, \mathbf{y}, \mathbf{W}, \lambda_L, \lambda_R, \eta, \xi, \rho, \mathbf{W}^{fwd}, \eta_{ro})$
else
// Output Calculation
    $\mathbf{S}_{last\_rnn} \leftarrow \mathbf{S}^L$ ▷ States of the last RNN layer
    $\mathbf{logits} \leftarrow \mathbf{W}^{fwd}\mathbf{S}^L$
    return logits
end if

Algorithm 2 ConversionDynamics

Require: States $\mathbf{S}$, Weights $\mathbf{W}$, Self-coupling J_d, Left strength λ_L, Right strength λ_R, Max iterations T_{max}

$t \leftarrow 0$
$\mathbf{W}_{temp} \leftarrow \mathbf{W}_{temp} + J_d \cdot \mathbf{I}$
while $t < T_{max}$ **do**
    $\mathbf{S}_{new} \leftarrow \mathbf{S}$

    for $l \leftarrow 1$ to L **do** ▷ Iterate through hidden layers
        $\mathbf{H}_{left} \leftarrow \mathbf{S}_{new}^{l-1} \odot \lambda_L^{l-1}$
        $\mathbf{H}_{main} \leftarrow \mathbf{W}_{temp}^{l-1} \mathbf{S}_{new}^{l}$
        if *train* **then**
            $\mathbf{H}_{right} \leftarrow \mathbf{S}_{new}^{l+1} \odot \lambda_R^{l-1}$
        else
            $\mathbf{H}_{right} \leftarrow \mathbf{0}$
        end if
        $\mathbf{Field} \leftarrow \mathbf{H}_{left} + \mathbf{H}_{main} + \mathbf{H}_{right}$
        $\mathbf{S}_{new}^{l} \leftarrow \text{sign}(\mathbf{Field})$
    end for

    $t \leftarrow t + 1$
    if $\mathbf{S} = \mathbf{S}_{new}$ **then** ▷ Check for overall convergence
        break
    end if
    $\mathbf{S} \leftarrow \mathbf{S}_{new}$
end while

return $\mathbf{S}$

Algorithm 3 UpdateSweep

Require: States $\mathbf{S}$, Target $\mathbf{y}$, Weights $\mathbf{W}$, Left strength λ_L, Right strength λ_R, Learning rate η, Weight decay ξ, Robustness ρ, Read-out weights $\mathbf{W}^{fwd}$, Read-out learning rate η_{ro}

// Hidden Layer Training
$\mathbf{S}_{fixed} \leftarrow \text{permute}(\mathbf{S})$ ▷ Shape: (batch_size, $L+2$, N)
$\mathbf{H}_{left} \leftarrow \mathbf{S}_{fixed}[:, 0 \text{ to } L-1] \odot \lambda_L$
$\mathbf{H}_{main} \leftarrow \text{einsum}(\text{``lji,bli->blj''}, \mathbf{W}, \mathbf{S}_{fixed}[:, 1 \text{ to } L])$
$\mathbf{InstabilityCond} \leftarrow ((\mathbf{H}_{left} + \mathbf{H}_{main}) \odot \mathbf{S}_{fixed}[:, 1 \text{ to } L]) \leq \rho$ ▷ Added robustness
$\Delta\mathbf{W} \leftarrow \text{einsum}(\text{``bln,blm->lnm''}, \mathbf{S}_{fixed}[:, 1 \text{ to } L] \odot \mathbf{InstabilityCond}, \mathbf{S}_{fixed}[:, 1 \text{ to } L])$
$\Delta\mathbf{W} \leftarrow \Delta\mathbf{W} \odot (\eta / \text{batch\_size})$
$\mathbf{W} \leftarrow \mathbf{W} \odot (1 - \eta \odot \xi) + \Delta\mathbf{W}$ ▷ Weight decay is included
Set diagonal of each $\mathbf{W}^l$ to 0 ▷ We set the self-couplings externally

// Read-out Layer Training through Cross-Entropy Loss
$\mathbf{S}_{last\_rnn} \leftarrow \mathbf{S}_{fixed}[:, L]$
$\mathbf{Logits} \leftarrow \mathbf{W}^{fwd} \mathbf{S}^{L}$
$\mathbf{Predictions} \leftarrow \text{softmax}(\mathbf{Logits})$
$\nabla\mathbf{Logits} \leftarrow \mathbf{Predictions} - \mathbf{y}$
$\nabla\mathbf{W}^{fwd} \leftarrow (\nabla\mathbf{Logits})^T \, \mathbf{S}_{last\_rnn}$
$\mathbf{W}^{fwd} \leftarrow \mathbf{W}^{fwd} - \eta_{ro} \cdot \nabla\mathbf{W}^{fwd}$

return $\mathbf{W}, \mathbf{W}^{fwd}$