POLITECNICO DI TORINO

Master's Thesis

Exploring the Robustness Spectrum of Neural Networks Through Enhanced Sampling Methods

Author:
Matteo VOGLIOLO

Supervisors:
Alfredo BRAUNSTEIN
Alessandro INGROSSO

A thesis submitted in fulfillment of the requirements for the Master's degree in

Physics of Complex Systems

Academic Year 2023/2024

Abstract

Neural networks are complex systems with huge computational power whose internal dynamics is still poorly understood. The intricate relation between the statistical properties of supervised tasks they are trained on and the geometry of the resulting internal representations is an open research problem. In particular, a coarse-grained description of internal representations seems fruitful in studying their computational and generalization capabilities.

In this work, we introduce a method to study the resilience of a network to neuronal death by employing the Wang Landau algorithm [4, 8], a non-markovian sampling method. While such method has been used to study coarse-graining of macromolecules [2], it is virtually unknown to the Machine Learning and Computational Neuroscience community.

Focusing on a simple data model that naturally induces a convolutional structure fully connected networks trained from scratch [3], we examine resilience to network attacks in relation to the amount of spatial localization of receptive fields in the fist layer. We further investigate the dynamics of the network's parameters and the features of the internal representations during training.

Contents

Chapter 1

Introduction

Neural networks are typically trained on a specific task using a given dataset, and their performance is evaluated through a measure called accuracy. In classification problems, for instance, accuracy is defined as the ratio of correct predictions to the total number of predictions.

When we refer to neuronal failure, we are describing a scenario in which, some neurons in the network stop functioning, meaning their output becomes zero. This phenomenon can impair the network performance and resilience to such disruptions becomes an important aspect to study. Network robustness can be described as the network's ability to maintain high accuracy on the task, even in the presence of neuronal failure.

Drawing an analogy with statistical physics, a neural network can be viewed as a physical system where each neuron exists in one of two possible states: "active" or "inactive." The global state of the network is then represented by a configuration where the activity of each neuron is specified. If the network consists of N neurons, its state can be described by an N-dimensional binary vector σ, with each component representing whether a neuron is active or inactive. In this framework, each configuration is associated with an energy, where low-energy configurations correspond to high-accuracy states.

One way to study network robustness is to examine the density of states of the system described above, a quantity that indicates how configurations are distributed across the available energy levels. A perfectly robust network would exhibit a density of states concentrated at the energy level corresponding to the configuration in which all neurons are active, as this would imply that neuronal failure does not compromise the network functioning. However, this is unrealistic; we expect the configurations to be distributed over a range of energy values between that of the fully active state and that of the fully inactive state. The more the distribution leans towards the former case, the more we consider the system robust.

Obtaining the density of states for a system is a complex computational challenge, particularly due to the need for comprehensive sampling of the configuration space. Traditional Monte Carlo methods, such as Metropolis-Hastings,

often face difficulties in efficiently exploring high-dimensional landscapes, leading to significant sampling biases and slow convergence. This can result in inaccurate estimates of how configurations are distributed across energy levels.

In order to overcome this issue, we employ the Wang-Landau algorithm, a sampling method specifically designed for estimating the density of states. This algorithm seeks to achieve a 'flat' histogram in energy, meaning it aims for a uniform sampling of all energy levels. By doing so, it enhances the likelihood of exploring configurations that might otherwise be neglected by conventional methods. The Wang-Landau algorithm operates in a non-Markovian manner, allowing updates based on the entire history of the sampling process. Despite its non-Markovian characteristics, the Wang-Landau method preserves detailed balance.

The present thesis is organized as follow. In this first chapter we provide a brief introduction to some fundamental concepts related to artificial neural networks and offer a concise overview of Monte Carlo methods, as these topics will recur throughout the project. The information on neural networks is primarily sourced from [7], while the introduction to Monte Carlo methods is based on the reference book [5].

In the second chapter, we analyze a feedforward multilayer perceptron (MLP) with a single hidden layer, focusing on the emergence of a convolutional structure in the weight vectors after training. We investigate the network's robustness, as well as the dynamics of its parameters and internal representations.

In the Methods and Materials chapter, we summarize the technical details regarding the tasks used to train the network, the statistics we tracked, and the functioning of the Wang-Landau algorithm, including our implementation.

The thesis concludes with a summary of our results and considerations. Additionally, we include an appendix that explores the conditions under which the aforementioned convolutional structure emerges.

1.1 Multi-layer Perceptron (MLP)

MLPs are a type of feedforward neural network where each layer is fully connected to the next. In addition to the input and output layers, MLPs consist of one or more intermediate layers, known as "hidden layers."

Each edge of the network is associated with a weight and each neuron has a bias, the set of all the weights and biases of the network $\{\mathbf{w}, \mathbf{b}\}$ are the parameters of our model.

The output of a neuron is calculated as the weighted sum of the outputs of the neurons from the previous layer, to which the bias is added, and the result is passed through an activation function σ. Thus we can define the output of the j-th neuron in layer l as:

$$a_j^l = \sigma(z_j^l) \qquad z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \tag{1.1}$$

Here, w_{jk}^l represents the weight of the connection between the k-th neuron in the

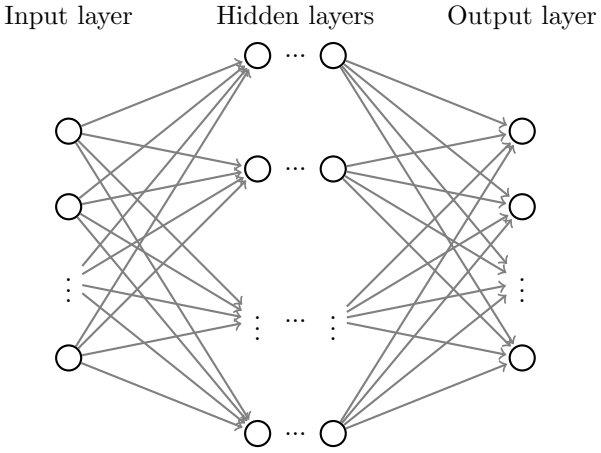
$(l-1)$-th layer and the j-th neuron in the l-th layer. Similarly b_j^l represents the bias of the j-th neuron in the l-th layer. The term z_j^l is called the pre-activation of the neuron.

We can express these equations in matrix form by defining a weight matrix w^l for each layer l with w_{jk}^l as entries of the matrix. Similarly, for each layer l, we define a bias vector b^l and an outputs vector a^l. Thus we have:

$$\mathbf{a}^l = \sigma(\mathbf{z}^l) \qquad \mathbf{z}^l = (\mathbf{w}^l)^T \cdot \mathbf{a}^{l-1} + \mathbf{b}^l \tag{1.2}$$

Denoting with L the total number of layers in the network, the output layer will then be represented as $\mathbf{a}^L$.

Typical choices for the activation function include the sigmoid, error function (erf), or hyperbolic tangent (tanh), as well as the rectified linear unit (ReLU). Although the term "perceptron" might suggest the use of the Heaviside step function as the activation function, this is generally avoided in practice due to issues with learning parameter updates.



In order to learn the parameters of our model $\{\mathbf{w}, \mathbf{b}\}$ we define a cost function C (also referred to as the loss function). This cost function measures how well the model performs on the task, reflecting how closely the outputs predicted by the model align with the true labels from the dataset. The goal of training is to find the optimal values for $\mathbf{w}$ and $\mathbf{b}$ that minimize the cost function, ensuring that the predictions closely match the true outputs.

An example of cost function is given by the mean squared error (MSE):

$$C(\mathbf{w}, \mathbf{b}) \equiv \frac{1}{n} \sum_x \|\mathbf{y}(x) - \mathbf{a}^L\|^2.$$

Where n is the total number of training inputs, and $\mathbf{y}(x)$ is the true output for the input x. Naturally, the output a depends on x, w, and b, but for simplicity, this dependence is not explicitly shown in the notation. The notation $\|v\|$ denotes the standard length (or norm) of a vector v.

1.2 Stochastic Gradient Descent (SGD)

A classical method for learning the parameters of a neural network is stochastic gradient descent (SGD). The goal is to minimize the cost function, and the general approach is based on the gradient of this function.

At first order, we have:

$$\Delta C = \nabla_{\mathbf{w}} C \cdot \Delta \mathbf{w} + \nabla_{\mathbf{b}} C \cdot \Delta \mathbf{b}$$

If we choose $\Delta \mathbf{w}$ and $\Delta \mathbf{b}$ appropriately, we can ensure that ΔC becomes negative, meaning the cost function decreases with each update step. The key idea is to select these parameter updates in the opposite direction of the gradient of the cost function.

In practice, this means that the updates are computed as:

$$\Delta \mathbf{w} = -\eta \nabla_{\mathbf{w}} C \quad \text{and} \quad \Delta \mathbf{b} = -\eta \nabla_{\mathbf{b}} C$$

where η is the learning rate, a small positive constant that controls the step size. By iterating these updates, SGD gradually refines the model parameters driving the cost function towards a minimum.

The main challenge is how to compute the gradient of the cost function with respect to the model parameters. Under the general assumption that the cost function can be written as $C = \frac{1}{n} \sum_x C_x$, where n is the number of inputs x, the gradient is given by

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x$$

In practice, to compute the gradient ∇C, we need to compute the gradients ∇C_x separately for each training input x, and then average them. Unfortunately, when the number of training inputs is very large, this process can be time-consuming, resulting in slow learning.

The key idea behind Stochastic Gradient Descent is to estimate the gradient ∇C by calculating ∇C_x for a small, randomly selected subset of training inputs (hence the term "stochastic"). By averaging over this subset, one can obtain a good approximation of the true gradient ∇C much faster, which accelerates gradient descent and the learning process.

To formalize this approach: SGD works by randomly selecting a small number m of training inputs, labeled $x_1, x_2, \ldots, x_m$, and forming what is known as a "mini-batch". Provided the mini-batch size m is sufficiently large, the average of the gradients ∇C_{x_j} for the mini-batch will be approximately equal to the average over all ∇C_x, i.e.:

$$\frac{1}{m} \sum_{j=1}^{m} \nabla C_{x_j} \approx \frac{1}{n} \sum_x \nabla C_x = \nabla C$$

This confirms that we can estimate the overall gradient by computing gradients for only the randomly selected mini-batch.

Once all the inputs in the dataset have been used for training, we say that one "epoch" has passed.

1.2.1 Backpropagation

The goal of backpropagation is to compute the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ of the cost function C with respect to any weight w or bias b in the network. To make backpropagation work, we need to make two key assumptions about the form of the cost function.

The first assumption is that the cost function can be written as an average $C = \frac{1}{n}\sum_x C_x$, where n is the number of training examples x and C_x is the cost function for each individual example.

This assumption is necessary because backpropagation computes the partial derivatives $\frac{\partial C_x}{\partial w}$ and $\frac{\partial C_x}{\partial b}$ for a single training example. We then obtain $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ by averaging these partial derivatives across all examples. With this assumption, we can simplify notation by considering a fixed training example x and dropping the subscript x, writing the cost C_x simply as C.

The second assumption we make about the cost is that it can be written as a function of the outputs from the neural network, namely a^L.

In order to compute ∇C, we introduce an intermediate quantity, δ_j^l, which we call the error of the j-th neuron in the l-th layer. The error δ_j^l is related to the neuron's pre-activation z_j^l, indeed we define the error δ_j^l for neuron j in layer l by:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

As per usual conventions, we use $\boldsymbol{\delta}^l$ to denote the vector of errors for all neurons in layer l.

Backpropagation provides a systematic way to compute the error δ_j^l and relate it to $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$.

Equation for the error in the output layer, δ^L:

The components of δ^L are given by:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

This expression follows directly from the chain rule, and we can observe that $\frac{\partial C}{\partial a_j^L}$ measures how rapidly the cost changes with respect to the j-th output.

Equation for the error δ^l in terms of the error in the next layer, δ^{l+1}:

Specifically, we have:

$$\boldsymbol{\delta}^l = ((\mathbf{w}^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot \sigma'(\mathbf{z}^l)$$

Here the operator $\odot$ denotes the Hadamard (element-wise) product. This equation describes how the error at layer $l+1$ is propagated backward to layer l. The matrix multiplication with $(\mathbf{w}^{l+1})^T$ can be intuitively thought of as moving the error backward through the network, giving us a measure of the error at layer l. The Hadamard product with $\sigma'(\mathbf{z}^l)$ adjusts the error as it moves backward through the activation function at layer l.

By iterating this process, starting with the error in the output layer, we can compute the error δ^l for any layer in the network. Specifically, we first use the equation for δ^L, and then apply the equation for δ^l to move backward through the layers, one by one, until the error is computed for all layers.

Equation for the rate of change of the cost with respect to any bias in the network:

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

Equation for the rate of change of the cost with respect to any weight in the network:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

This tells us how to compute the partial derivatives $\frac{\partial C}{\partial w^l_{jk}}$ in terms of the error $\boldsymbol{\delta}^l$ and the outputs $\mathbf{a}^{l-1}$, which we already know how to compute.

1.3 Monte Carlo methods

The name "Monte Carlo" was applied to a class of mathematical methods first used by scientists working on the development of nuclear weapons in Los Alamos in the 1940s. The core idea of these methods involves the creation of games of chance, where the behavior and outcomes can be employed to study various interesting phenomena. Although there is no inherent connection to computers, the effectiveness of numerical or simulated gambling as a serious scientific approach is significantly enhanced by modern digital computing technology.

To illustrate what we mean by Monte Carlo calculations, consider a circle and its circumscribed square. The ratio of the area of the circle to that of the square is $\pi/4$. It is reasonable to hypothesize that if points are placed randomly within the square, the fraction $\pi/4$ will also lie within the circle. If this holds true (and we will later demonstrate that it does, in a certain sense), one could measure $\pi/4$ by placing a round cake pan with a diameter L inside a square cake pan with side L and collecting rainwater in both. Additionally, a computer can be programmed to generate random pairs of Cartesian coordinates to represent

random points within the square and count the fraction that falls inside the circle. Based on many experiments, this fraction should approximate $\pi/4$, and it will be referred to as an estimate for $\pi/4$. In one million experiments, there is a high probability (95% chance) that the number of points inside the circle will range from 784,600 to 786,200, resulting in estimates of $\pi/4$ between 0.7846 and 0.7862, compared to the true value of 0.785398.

This example demonstrates how random sampling can be utilized to solve a mathematical problem, specifically the evaluation of a definite integral.

$$I = \int_0^1 \int_0^{\sqrt{1-x^2}} dx\, dy. \tag{1.3}$$

The answers obtained by the above procedure are statistical in nature and subject to the laws of chance. This aspect of Monte Carlo is a drawback, but not a fatal one, since one can determine how accurate the answer is, and obtain a more accurate answer, if needed, by conducting more experiments. Despite the randomness, Monte Carlo can sometimes provide the most accurate result for the amount of computer time used. For calculating the value of π, non-Monte Carlo methods are faster and more precise. However, for problems involving many dimensions, Monte Carlo methods are often the only practical way to solve integrals.

A second and complementary example of a Monte Carlo calculation is the one illustrated by Stanisław Ulam, one of the pioneers behind the development of Monte Carlo methods, which gave him the idea of using random sampling to solve complex problems. Consider the task of estimating the probability of winning at solitaire, given that the deck is perfectly shuffled before the cards are dealt. Once a specific strategy for stacking the piles of cards is selected, the problem becomes a simple one in basic probability theory, though it can be quite labor-intensive. Alternatively, it wouldn't be too challenging to create a computer program that randomizes a list representing the 52 cards in the deck, organizes them into different piles, and simulates the game until it concludes. By observing the outcomes over numerous iterations, one could obtain a Monte Carlo estimate of the likelihood of winning. In fact, this approach would be the most straightforward way to arrive at such an estimate.

Nowadays, random numbers are utilized in various applications related to computers, such as in video games and the generation of synthetic data for testing purposes. While these applications are certainly intriguing, they do not fall under the category of Monte Carlo methods, as they do not yield numerical outcomes. A Monte Carlo method can be defined as one that intentionally incorporates random numbers into a calculation structured around a stochastic process. By stochastic process, we refer to a series of states whose progression is influenced by random events. In computing, these random events are produced by a deterministic algorithm that generates a sequence of pseudo-random numbers, which emulate the characteristics of genuine random numbers.

We want to revisit the question of whether Monte Carlo calculations are genuinely worthwhile. A practical answer to this is that many individuals uti-

lize them, and they have become an established aspect of scientific practice across various disciplines. The motivations for their use aren't solely based on computational efficiency. As illustrated in our solitaire example, the convenience, simplicity, directness, and expressiveness of the method are significant advantages, particularly as computational power becomes more affordable. Furthermore, as noted in discussions about π, Monte Carlo methods can be more computationally efficient than deterministic approaches when dealing with problems in multiple dimensions. This is one reason why they are widely employed in operations research, radiation transport (where challenges can involve up to seven dimensions), and particularly in statistical physics and chemistry, where systems consisting of thousands of particles can be analyzed routinely.

Chapter 2

MLPs and emergence of convolutional structure

In this chapter we analyze the behavior of a MLP with a single hidden layer in the case where, after training, two distinctly different classes of neurons emerge: localized neurons and oscillating neurons. In order to obtain such result we exploited what has been shown in [3]. Then we used the Wang Landau algorithm to test the robustness of the trained network.

2.1 Network Training

The network model we train is a MLP with two fully connected layers and single output. We denote with D the number of input nodes and with K the number of nodes in the hidden layer. As activation function we use $tanh$, while no activation function is applied to the output layer. Thus the output of the network to an input $\mathbf{x}$ is given by:

$$\phi(\mathbf{x}) = \sum_{k=1}^{K} v_k \tanh\left(\sum_{i=1}^{D} w_{ki} x_i + b_k\right) + b_{out} \tag{2.1}$$

where:

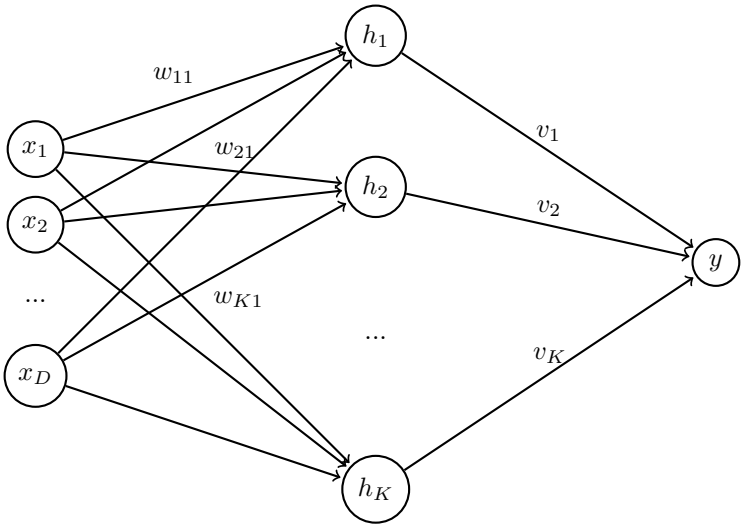
- $\mathbf{W} \in \mathbb{R}^{K \times D}$ is the matrix of first-layer weights. We initialized $\mathbf{W}$ with independent identically distributed (i.i.d.) zero-mean Gaussian entries with variance $1/D$. For each neuron k in the hidden layer we can identify its receptive field (RF) (or equivalently its weight vector $\mathbf{w}_k$) as the k-th row of W.
- $\mathbf{V} \in \mathbb{R}^{1 \times K}$ is the matrix of second-layer weights. We set $v_k = 1/\sqrt{K}$ and we don't train this layer unless otherwise specified; with this setting is easier to obtain localized neurons.

- b_k are the hidden unit biases and b_{out} is the bias of the network output. We set $b_{out} = 0$ and we don't train this parameter unless otherwise specified.

Given an input $\mathbf{x}$, we can define the internal representation of that input as the (K dimensional) vector $\mathbf{h}$ output of the hidden layer, i.e:

$$h_k(\mathbf{x}) = \tanh\left(\sum_{i=1}^{D} w_{ki} x_i + b_k\right) \qquad {\scriptstyle k=1,\dots,K} \tag{2.2}$$

Here is a diagram of the model:



Throughout this work, we train our networks on the synthetic NLGP and GP datasets introduced in [3], where one or two-dimensional inputs are drawn from a stationary Gaussian process and then optionally passed through a nonlinearity (see 3.1 for more details). Such synthetic datasets allows us to tune both the size of the resulting receptive fields and the task difficulty, in a manner explained in the following. We define with α_{train} the ratio between the size of the train dataset and the input dimension D (and similarly for α_{test}); we set $D = 50$ unless otherwise stated. With regards to the difficulty of the task, the more ξ^- and ξ^+ are similar to each other the more the task will be difficult to solve.

The network is trained using stochastic gradient descent (SGD). We use the mean squared error (MSE) loss function and introduce an additional L2 regularization factor λ_{L2} in order to control the size of the weights. We set $\lambda_{L2} = 0.1$, unless otherwise specified. Our cost function is thus given by:

$$loss = \frac{1}{N}\sum_{\mu=1}^{N}\left(\phi(\mathbf{x}^\mu) - y^\mu\right)^2 + \lambda_{L2}\sum_i w_i^2 \tag{2.3}$$

where $N = \alpha_{train} D$ is the number of inputs in the training set; $\sum w_i^2$ is the sum over the square of all the weights of the network; $y^\mu \in \{-1, 1\}$, $\phi(\mathbf{x}^\mu)$ are

respectively the true output and the network output to the input $\mathbf{x}^\mu$. We set the learning rate $\eta = 0.1$, unless otherwise specified.

Since our model is ultimately a binary classifier, an input $\mathbf{x}^\mu$ is correctly classified if $\text{sgn}(\phi(\mathbf{x}^\mu) \cdot y^\mu) > 0$. That means that, given a dataset $\mathcal{D}$, we can compute the accuracy of the model as:

$$\text{acc}_\mathcal{D} = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x}^\mu, y^\mu) \in \mathcal{D}} \Theta(\phi(\mathbf{x}^\mu) \cdot y^\mu) \tag{2.4}$$

with Θ the Heaviside step function.

2.2 Localized and oscillatory neurons

Training our model on an NLGP dataset, we has been able to replicate the results obtained in [3]. More specifically, after learning the hidden neurons split into two groups of similar size: localized and oscillating.

Localized neurons features localized RFs: they only have a few synaptic weights whose magnitude is significantly larger than zero in a small region of input space (Fig. 2.1). On the contrary oscillating neurons RFs show highly oscillating patterns. (Fig. 2.2)

In order to classify a neuron as localized or oscillatory, we use a clustering method (see 3.4) and compute the average inverse participation ratio (IPR) for the weight vectors in each cluster. The IPR is considered a measure of how localized a weight vector is (see 3.2.1). If, at the end of training, we observe the emergence of two distinct clusters, and one of them exhibits a high average IPR, we infer that this cluster likely contains localized neurons. To verify this, we examine the plots of the neurons in each cluster.

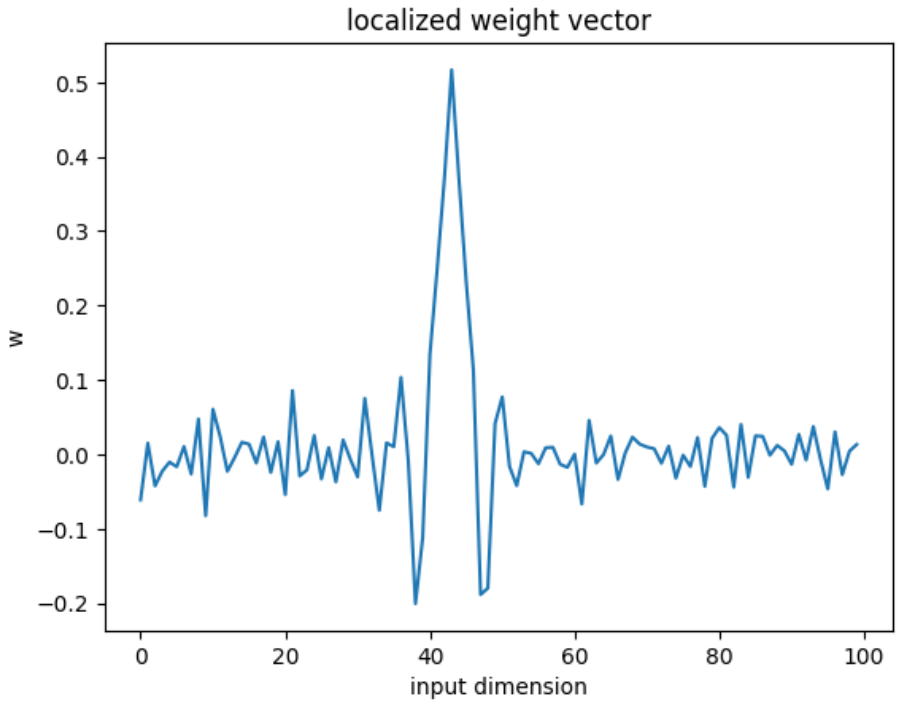


Figure 2.1: Example of localized weight vector. Parameter of the network: $N = 100$; $K = 100$; $\alpha_{train} = 8000$; $\xi^- = \sqrt{3}$, $\xi^+ = 3$; $\lambda_{L2} = 0.2$; $\eta = 0.1$

2.3 Network Robustness

By network robustness, we refer to the network's ability to effectively perform its task even if some of its hidden neurons become inactive after training. Depending on which neurons are active or inactive, our system can exist in 2^K different configurations $\sigma \in \{0,1\}^K$. Given a dataset $\mathcal{D}$, each possible state of the system σ is associated with an energy that depends on the number of incorrectly classified inputs:

$$E_{\mathcal{D}}(\sigma) = \sum_{(\mathbf{x},y)\in\mathcal{D}} \Theta\left(-\phi_\sigma(\mathbf{x}) \cdot y\right) \tag{2.5}$$

where the output of the model in configuration σ for input $\mathbf{x}$ is defined as:

$$\phi_\sigma(\mathbf{x}) = \sum_{k=1}^{K} \sigma_k v_k \tanh\left(\sum_{i=1}^{D} w_{ki} x_i + b_k\right) + b_{out} \tag{2.6}$$

We will refer to state of the system where all the neurons are active as $\sigma_{\mathbf{1}}$.

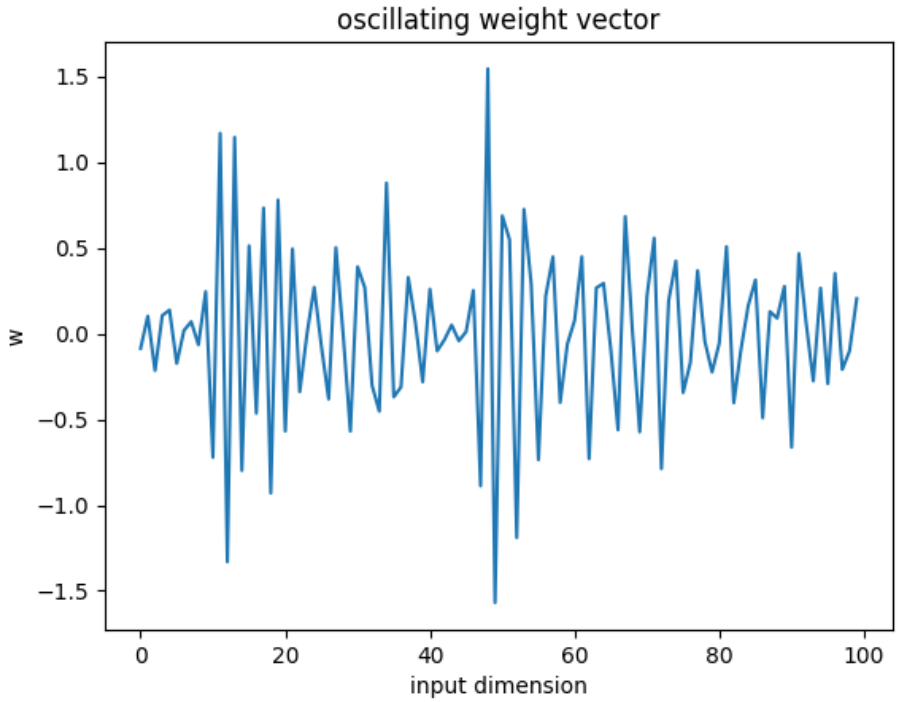


Figure 2.2: Example of localized weight vector. Parameter of the network: $N = 100$; $K = 100$; $\alpha_{train} = 8000$; $\xi^- = \sqrt{3}$, $\xi^+ = 3$; $\lambda_{L2} = 0.2$; $\eta = 0.1$

On way to characterize the robustness of the network is thus to compute its density of state (DOS):

$$g(E) = \frac{1}{Z} \sum_{\sigma} \delta(E - E_{\mathcal{D}}(\sigma)) \propto \sum_{\sigma} \delta(E - E_{\mathcal{D}}(\sigma)) \tag{2.7}$$

with Z a normalization factor that will be omitted from now on.

Since the number of configurations grows exponentially with K, computing $g(E)$ exhaustively becomes computationally intractable even for moderately large values of K. Therefore, we use the Wang-Landau algorithm to estimate $g(E)$ more efficiently (see 3.3).

For the sake of clarity, what we actually compute using the WL algorithm is a numerical (discretized) approximation of the density of states $g(E)$, where energy values are grouped into intervals of width δE. This approximation can be expressed as:

$$g(E) \propto \sum_{\sigma} \chi_{[E,E+\delta E)} \left(E_{\mathcal{D}}(\sigma)\right) \tag{2.8}$$

where we defined the boxcar function χ as:

$$\chi_{[E,E+\delta E)} \left(E_{\mathcal{D}}(\sigma)\right) = \begin{cases} 1 & \text{if } E \leq E_{\mathcal{D}}(\sigma) < E + \delta E, \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

The quantity $g(E)$ thus represents the number of configurations with energies within the interval $[E, E + \delta E)$.

We are also interested in determining whether the configurations perform consistently on both the training set and the test set, or if there are outliers that perform well on one set but poorly on the other. To explore this, wecan compute the joint density of states $g(E_{\text{tr}}, E_{\text{test}})$, which quantifies the number of configurations with energy values within specific intervals for both sets. The discretized form of this joint density of state is given by:

$$g(E_{tr}, E_{test}) \propto \sum_{\sigma} \chi_{[E_{tr},E_{tr}+\delta E_{tr})} \left(E_{\mathcal{D}_{tr}}(\sigma)\right) \cdot \chi_{[E_{test},E_{test}+\delta E_{test})} \left(E_{\mathcal{D}_{test}}(\sigma)\right) \tag{2.10}$$

Intuitively, we expect that configurations with more active neurons generally perform better. To test this hypothesis, we compute the joint density of states $g(E, n)$, where n represents the number of active neurons. Its discretized form is given by:

$$g(E, n) \propto \sum_{\sigma} \chi_{[E,E+\delta E)} \left(E_{\mathcal{D}}(\sigma)\right) \cdot \delta_{n,\sum_k \sigma_k} \tag{2.11}$$

This allows us to compare the density of states for configurations with different numbers of active neurons by conditioning on n.

Ultimately we want to show how the fraction of localized neurons among the active ones affect the performance of the network. If we define the set $\mathcal{L} = \{k \in \{0, 1, \dots, K\} | \mathbf{w}_k \text{ is localized}\}$, we can define this fraction as:

$$\lambda(\sigma) = \frac{\sum_{k \in \mathcal{L}} \sigma_k}{n} \tag{2.12}$$

In particular it is useful for later results define $\lambda_{\mathbf{1}} = \lambda(\sigma_{\mathbf{1}})$, i.e. the fraction of localized neurons for the state where all the neurons are active $\sigma_{\mathbf{1}}$.

We then computed the joint density of states $g(E, n, \lambda)$, whose discretized form is given by:

$$g(E, n, \lambda) \propto \sum_{\sigma} \chi_{[E, E+\delta E)} \left(E_{\mathcal{D}}(\sigma)\right) \cdot \delta_{n, \sum_k \sigma_k} \cdot \chi_{[\lambda, \lambda+\delta\lambda)} \left(\lambda(\sigma)\right) \tag{2.13}$$

2.4 Results

We now present the results from a specific case study in which we can observe the emergence of localized and oscillatory neurons.

The number of hidden neurons we use is $K = 50$. We generate an NLGP dataset for training the network. The input dimension was set to $D = 50$, with $\alpha_{\text{train}} = 1000.0$ and $\alpha_{\text{test}} = 30.0$, resulting in a training set of $N_{\text{tr}} = 50,000$ elements and a test set of $N_{\text{test}} = 1,500$ elements. Finally, we set $\xi^- = 1$ and $\xi^+ = 2$. Given the network's size, these values make the task challenging enough to allow for the emergence of localized and oscillating RFs, while still enabling us to achieve high accuracy both in the training and test sets.

We train the network for 2500 epochs reaching as final accuracy on the training and on the test set respectively $\text{acc}_{tr} = 0.9869$ and $\text{acc}_{test} = 0.9813$ (2.3).

Throughout the training, we monitor several key statistics related to the model's parameters and the internal representation of the inputs (3.2). This allow us to investigate the model dynamics and gain insights into its functioning.

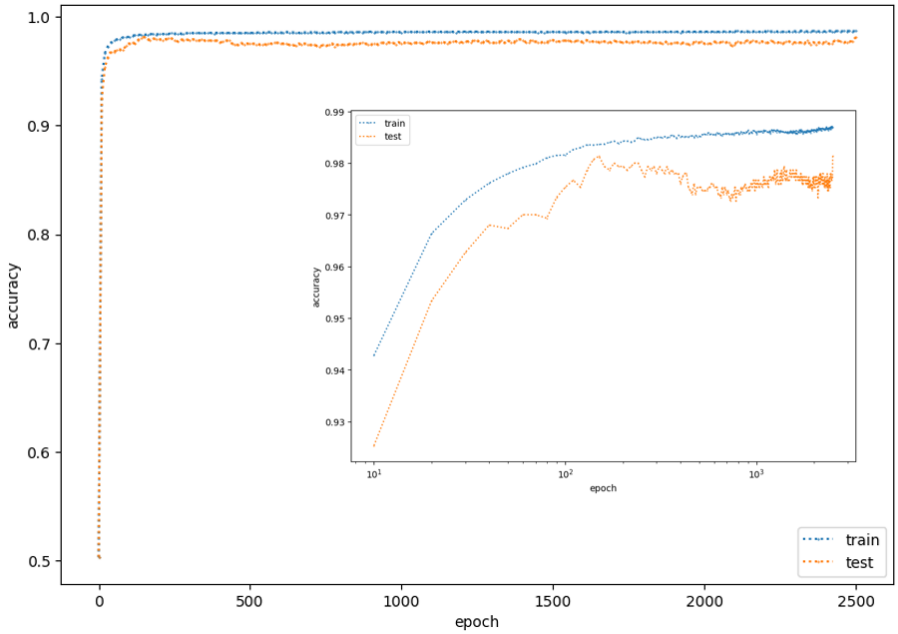


Figure 2.3: Accuracy per epoch for the training and test sets. The inset displays the same plot with a logarithmic scale on the x-axis.

2.4.1 Training dynamics

After training, we observe that each neuron in the hidden layer can be classified as either oscillating or localized (Fig. 2.4, Fig. 2.5).

Out of the 50 weight vectors, 28 are classified as "localized" and the remaining 22 as "oscillating". Therefore the localization factor of the network in case all neurons are active is $\lambda_1 = 0.56$.

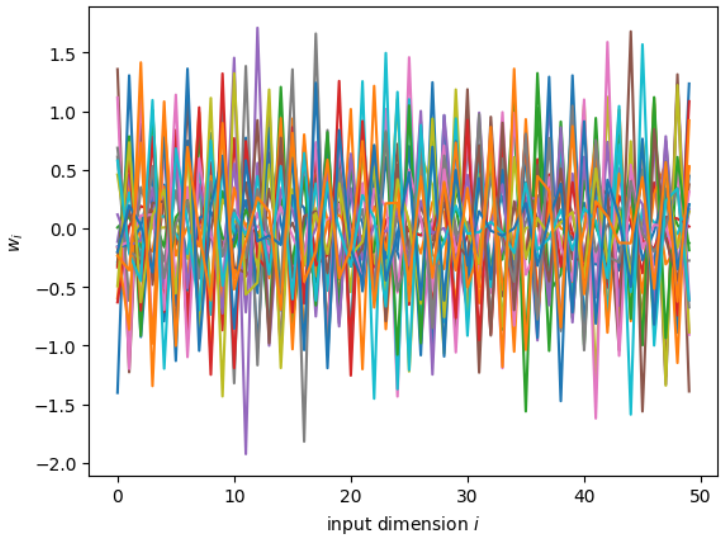


Figure 2.4: RFs classified as "oscillating" after training. ($K = 50$, $\alpha_{tr} = 10^3$, $\xi^- = 1.0$, $\xi^+ = 2.0$)

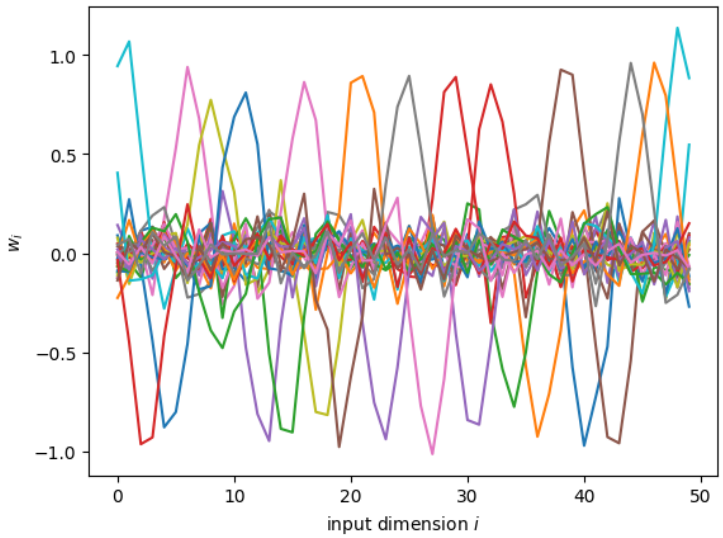


Figure 2.5: RFs classified as "localized" after training. ($K = 50$, $\alpha_{tr} = 10^3$, $\xi^- = 1.0$, $\xi^+ = 2.0$)

By analyzing the mean and variance of the weight vectors over epochs, we find that, as expected, oscillating neurons typically exhibit a near-zero mean and

high variance, while localized neurons show a non-zero mean and low variance (Fig. 2.6).

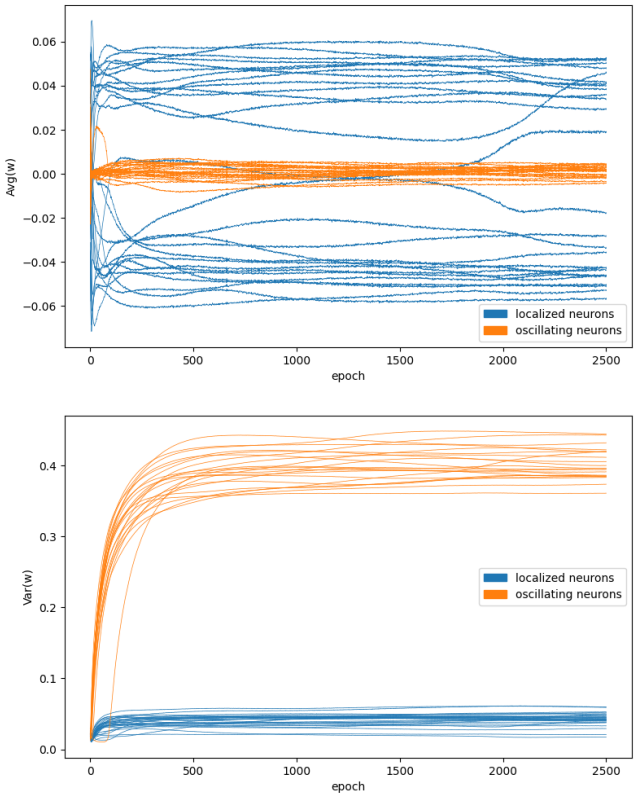


Figure 2.6: Mean of weight vectors per epoch (Top) and variance of weight vectors per epoch (Bottom). For each weight vector $\mathbf{w}_i$, we tracked the mean and variantce of its components across epochs. The classification into "localized" and "oscillatory" is performed after training. This implies that a receptive field (RF) classified as localized at the end of training may have been classified as oscillatory, or neither, at some earlier stage of the process. We observe that localized neurons exhibit high variance and a near-zero mean, while oscillatory neurons display a positive or negative mean (around ± 0.4) and low variance.

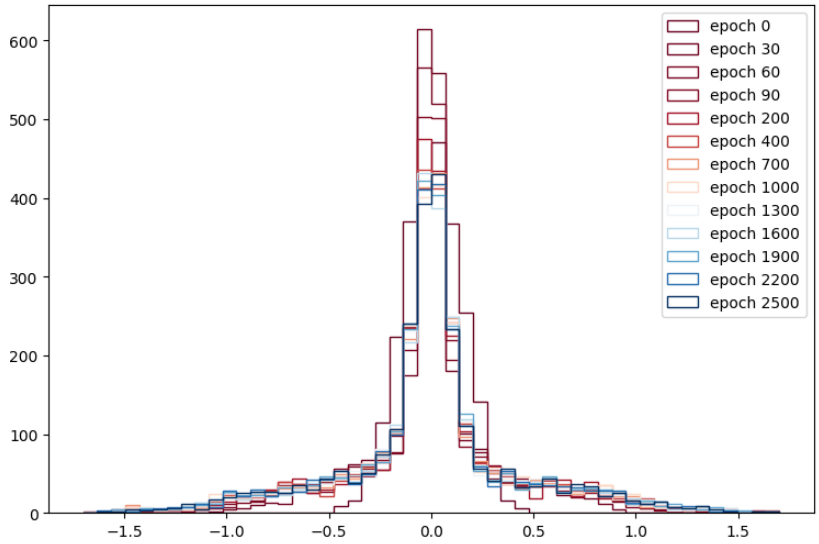


Figure 2.7: Flatten weights distribution (histogram of all the entries of W) epoch by epoch. We can see how the regularization factor λ_{L2} prevents the weights from growing indefinitely in magnitude.

2.4.2 Internal representations and Outputs

Examining the internal representations, we observe that each neuron attempts to distinguish between inputs of different classes. A localized neuron $\mathbf{w}_l$ mainly produces negative outputs h_l. These outputs become more negative for inputs labeled $y = -1$, while inputs labeled $y = 1$ result in outputs with lower magnitude, and slightly positive outputs occur more often. Conversely, oscillating neurons mainly generate positive outputs, which are more positive for inputs labeled $y = 1$, while inputs labeled $y = -1$ produce outputs with lower magnitude, and in some cases, slightly negative outputs. (Fig. 2.8 and 2.9).

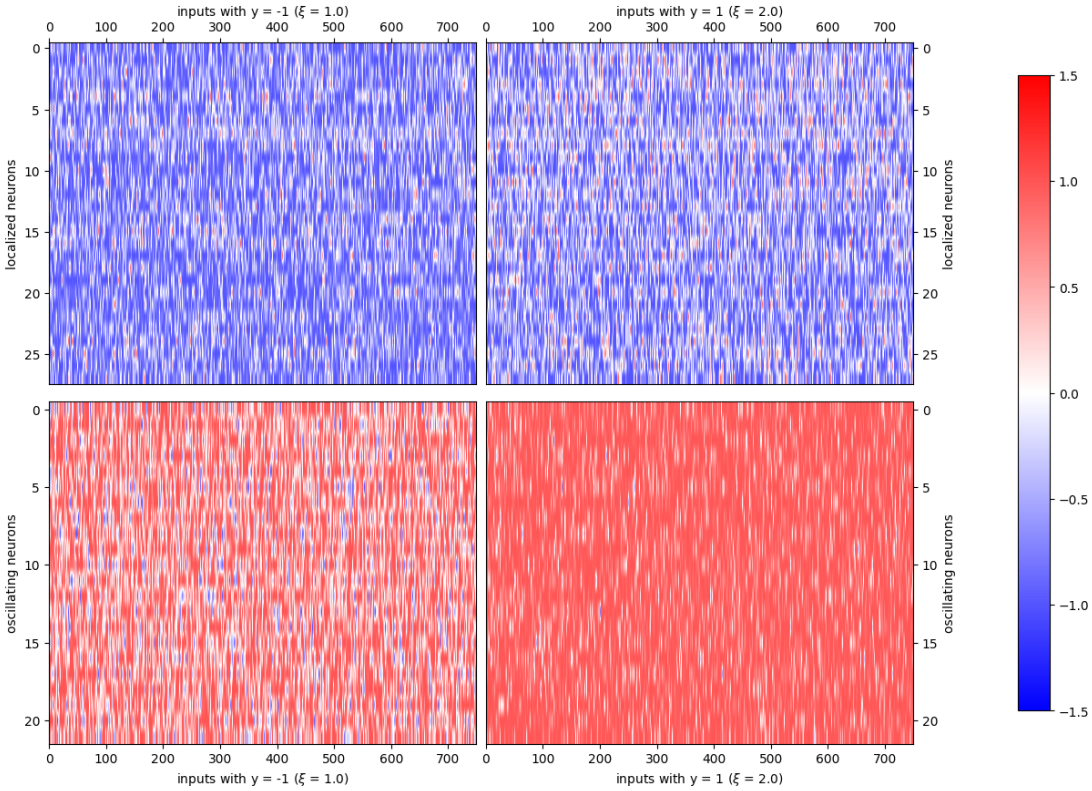


Figure 2.8: Training set internal representations. Each column μ is the internal representation of the input $\mathbf{x}^\mu$, i.e. the element at row k and column μ is given by $h_k(\mathbf{x}^\mu)$. The resulting matrix is divided in four quadrant conditioning on the label of the inputs and the neuron type.

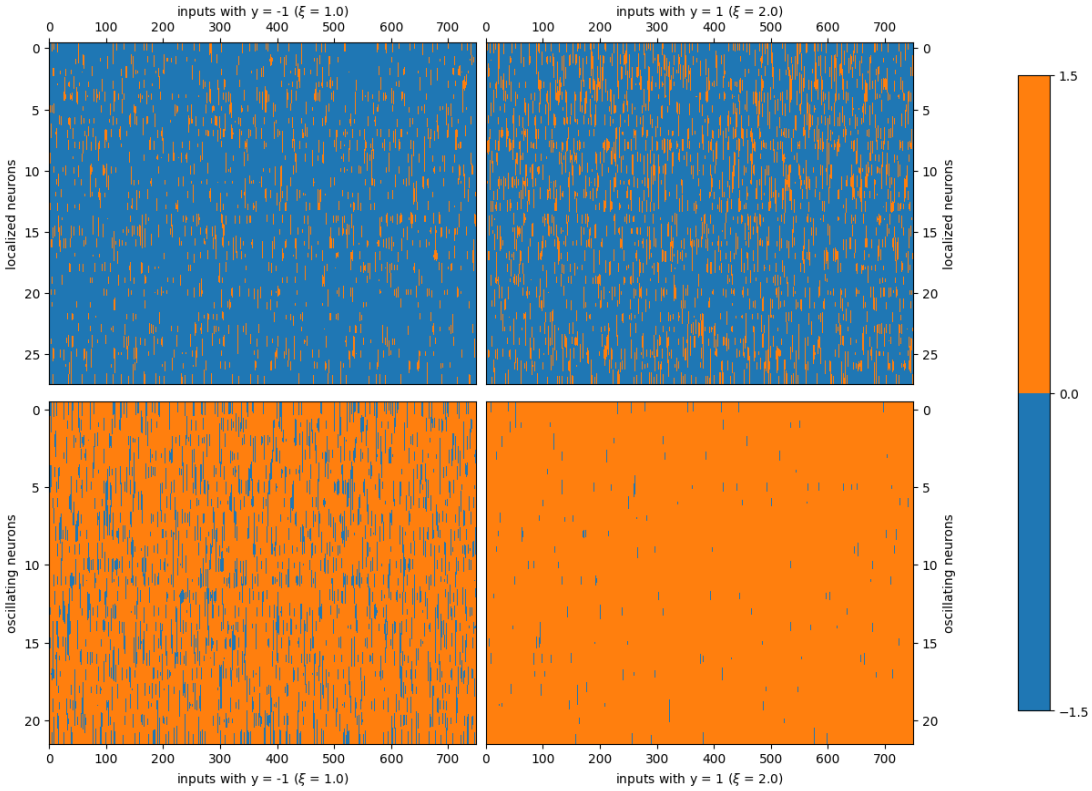


Figure 2.9: A variant of Fig. 2.8. The color map used here distinguishes internal representations in the following way: values less than 0 are shown in light blue, and values greater than 0 are shown in orange. Comparing the two figures, we observe that the negative outputs of oscillating neurons have a small magnitude, as do the positive outputs of localized neurons.

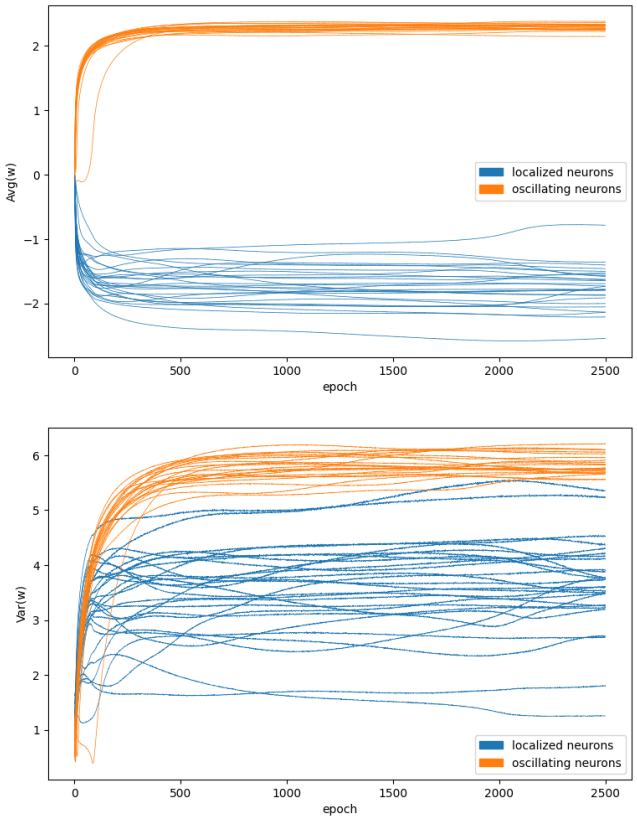


Figure 2.10: Mean of the internal representations (Top) and variance of the internal representations (Bottom). For each component of $h(\mathbf{x})$, at each epoch, we computed the mean and variance across the entire training set. The classification into "localized" and "oscillatory" is performed after training, meaning that a receptive field (RF) classified as localized at the end of training might have been classified as oscillatory, or neither, at an earlier stage. We observe that localized neurons exhibit a negative mean, while oscillatory neurons display a positive mean. Additionally, oscillatory neurons show higher variance compared to localized ones.

If we look at the variance of the internal representations (Fig. 2.10 (bottom)) we can notice that in general oscillating neurons shows a greater variance. This fact is confirmed in Fig. 2.8. Indeed, we can observe how the lower quadrants (oscillating neurons) are more different from each other compared to the upper ones (localized neurons). This suggests that oscillating neurons generally perform better at distinguishing inputs with different labels, which one can confirm by looking at the outputs of the network in different scenarios.

Consider the system state σ_{loc}, where all localized neurons are active and all oscillating neurons are inactive. By plotting histograms of the model outputs for each input class (see Fig. 2.11 (top)), we find that the two curves overlap

significantly. This overlap suggests that localized neurons are less effective at distinguish inputs with different labels. Conversely, when we examine the mirror state σ_{osc} (where oscillating neurons are active and localized neurons are inactive), the overlap between the two histograms is markedly less.

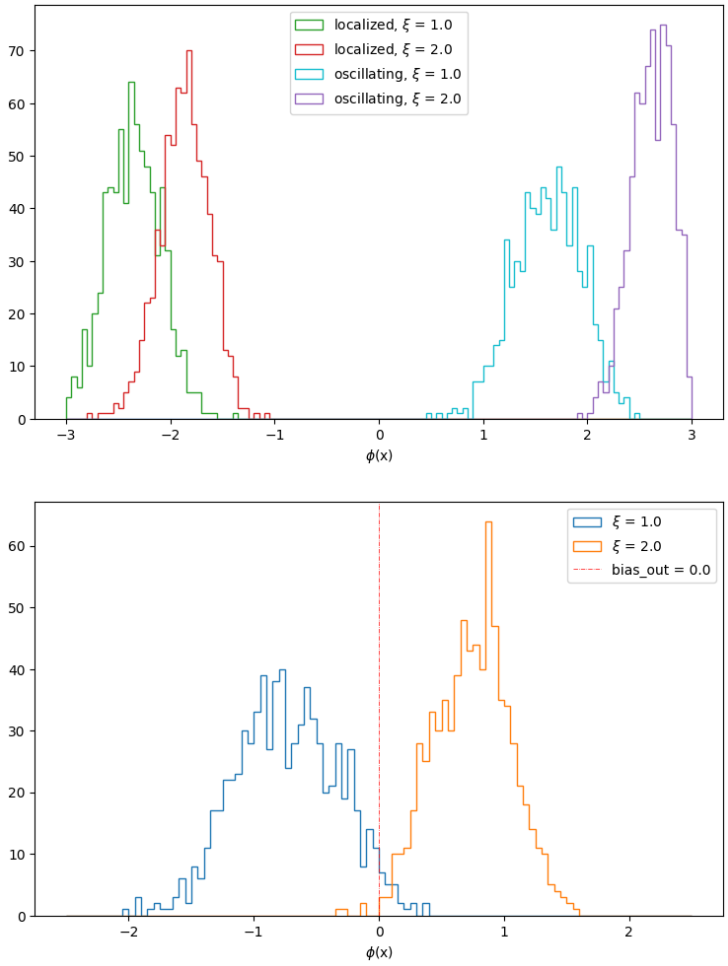


Figure 2.11: Output distributions. In the bottom figure, we examine the case where the system is in the state σ_1, where all neurons are active. The blue curve represents the distribution of network outputs for inputs labeled $y = -1$ (corresponding to $\xi = 1.0$), while the orange curve shows the distribution for inputs labeled $y = 1$ (i.e., $\xi = 2.0$). In the top figure, we perform a similar analysis but consider the states σ_{loc}, where all localized neurons are active and all oscillatory neurons are inactive, and σ_{osc}, where the opposite is true.

Another interesting aspect of the internal representations is their effective dimensionality, which is a quantitative measure of how distributed the repre-

sentations are across different dimensions (3.2.2). We observed that during training, this dimensionality initially decreases but then increases, ultimately reaching a value higher than the initial one. This non-monotonic dynamic has been observed in the training of various networks, including the work [1], and could be a focus for further investigation. Notably, this behavior may influence how the gradient progressively learns a task, reflecting an inductive bias that affects the network's ability to generalize.

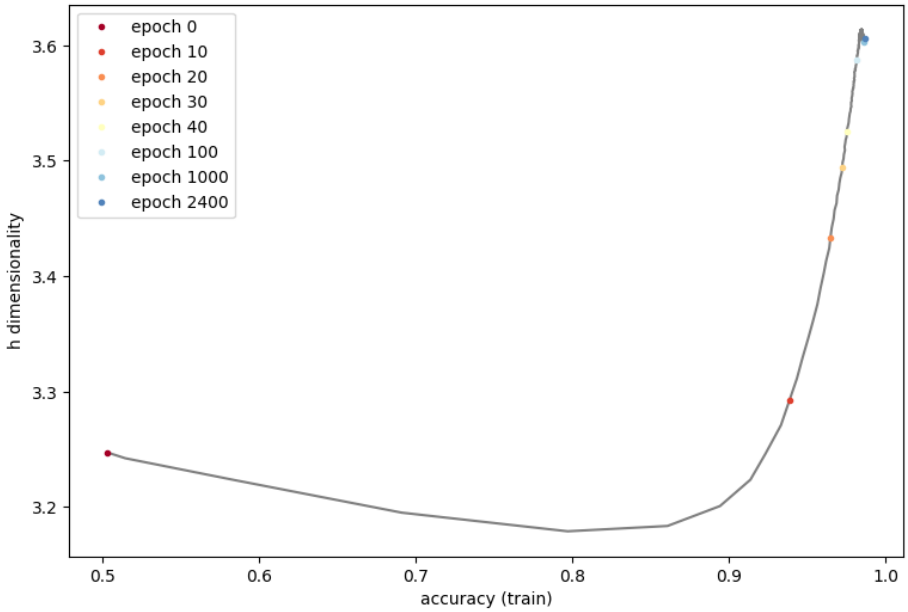


Figure 2.12: Internal representations effective dimensionality

2.4.3 Network Robustness

As anticipated in section 2.3, to fully understand the network's resilience to neuronal failure, it is useful to consider two order parameters that capture the network's key features: the total number of active neurons n and the fraction of localized neurons among the active ones λ. For this reason, it is particularly valuable to compute the joint density of states $g(E, n, \lambda)$, from which all other density of states can be derived by marginalizing over one or more variables.

The main finding of our analysis is that the network's performance in a given configuration is strongly influenced by both the number of active neurons and the difference between the system's localization parameter and that of the configuration where all neurons are active, $\lambda_{\mathbf{1}}$. Generally, configurations with a greater number of active neurons and localization closer to $\lambda_{\mathbf{1}}$ exhibit better performance. Nonetheless, it is important to emphasize that both factors are highly significant; configurations with fewer active neurons can still outperform

those with more if they demonstrate better localization.

In this section, we will first present the results on the marginalized density of states to enhance our understanding of the phenomenon. Finally, we will illustrate the complete results obtained for the full joint density of states, aiming to summarize the main results expressed above in a single visualization (Fig. 2.18).

Let's begin by examining the density of states $g(E_{test})$ across different epochs (Fig 2.13). At epoch 0, the density of states is essentially a Gaussian distribution centered around the midpoint of the system's possible energy range (i.e. acc = 0.5). This occurs because the network parameters are randomly initialized, and correct labels are guessed by chance.

As the network begins to learn the task, the distribution shifts towards lower energies (i.e. higher accuracy), as expected. We also observe that it becomes wider with respect to the initial distribution, but we don't observe any significant change in the shape of the distribution through the epochs.

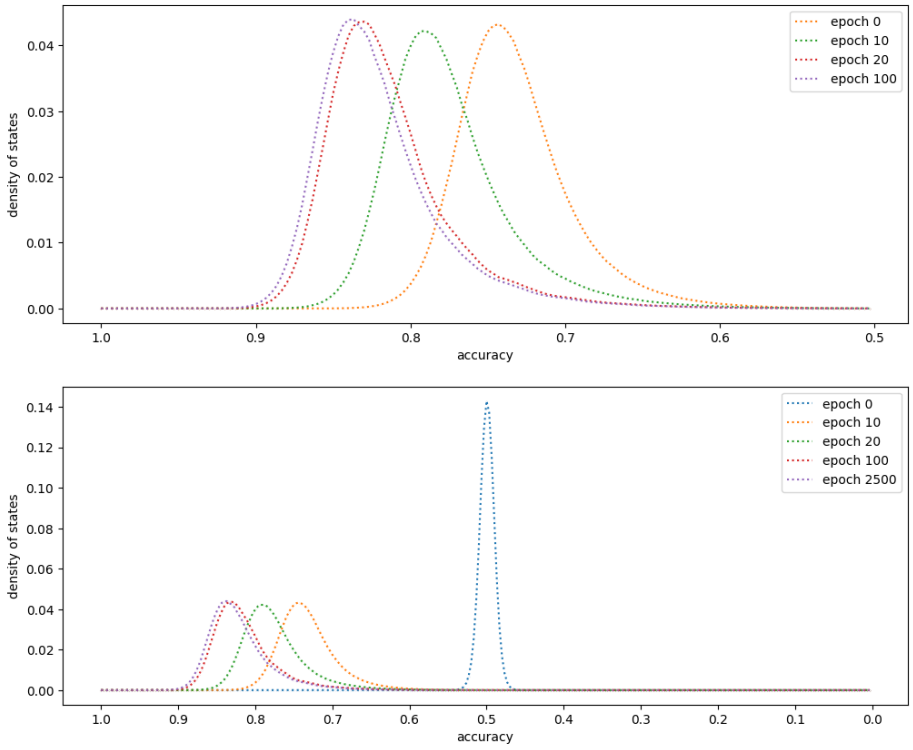


Figure 2.13: Density of states $g(E_{test})$ at different epochs. The bottom figure illustrates that at epoch 0, the density of states is concentrated around acc = 0.5. The top figure provides a zoomed-in view of the bottom figure, allowing us to better observe how the distribution progressively shifts toward higher accuracy during training.

Let us proceed to compare the density of states for the training set and the test set. The initial comparison is presented in Fig. 2.14 (top), where we have plotted the two distributions $g(E_{\text{train}})$ and $g(E_{\text{test}})$ in the same figure. However, to better understand their relationship, we computed the joint density of states $g(E_{tr}, E_{test})$ (Fig. 2.14 (bottom)). We observe that the distribution clusters along the diagonal of the $E_{tr} - E_{test}$ plane, indicating that configurations that perform well on the training set also tend to perform well on the test set, and vice versa. This further confirms that the model generalizes effectively across the input space.

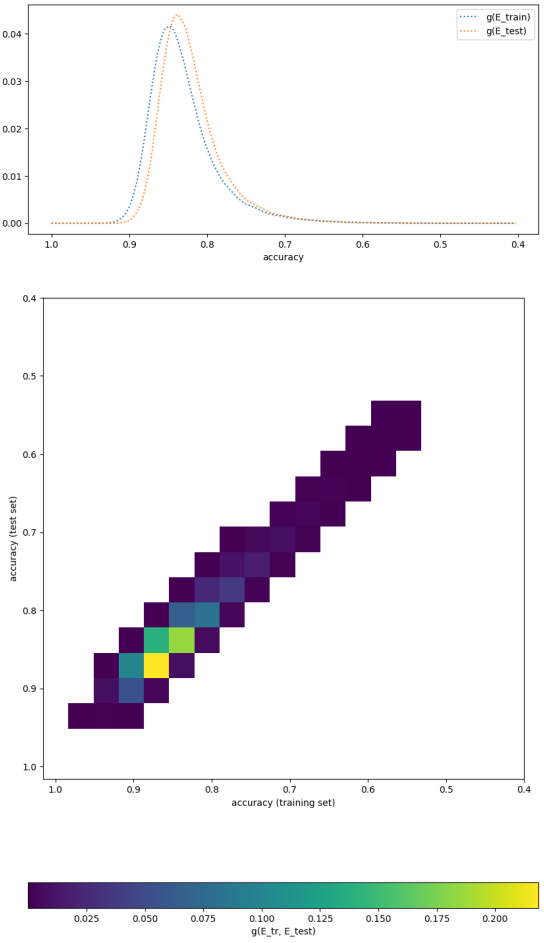


Figure 2.14: Comparison between density of states on the training set $g(E_{tr})$ and in the test set $g(E_{test})$

Let's now consider the joint density of states $g(E_{test}, n)$ (Fig 2.15). Intuitively, configurations with a higher number of active neurons should perform

better than those with fewer. This trend is generally observed, though, as anticipated, there are configurations that outperform others despite having a higher number of active neurons.

Another notable observation is that the distributions $g(E_{test}|n = i)$ ($i \in \{1, 2, \ldots, K\}$) (some examples are given in Fig 2.16) exhibit a spiky profile. To explain these phenomena, we must take into account the order parameter λ.

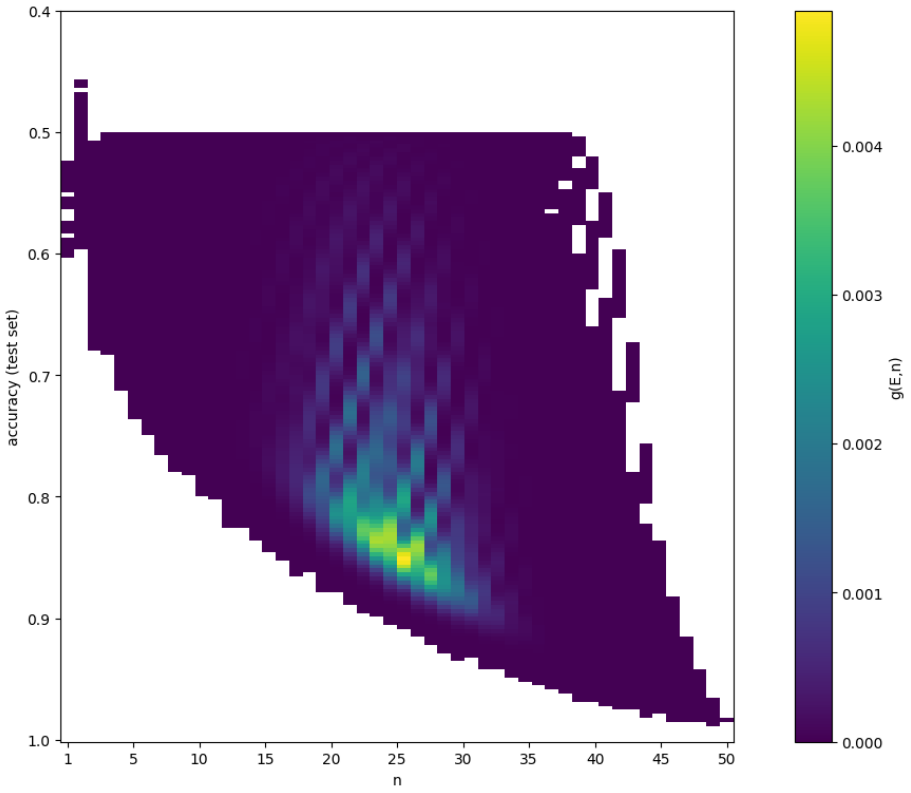


Figure 2.15: (Joint) density of states $g(E_{test}, n)$.

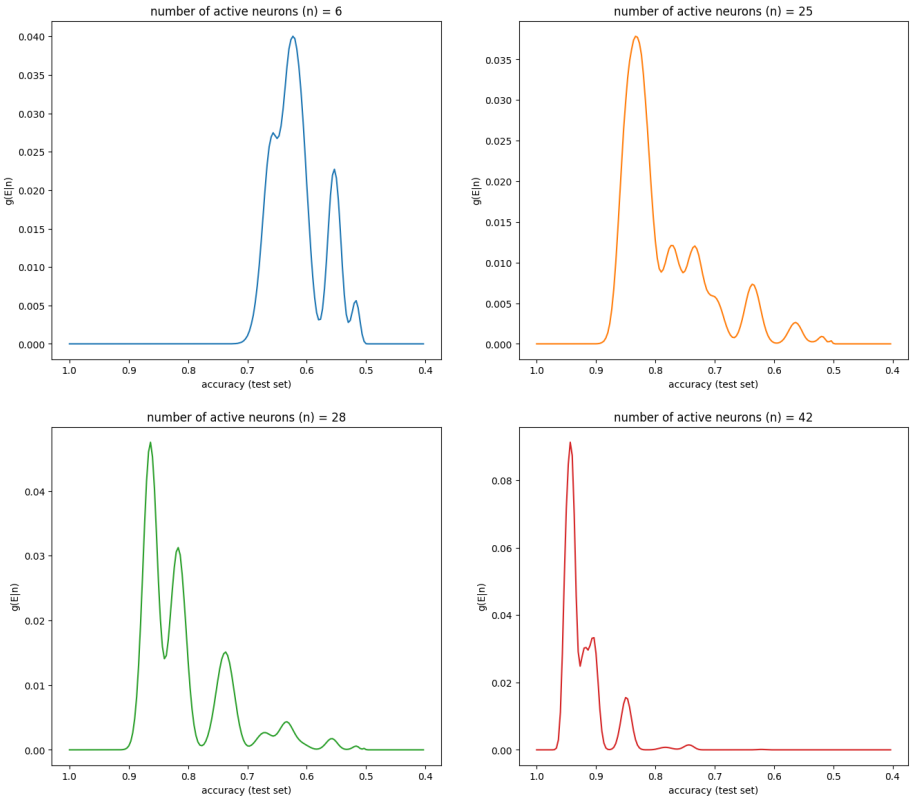


Figure 2.16: Conditional density of states $g(E_{test}|n)$ for different values of total active neurons n.

To understand the significance of the order parameter λ, let us begin by examining the joint density of states $g(E_{tr}, \lambda)$. By calculating this quantity, we can observe that the configurations achieving lower energy (i.e., better accuracy) correspond to values of λ that are closer to $\lambda_{\mathbf{1}} = 0.567$ (Fig. 2.17). This relationship becomes even more apparent when analyzing the density of states with respect to λ and the energy of the system conditioned on the number of active neurons, namely $g(E_{tr}, \lambda|n)$ (Fig. 2.18).

Looking at these conditioned probabilities moreover we can notice that, depending on n, not all values of λ can be reached. This is due to the fact that the number of active/inactive neurons is discrete, and the maximum number of active localized neurons is fixed ($|\mathcal{L}| = 28$ in our case).

For example, in the case of $n = 6$, it is clear that λ can only take values in $\{0, 1/6, 1/3, 1/2, 2/3, 5/6, 1\}$. Conversely, with a larger number of active neurons the discretization of possible λ values is finer (the maximum is reached for $n \in \{22, \dots, 28\}$) , but the lower and upper limits are no longer 0 and 1 for $n > 22$.

Generally speaking, if there are n active neurons the number of possible

value for the localization is given by $min(n, |\mathcal{L}|) - max(0, n - |\mathcal{O}|)$. Where $|\mathcal{O}| = K - |\mathcal{L}|$ is the number of oscillating neurons. This explains the spikiness of $g(E \mid n = i)$ and accounts for the spikiness of $g(E, \lambda)$ as well.

The reason why the best configuration are the ones where the fraction of localized neurons is similar to $\lambda_\mathbf{1}$ can be understood recalling that localized neurons show negative outputs $h_k(\mathbf{x}^\mu)$ for any kind of input $\mathbf{x}^\mu$ while oscillating neurons always shows positive outputs. Each neuron output contributes equally to the final output since the last-layer weights are set to $v_k = 1/\sqrt{K}$. Moreover set the output bias $b_{out} = 0$ and we did not trained it.

Consider the case where the input has a label $y^\mu = 1$. While it is true that the magnitude of the output from a localized neuron is lower than that from an oscillating neuron, the overall output should be positive and thus correctly classify the input. However, if there are too many oscillating neurons relative to the localized ones, the total output could become negative, as each neuron contributes equally. This imbalance could lead to a misclassification.

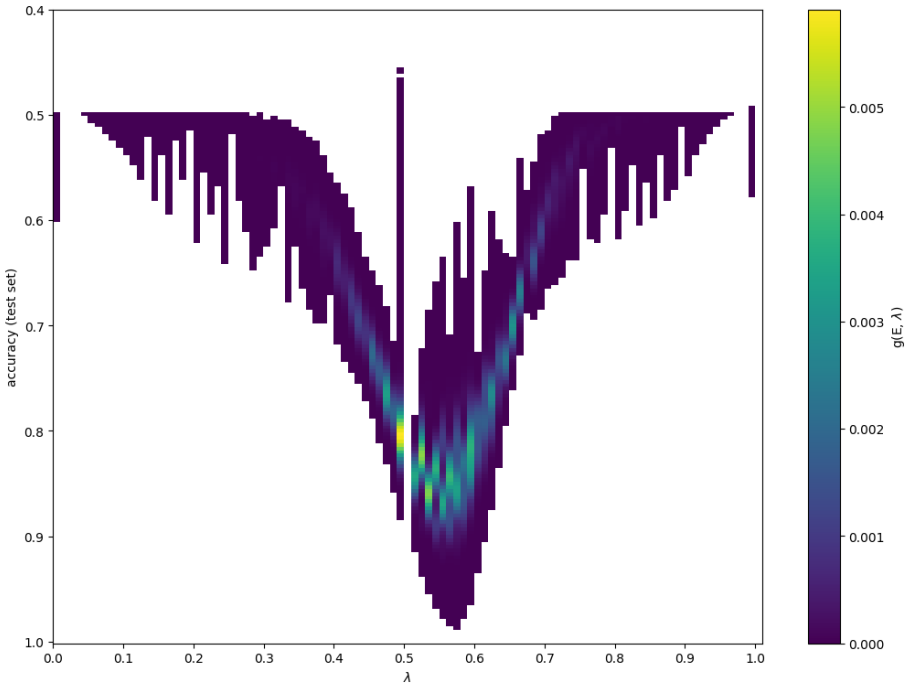


Figure 2.17: (Joint) density of states $g(E_{test}, \lambda)$. The configurations with higher accuracy are those with localization λ close to $\lambda_\mathbf{1} = 0.567$. The spikiness of the distribution arises from the non-smooth nature of the available localization distribution $g(\lambda)$. For example, there are many configurations where half of the neurons are localized, but none with λ in the range $[0.5, 0.51]$.

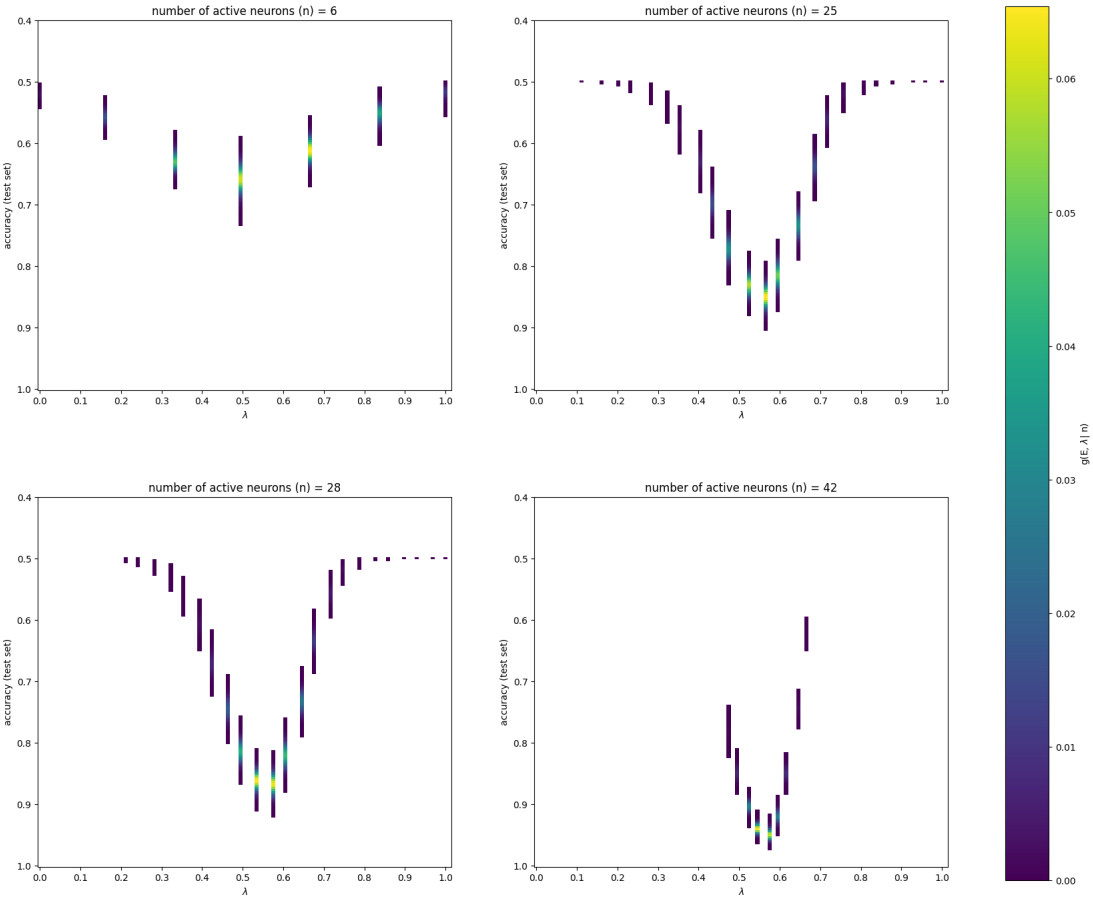


Figure 2.18: Conditional (Joint) density of states $g(E_{tr}, \lambda|n)$ for different values of total active neurons n.

We summarize the fact that the accuracy increases with the number of active neurons and with λ closer to $\lambda_{\mathbf{1}}$ in Fig. 2.19.

For each tuple (E_{tr}, n), we computed:

$$\hat{\lambda}(E_{tr}, n) = \frac{\sum_{\lambda} |\lambda - \lambda_{\mathbf{1}}| g(E_{tr}, n, \lambda)}{\sum_{\lambda} g(E_{tr}, n, \lambda)} = \langle |\lambda - \lambda_{\mathbf{1}}| \rangle_{g(E_{tr}, n, \lambda)} \tag{2.14}$$

Given a state σ with energy E_{tr} and n active neurons, this quantity measures the average deviation of the localization of the state σ from the optimal localization value $\lambda_{\mathbf{1}}$.

Examining each column reveals a gradient in the values of $\hat{\lambda}$, which supports our previous observations. A notable exception arises in the case where $n = 2$, where configurations with one localized neuron and one oscillatory neuron perform worse than configurations in which both neurons are of the same type.

Another apparent exception is the case of $n = 1$, although in this instance, only two values are possible for localization: $\lambda = 0$ and $\lambda = 1$. Since $\lambda_{\mathbf{1}} = 0.56$, this indicates that a single oscillating neuron outperforms a single localized neuron. This phenomenon is illustrated more clearly in Fig. 2.9. We observe that the majority of inputs labeled $y = -1$ are correctly classified by oscillating neurons, while most inputs labeled $y = 1$ go unrecognized; however, a significant portion of these are still correctly classified.

Conversely, the errors made by localized neurons on the $y = -1$ inputs are comparable to those made by oscillating neurons on the $y = 1$ inputs. However, the number of errors made by oscillating neurons on $y = -1$ inputs is significantly higher than those made on $y = 1$ inputs. We hypothesize that a similar explanation may account for the trend observed in the case of $n = 2$, but further analysis is necessary.

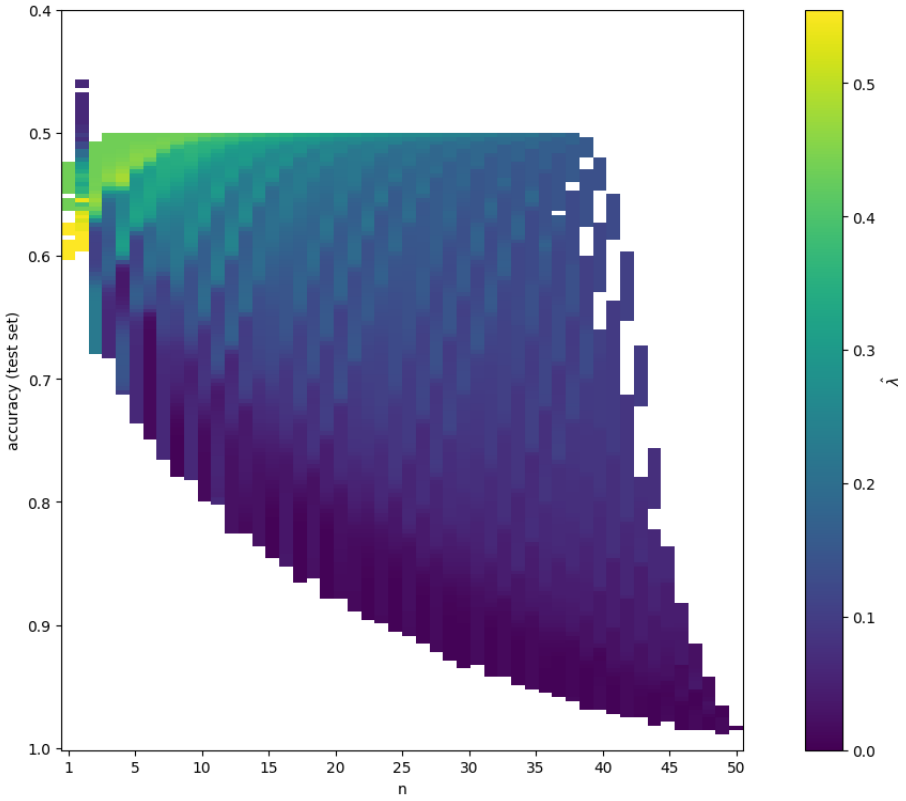


Figure 2.19: For each possible value of energy (or equivalently accuracy) and number of active neurons n, we display $\hat{\lambda} = \langle |\lambda - \lambda^*| \rangle$, representing the average distance from the optimal localization $\lambda^* = \lambda_{\mathbf{1}} = 0.567$.

Chapter 3

Methods and Materials

3.1 Tasks

3.1.1 NLGP

This is a binary classification task presented in [3]. The dataset used in this task is synthetically generated, allowing us to control the size of the task and its difficulty.

A data vector of the NLGP dataset is given by:

$$\mathbf{x}^\mu = \frac{\psi(g\mathbf{z}^\mu)}{Z(g)} \tag{3.1}$$

Where:

- $\mathbf{z}^\mu$ is a zero-mean Gaussian vector of length L and covariance matrix $C^\mu_{ij} = \langle z^\mu_i z^\mu_j \rangle = e^{-|i-j|/\xi^\mu}$, with $i, j = 1, 2, \ldots, L$. The covariance thus only depends on the distance between sites i and j, given by $|i-j|$; we also enforce periodic boundary conditions.
- $g > 0$ is a gain factor; it controls the sharpness in the images: a large gain factor results in images with sharp edges and important non-Gaussian statistics, while images with a small gain factor are close to Gaussians in distribution. Throughout this work we took $g = 3$, unless otherwise specified.
- $Z(g)$ is a normalization factor chosen such that $\mathrm{Var}(\mathbf{x}) = 1$ for all values of g.
- ψ is a non linear function; throughout this work, we took ψ to be a symmetric saturating function $\psi(z) = \mathrm{erf}(z/\sqrt{2})$, for which $Z(g)^2 = \frac{2}{\pi} \arcsin\left(\frac{g^2}{1+g^2}\right)$.

For all NLGP datasets generated for this work the inputs are one dimensional. They are divided in $M = 2$ classes, labeled $y = \pm 1$, that differ by correlation length $\xi^{\pm}$ between pixels. The number of inputs with label $y = 1$ is equal to the number of inputs with label $y = -1$.

In Fig. 3.1 we show some examples of NLGP inputs for different values of ξ.

3.1.2 GP

Given an NLGP dataset we can create its Gaussian clone (GP) by drawing inputs from a Gaussian distribution with mean zero and the same covariance as the corresponding NLGP. The covariance of the NLGP can be evaluated analytically for $\psi(z) = \mathrm{erf}(z/\sqrt{2})$ and reads

$$\langle \kappa_i^\mu \kappa_j^\mu \rangle = \frac{2}{\pi Z(g)} \arcsin\left(\frac{g^2}{1+g^2} C_{ij}^\mu\right) \tag{3.2}$$

where we have used that fact that $C_{ii} = 1$. The experiments on GPs are thus not performed on the Gaussian variables $\mathbf{z}$; they are performed on Gaussian random variables with covariance given by the equation above. In this way, we exclude the possibility that the change in the two-point correlation function from applying the nonlinearity ψ is responsible for the emergence of receptive fields (RFs).

In Fig. 3.2 we show some examples of GP inputs for different values of ξ.

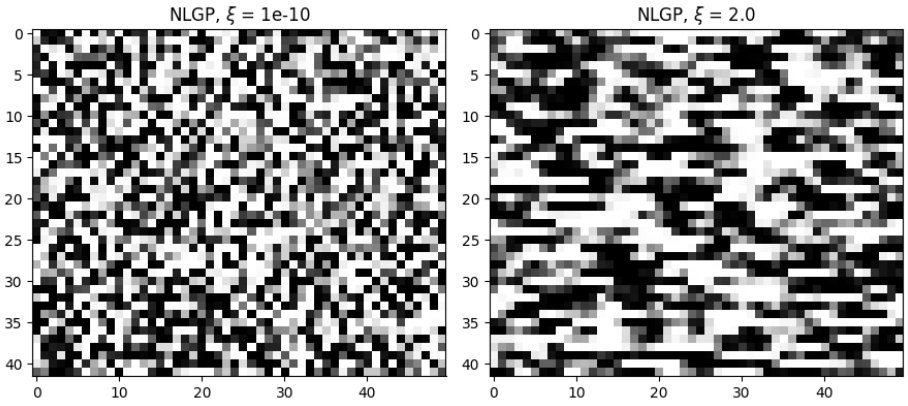


Figure 3.1: Each row represents a D-dimensional input displayed in grayscale (here, $D = 50$). We are displaying 42 inputs for each class out of the N available inputs in the dataset.

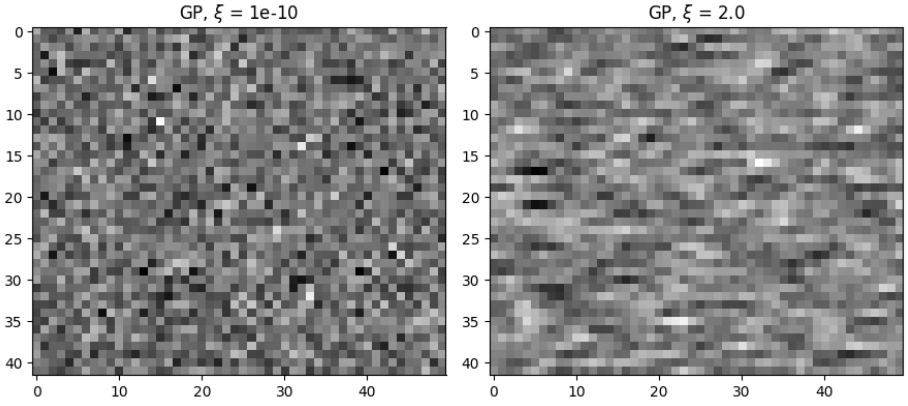


Figure 3.2: Each row represents a D-dimensional input displayed in grayscale (here, $D = 50$). We are displaying 42 inputs for each class out of the N available inputs in the dataset.

3.2 Statistics of the Model

3.2.1 Layers

Weights distribution

It's the histogram of all entries in W. During training, the initial distribution ($\sim\mathcal{N}(0, \frac{1}{D})$) gradually broadens. This widening continues until the regularization factor in the loss function becomes significant, causing the distribution to stabilize.

Mean and variance of weight vectors

$$m_k = \frac{1}{D}\sum_{i=1}^{D} w_{ki} \qquad \sigma_k^2 = \frac{1}{D}\sum_{i=1}^{D}(w_{ki} - m_k)^2 \tag{3.3}$$

We keep track of them to better understand the dynamics of the model; looking at them, indeed, we can see if the network reach a relatively steady state.

Inverse participation ratio (IPR)

$$\mathrm{IPR}(\mathbf{w}_k) = \frac{\sum_{i=1}^{D} w_{ki}^4}{\left(\sum_{i=1}^{D} w_{ki}^2\right)^2} \tag{3.4}$$

As has already been done in [3], we used IPR to distinguish localized RFs by oscillating ones. Indeed it can be viewed as an indicator of the number of non-zero components in a vector, in particular localized weight vectors will feature an high IPR.

3.2.2 Internal representations

Mean and variance

The mean and variance of the internal representations are computed over all the reference dataset, namely:

$$m_k = \frac{1}{N}\sum_{\mu=1}^{N} h_k(\mathbf{x}^\mu) \qquad \sigma_k^2 = \frac{1}{N}\sum_{\mu=1}^{N} \left(h_k(\mathbf{x}^\mu) - m_k\right)^2 \tag{3.5}$$

Dimensionality

Given a dataset $\mathcal{D}$, each input $\mathbf{x}^\mu$ has an internal representation h^μ that is a point in a K-dimensional space. To estimate the effective dimensionality of these internal representations, we employed Principal Component Analysis (PCA). PCA allows us to identify the orthogonal dimensions that contribute most to the variance within these internal representations. The effective dimensionality we obtain is then a quantitative measure of how much the internal representations are spread across the different dimensions.

We computed the covariance matrix of all the internal representations as:

$$C_{kj} = \langle (h_k - \langle h_k \rangle)(h_j - \langle h_j \rangle) \rangle \tag{3.6}$$

Where $\langle \cdot \rangle$ is the average over all the input in the data set. Throughout this work, when we refer to the effective dimensionality we always consider the internal representations of the training set.

The eigenvalues $\{\lambda\}$ of the covariance matrix inform us on what are the most relevant dimension. The higher contribution to the variance will be along the eigenvector corresponding to the higher eigenvalue and so on.

We normalized the eigenvalues (i.e. $\hat{\lambda}_k = \frac{\lambda_k}{\sum_k \lambda_k}$) and we compute the effective dimension as the Shannon entropy of the distribution so obtained:

$$d_{eff} = -\sum_k \hat{\lambda}_k \log \hat{\lambda}_k \tag{3.7}$$

Shannon entropy measures the uncertainty of a given distribution. For example, a uniform distribution has high entropy, while a distribution concentrated around a few values has low entropy. For that reason we considered this a good measure of how well the variance is distributed along the different dimensions.

3.3 Wang Landau Algorithm

The Wang Landau (WL) algorithm is a non-markovian Monte Carlo (MC) sampling method. It was introduced for the first time by Fugao Wang and D. P. Landau in the context of condensed matter physics [8], but while such method has been already used to study coarse-graining of macromolecules [2, 6], it is

virtually unknown to the Machine Learning and Computational Neuroscience community.

More conventional MC methods (e.g. Metropolis-Hastings algorithm) are able to sample from a canonical distribution at a given temperature ($P(E) = g(E)e^{-E/(K_B T)}$). This is achieved by using a Markov process which asymptotically reaches such distribution.

On the other hand the WL algorithm is non-markovian, but enable us to estimate the whole density of state $g(E)$. This is possible by performing a random walk which produces a flat histogram in energy space. Please note that this is equivalent to say that energy barriers are "invisible" to the algorithm, thus states that are difficult to reach by the Metropolis-Hasting algorithm can be can be easily reached by the WL algorithm.

The main idea behind the WL algorithm is that a random walk in energy space with a probability proportional to the reciprocal of the density of states $\frac{1}{g(E)}$ generates a flat histogram for the energy distribution. This means that we can use the "flatness" of the histogram generated by the random walk to evaluate the goodness of our estimate for the density of state.

Since the density of state is a priori unknown, we initially set $g(E) = 1$ for all energies E (uninformative prior). The system is set in the initial state with corresponding energy E_i, thus we have to update the histogram $H(E)$ and the density of state $g(E)$. Indeed, in general, at each iteration the following update are made: $H(E_s) \to H(E_s) + 1$ and $g(E_s) \to g(E_s)f$, where E_s is the energy of the current state and f is the modification factor that allow us to improve the estimate for $g(E)$. The modification factor f will be refined while the algorithm is running, but a typical initial choice is $f_0 = e$. Then we start our random walk by proposing a new state (e.g. if our system is an Ising model, that means flipping spins randomly). The probability of accepting the proposed move is given by:

$$p(E_{old} \to E_{new}) = \min\left[\frac{g(E_{old})}{g(E_{new})}, 1\right] \tag{3.8}$$

Recalling that the density of state and the entropy are related according to $g(E) \propto e^{S(E)}$, this probability can be equivalently written as:

$$p(E_{old} \to E_{new}) = \min\left[e^{-\Delta S}, 1\right], \qquad \text{with: } \Delta S = S(E_{new}) - S(E_{old}) \tag{3.9}$$

The random walk goes on until the histogram $H(E)$ is "flat". Of course it's impossible to obtain a perfectly flat histogram, when we define an histogram as "flat" we mean that, for all possible E, $H(E)$ never exceed a certain percentage distance from the average of the histogram $\langle H(E)\rangle$, i.e.:

$$\left|\frac{H(E) - \langle H(E)\rangle}{\langle H(E)\rangle}\right| < 1 - p_{flat} \tag{3.10}$$

We set $p_{flat} = 0.9$, except when otherwise stated.

Once the flatness condition is satisfied, we reduce the modification factor according to $f_{t+1} = \sqrt{f_t}$ (any function that monotonically decreases to 1 will do) and we reset the histogram $H(E) = 0$ for all E.

We iterate this process until the modification factor is smaller than some final value f_{final}. We set $f_{final} = \exp(10^{-5})$, except when otherwise stated.

A pseudo-code of the algorithm is provided (Algorithm 1).

Algorithm 1 Wang-Landau

Initialize:
$\log g(E) \leftarrow 0 \quad \forall E$
$H(E) \leftarrow 0 \quad \forall E$
$\log f \leftarrow \log f_0$

while $\log f > \log f_{final}$ **do**
    $C \leftarrow$ Initial configuration
    $E \leftarrow$ Compute energy of C
    while flat histogram criterion is not met **do**
        Generate a trial move C'
        $E' \leftarrow$ Compute energy of C'
        if $\log g(E) - \log g(E') > \log(\text{rand}(0, 1))$ **then**
            Accept move: $C \leftarrow C'$, $E \leftarrow E'$
        else
            Reject move
        end if
        $\log g(E) \leftarrow \log g(E) + \log f$
        $H(E) \leftarrow H(E) + 1$
    end while
    $\log f \leftarrow \log f/2$
    $H(E) \leftarrow 0 \quad \forall E$
end while

Output: Estimated log density of states $\log g(E)$

3.3.1 Wang Landau 2D

Throughout our work, other then computing standard density of states $g(E)$, we were also interested in computing joint density of states (JDOS), such as $g(E_{\text{train}}, E_{\text{test}})$, $g(E, n)$ and $g(E, n, \lambda)$ (see section 2.3).

In order to compute these quantities we employed two different approaches. To compute $g(E_{\text{train}}, E_{\text{test}})$ we just generalized the WL algorithm to the 2D case, this means that we now deal with a two dimensional histogram $H(E_{train}, E_{test})$ and when a configuration will be proposed we have to compute both the energy with respect to the training test and the test set.

To compute these quantities, we employed two different approaches. For $g(E_{\text{train}}, E_{\text{test}})$, we extended the Wang-Landau algorithm to the two-dimensional

case. This approach involves working with an histogram $H(E_{\text{train}}, E_{\text{test}})$ which is two-dimensional. When a new configuration is proposed, we calculate the energy with respect to both the training set and the test set, and then update the histogram accordingly.

On the other hand, when one of the variables involved is n, we adopted a different strategy. For each possible value of n (i.e., $\forall n \in \{1, \ldots, K\}$), we used a modified version of the algorithm in which every proposed configuration has exactly n active neurons in the hidden layer. For instance, to compute $g(E, n)$, we first calculate $g(E|n = i)$ for all $i \in \{1, \ldots, K\}$. Then, using the fact that $g(i) = \binom{k}{i}$, we can obtain $g(E, n)$ as:

$$g(E, n) = \sum_{i=1}^{K} g(E|n = i)g(i) \tag{3.11}$$

The main advantage of this approach is that we run multiple one-dimensional WL algorithms in parallel, which significantly speeds up the process. Additionally, when $g(i)$ is sufficiently small, we can compute $g(E|n = i)$ exhaustively. In fact, there is a threshold value below which an exhaustive search for $g(E)$ is faster than the WL algorithm.

When computing $g(E, n, \lambda)$, we tried to combined both the approaches: for each value of n, we implemented a two-dimensional version of the WL algorithm. Similarly to the $g(E, n)$ case, we can reconstruct the full joint distribution as:

$$g(E, n, \lambda) = \sum_{i=1}^{K} g(E, \lambda|n = i)\, g(i) \tag{3.12}$$

However, although this approach might be optimal for higher values of K, in the specific case discussed in 2.4, we decided to reconstruct the joint density of states $g(E, n, \lambda)$ by computing the conditional density of states $g(E|n, l)$ for all possible values of n and l, where l is the number of localized active neurons.

In order to reconstruct the full 3-dimensional density of state we need take into account the degeneracy factor $g(n, l)$, i.e. how many configurations have n active neurons and l localized neurons. This is given by: The distribution $g(n, l)$ is given by:

$$g(n, l) = \binom{|\mathcal{L}|}{l} \binom{K - |\mathcal{L}|}{n - l} \tag{3.13}$$

Where $|\mathcal{L}|$ is the total number of localized neurons.

Ultimately we can notice that, since $\lambda = \frac{l}{n}$, defining a tuple (n, l) is equivalent to defining a tuple (n, λ) and the introduction of l was mad just for operative reasons. For this reason $g(n, l) = g(n, \lambda)$ and $g(E|n, l) = g(E|n, \lambda)$ and we can write:

$$g(E, n, \lambda) \propto \sum_{n,\lambda} g(E|n, \lambda)\, g(n, \lambda) \tag{3.14}$$

3.4 Weight vectors clustering

In order to compare different weight vectors $\mathbf{w}_k$, we introduce a similarity measure that is invariant to translation, similarly to what was done in [3]. Given two weight vectors $\mathbf{w}_k$ and $\mathbf{w}_l$, we normalize them, then apply the Fourier transform to obtain $\tilde{\mathbf{w}}_k$ and $\tilde{\mathbf{w}}_l$, respectively. Their overlap is given by:

$$\tilde{q}_{kl} = \frac{1}{D}|\tilde{\mathbf{w}}_k||\tilde{\mathbf{w}}_l| \tag{3.15}$$

By taking the absolute value of the Fourier transform of the weight vectors, $\tilde{q}_{kl}$ becomes invariant to translation, as the phase information is removed. We then define a pairwise distance matrix of size $K \times K$, with entries given by:

$$d_{kl} = 1 - \tilde{q}_{kl} \tag{3.16}$$

Finally, we cluster the weight vectors using complete-linkage clustering. The dendrogram is cut at half the maximum distance in the distance matrix.

Chapter 4

Conclusion

In this work we showed the potential of using the Wang-Landau algorithm to explore the resilience of neural networks in case of neuronal death.

In the case under consideration, the introduction of order parameters such as the number of active neurons n and their localization ratio λ proved useful in describing the characteristics of configurations that best performed the task.

The results we obtained inherent the (joint) density of states of the system are in agreement with the interpretation of the internal representation of the network. Localized neurons primarily produce negative outputs, which become more negative for inputs with short correlation length; oscillating neurons mainly generate positive outputs, producing for inputs with longer correlation length stronger positive responses. Since each hidden neuron's output contributes uniformly to the total output, in the event of neuronal death, maintaining the balance between localized and oscillating neurons is important to ensure the network can still perform the task effectively. When this balance is disrupted, the network's performance on the task suffers significantly.

A similar approach could be used to study another scenario involving two different types of neurons, such as in MLPs with a layer of excitatory/inhibitory (E/I) neurons. Investigating the robustness of such networks would be particularly interesting, as they are more closely aligned with biological neuronal circuits.

Acknowledgments

Desidero ringraziare il mio surpervisor, Alessandro Ingrosso, per tutto il supporto che mi ha sempre dimostrato nel corso di questo progetto. In particolar modo per le conversazioni in pausa pranzo all'ICTP durante i quali mi ha aiutato a orientarmi meglio nel mondo della ricerca e offerto nuovi spunti di riflessione. Darmi la possibilità di entrare in contatto con un mondo così intellettualmente stimolante ha aumentato il mio desiderio di continuare ad ampliare le mie conoscenze.

Un ringraziamento speciale va a Camilla, la mia ragazza. Lei più di tutti conosce la fatica e l'impegno che mi sono serviti per raggiungere questo traguardo, il tuo supporto è stato fondamentale e non penso ce l'avrei fatta senza di te. Ti amo.

Ringrazio tutta la mia famiglia che mi ha sempre voluto bene e che troppe volte tendo a dare per scontata, grazie per tutto quello che avete sempre fatto per me. In particolare voglio ringraziare i miei genitori e mia sorella, sono molto felice del rapporto che abbiamo costruito negli anni, ma mi rendo conto che difficilmente lo dimostro apertamente. Vi voglio bene, ogni tanto è importante ricordarlo.

In fine voglio ringraziare tutti i miei amici, che in tutti questi anni mi hanno aiutato a staccare la testa e rilassarmi con loro, ma che al tempo stesso si sono rivelati sempre presenti nei momenti di difficoltà. Grazie a tutti (cit.).

Appendix A

Emergence of Localized Receptive Fields: impact of training set size and number of hidden neurons

In Section 2.4, we considered a specific case where receptive fields (RFs) split into "localized" and "oscillating" patterns after training. However, this behavior is not universally applicable.

We observe that the emergence of localized and oscillating neurons is significantly influenced by the number of neurons in the hidden layer K and the size of the training set α_{train}. To better understand the conditions under which these patterns arise, we can train networks with varying values of K and α_{train} while keeping all other parameters fixed.

Utilizing the clustering methods described in 3.4, we can classify the RFs into different shapes. However, it is essential to quantify how "localized" the weight vectors are for each class and assign an index to each trained model indicating the level of localization of its hidden neurons—what we refer to as a localization proxy.

We found that, under identical parameter settings and initial conditions, training a network to solve the dual GP task results in final RFs that do not localize in the same way as those in the NLGP case. While both tasks yield two distinct classes of RFs, we observe what can be characterized as fast and slow oscillations rather than a clear oscillating versus localized pattern (see Fig. A.1 and Fig. A.2).

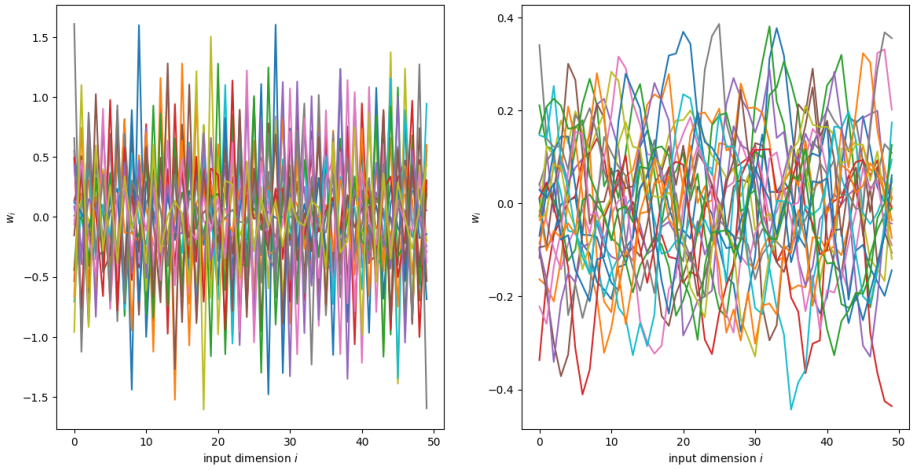


Figure A.1: Weight vectors in a network trained on the GP task. $K = 42$, $\alpha_{train} = 500.0$, $\xi^- = 1$, $\xi^+ = 2$

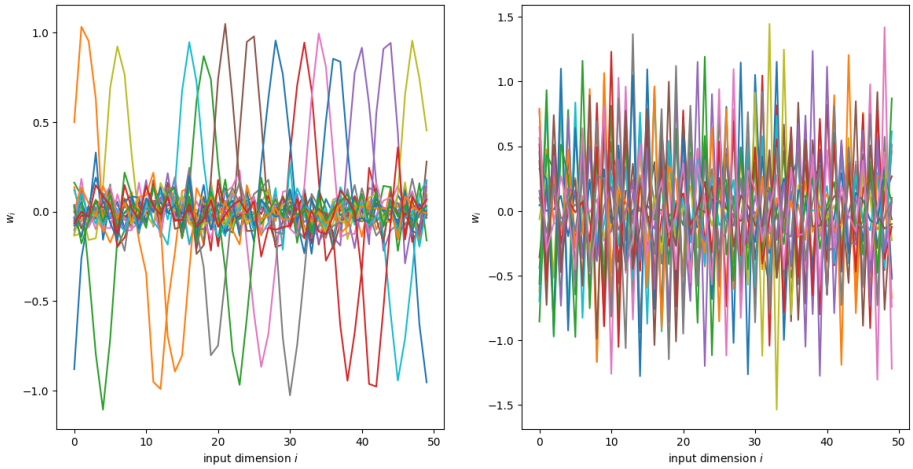


Figure A.2: Weight vectors in a network trained on the NLGP task. $K = 42$, $\alpha_{train} = 500.0$, $\xi^- = 1$, $\xi^+ = 2$

To leverage this phenomenon, we train an identical network on the dual GP task for each network trained on the NLGP task, maintaining the same parameters and initial conditions. We then employ the difference in the average inverse participation ratio (IPR) (3.2.1) between the NLGP and GP cases as our localization proxy.

The observed trend indicates that networks with lower K and larger training sets exhibit higher localization. However, localization does not appear to be

essential for achieving good generalization. For instance, networks with a high K can attain high accuracy even in the absence of localized RFs. This observation may suggest that the task was too simple, eliminating the necessity for this feature to develop. To explore this further, we replicated the study with different values of (ξ^-, ξ^+), specifically considering the cases $(\xi^- = 10^{-10}, \xi^+ = 2)$, $(\xi^- = 1, \xi^+ = 2)$, and $(\xi^- = 1.6, \xi^+ = 2)$, which we categorize as "easy," "medium," and "hard," respectively.

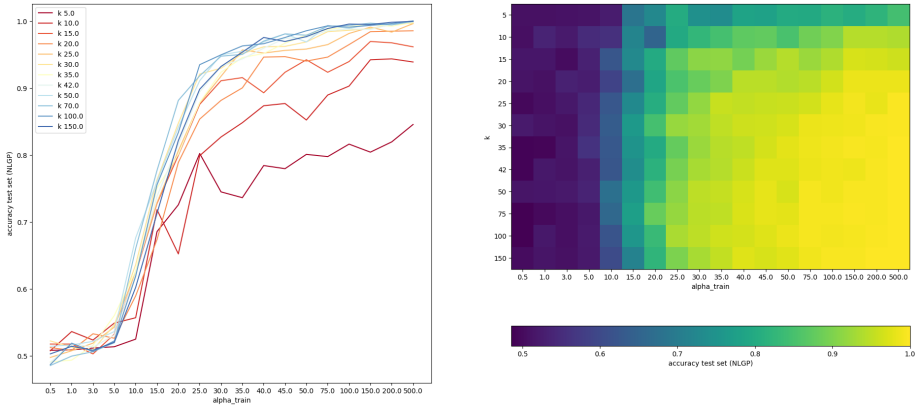


Figure A.3: Accuracy on the test set w.r.t K and α_{train} for a network trained on NLGP task. $\xi^- = 10^{-10}$, $\xi^+ = 2$ ("easy")

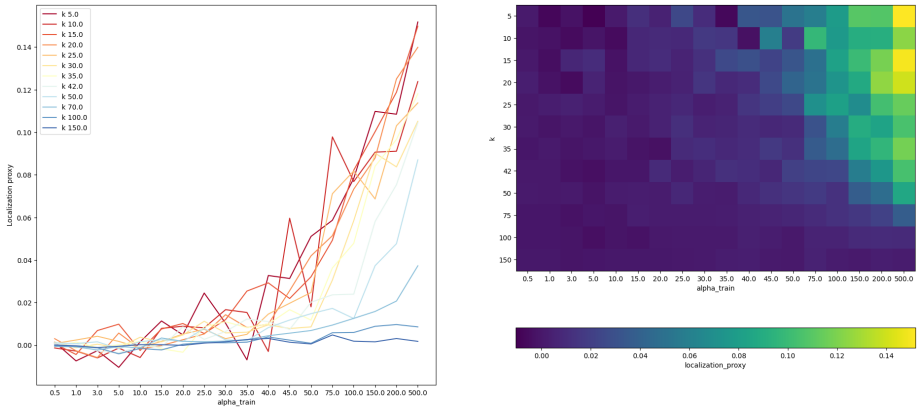


Figure A.4: Localization proxy w.r.t K and α_{train}. $\xi^- = 10^{-10}$, $\xi^+ = 2$ ("easy")

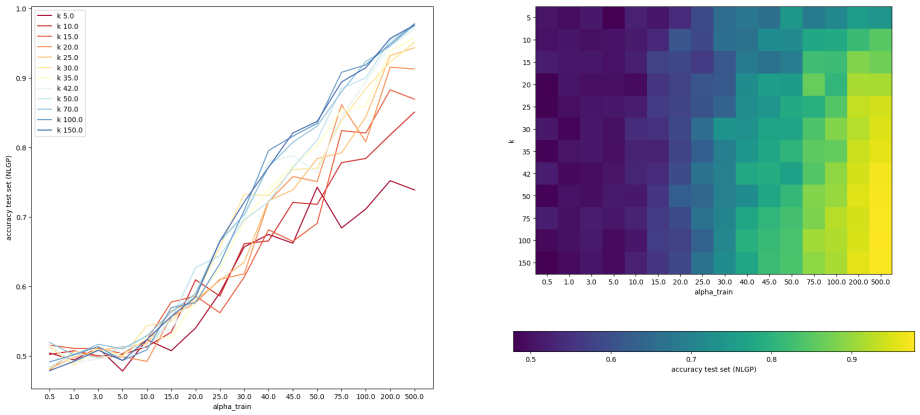


Figure A.5: Accuracy on the test set w.r.t K and α_{train} for a network trained on NLGP task. $\xi^- = 1$, $\xi^+ = 2$ ("medium")

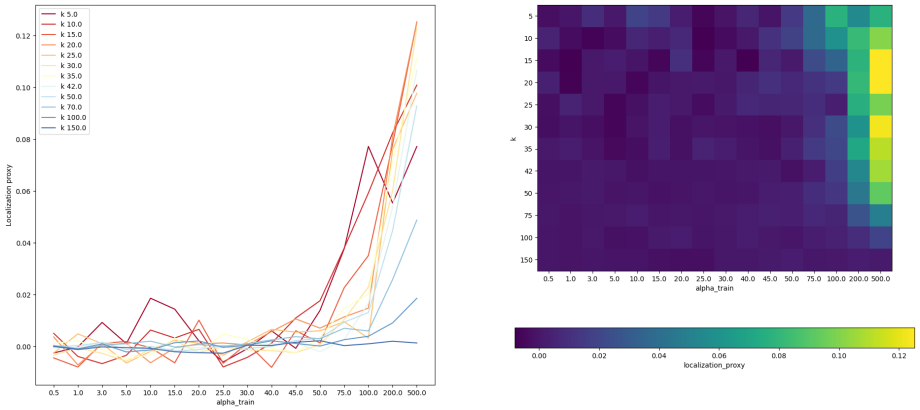


Figure A.6: Localization proxy w.r.t K and α_{train}. $\xi^- = 1$, $\xi^+ = 2$ ("medium")

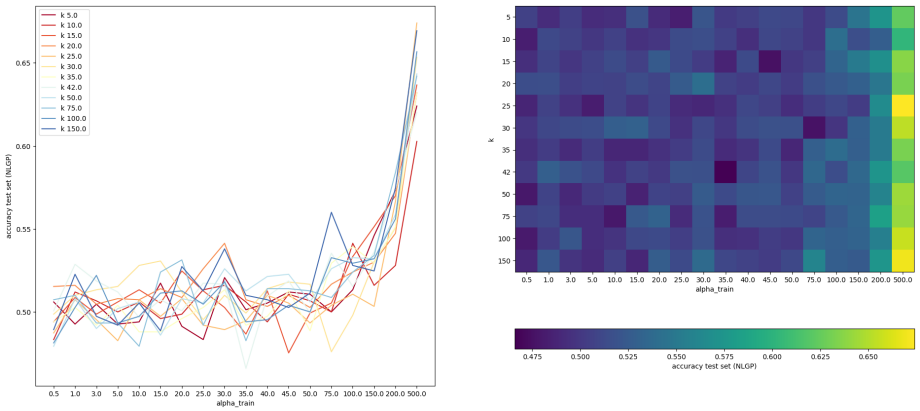


Figure A.7: Accuracy on the test set w.r.t K and α_{train} for a network trained on NLGP task. $\xi^- = 1.6$, $\xi^+ = 2$ ("hard")

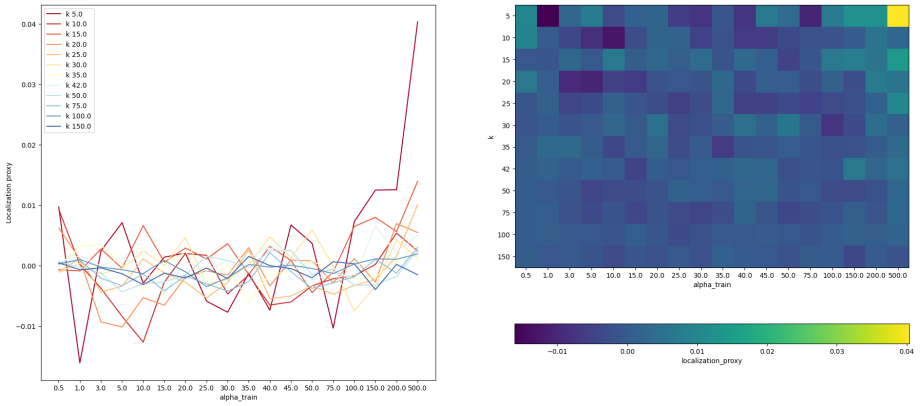


Figure A.8: Localization proxy w.r.t K and α_{train}. $\xi^- = 1.6$, $\xi^+ = 2$ ("hard")

In conclusion, further investigation is necessary to explore a broader range of K and α_{train}, along with additional values of ξ, to gain deeper insights into the conditions influencing the emergence of localized and oscillating neurons. Nevertheless, the general trend indicates that localization tends to emerge when α is sufficiently high, K is not excessively large, and the task presents the right level of difficulty: challenging enough for the network to learn but still allowing for good accuracy on the test set.

Bibliography

[1] Simone Ciceri, Lorenzo Cassani, Matteo Osella, Pietro Rotondo, Filippo Valle, and Marco Gherardi. Inversion dynamics of class manifolds in deep learning reveals tradeoffs underlying generalization. *Nature Machine Intelligence*, 6(1):40–47, January 2024.

[2] Roi Holtzman, Marco Giulini, and Raffaello Potestio. Making sense of complex systems through resolution, relevance, and mapping entropy. *Phys. Rev. E*, 106:044101, Oct 2022.

[3] Alessandro Ingrosso and Sebastian Goldt. Data-driven emergence of convolutional structure in neural networks. *Proceedings of the National Academy of Sciences*, 119(40), September 2022.

[4] Christoph Junghans, Danny Perez, and Thomas Vogel. Molecular dynamics in the multicanonical ensemble: Equivalence of wang–landau sampling, statistical temperature molecular dynamics, and metadynamics. *Journal of Chemical Theory and Computation*, 10(5):1843–1847, 2014. PMID: 26580515.

[5] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo Methods*. Wiley-VCH, 2008.

[6] Roberto Menichetti, Marco Giulini, and Raffaello Potestio. A journey through mapping space: characterising the statistical and metric properties of reduced representations of macromolecules. *Eur. Phys. J. B*, 94, Oct 2021.

[7] Michael Nielsen. *Neural Networks and Deep Learning*. Self-published, 2015.

[8] Fugao Wang and D. P. Landau. Efficient, multiple-range random walk algorithm to calculate the density of states. *Phys. Rev. Lett.*, 86:2050–2053, Mar 2001.