Università degli Studi di Verona

FACOLTÀ DI SCIENZE E INGEGNERIA
Corso di Laurea in Mathematics

Master Thesis

Kinetic Description of Neural Differential Equations

Candidate:
Valerio Flora
Matricola VR481426

Relatore:
Giacomo Albi

Anno Accademico 2022/2023

Alla donna più importante della mia vita,
mamma.

Contents

Introduction

In recent years, neural networks have emerged as a powerful tool for solving a wide range of complex problems, spanning from regression and classification to image recognition and natural language processing. The ability of neural networks to learn from large datasets and generalize their knowledge to new inputs has revolutionized various fields, including computer vision, robotics, and finance. However, a deeper understanding of the underlying mathematical principles and connections between neural networks and other mathematical frameworks can unlock even more potential and open up new avenues for research and application.

One intriguing connection lies between neural networks and partial differential equations (PDEs). PDEs are fundamental mathematical models used to describe various physical, biological, and engineering phenomena.
They involve derivatives with respect to multiple variables and provide a powerful framework for modeling continuous systems.
Neural networks, on the other hand, excel at approximating complex functions and capturing intricate patterns within datasets.
The link between neural networks and PDEs becomes particularly fascinating when considering kinetic neural differential equations. Kinetic equations describe the evolution of systems consisting of a large number of particles, where the behavior of each individual particle is influenced by its interactions with others or itself.
By extending the concept of kinetic equations to the realm of neural networks, we can unlock a new level of understanding and control over these systems.
The aim of this master's thesis is to explore the importance of neural networks in tackling a diverse set of tasks, ranging from regression to classification, while establishing their connection with partial differential equations.
Specifically, we will delve into the development of kinetic neural differential equations, which model the behavior of neural networks as dynamic systems of interacting particles.
The framework will be developed by extending traditional mean-field equations, which provide a macro-description of large ensembles of particles, to capture the individual behavior of neural network components.

By examining the dynamics of neural networks at the particle level, we aim to gain insights into their learning capabilities, stability, and generalization abilities. Furthermore, the exploration of kinetic neural differential equations holds the promise of novel algorithms and techniques for training neural networks, as well as a deeper understanding of their role in the broader landscape of mathematical modeling.
In other words, the approach of this master's thesis is to approximate the solu-

tion of a class of PDEs *(Vlasov-type equation)* by leveraging the potential of a specific class of neural networks and under theoretical certainty regarding the convergence of the solutions.
In greater detail, the primary objective of this thesis is to initially provide a comprehensive explanation within a low-dimensional context. Subsequently, the aim is to extend this explanation to encompass more complex multivariate scenarios.
Within this conceptual framework, the ultimate goal is to demonstrate enhanced performance in higher dimensions through the utilization of a kinetic formulation of the problem, as opposed to relying solely on mean-field equations.
Next, we will delve into a Boltzmann description of the neural network, examining its applicability in noisy settings and getting an asymptotic limit leading to a Fokker-Planck equation. This exploration will lead to an extension of the model, presenting a broader solution compared to the previous results.
Delving deeper into the realm of noisy dynamics, we will explore aspects such as distribution fitting and classification tasks.
To substantiate these theoretical developments, we will employ numerical simulations implemented using MATLAB.
Summarizing, this thesis will provide a comprehensive investigation into the relationship between neural networks and partial differential equations, culminating in the development of kinetic neural differential equations.

Chapter 1

Neural Networks

In this chapter, we will explore the main neural networks that will be the focus of this thesis. We begin by introducing Neural Differential Equations, which form the foundation of the model. Next, we will investigate the connection between these types of networks and Residual Neural Networks *(ResNets)*, particularly in the context of discretization. Residual Neural Networks will serve as the central component of the model, as the goal of this work is to develop a class of ResNets that offer a concise and efficient interpretation of the learning process.

1.1 Neural Differential Equations

Neural Differential Equations (NDEs) are a class of mathematical models that combine principles from neural networks and differential equations. NDEs offer a framework for describing the dynamics and behavior of complex systems by representing them as continuous-time differential equations that are influenced by neural network architectures.

Traditional differential equations describe how variables change with respect to time or other independent variables. They have been widely used to model a wide range of natural phenomena in physics, biology, and engineering. On the other hand, neural networks are powerful machine learning models that can learn complex patterns and relationships from data.
NDEs bridge the gap between these two frameworks by leveraging the expressive power of neural networks while retaining the interpretability and mathematical structure of differential equations. Instead of directly specifying the form of the differential equation, NDEs learn the dynamics of the system using neural networks. The neural network acts as a continuous-time function approximator, approximating the derivative or the rate of change of the variables at each point in time.

The key idea behind NDEs is that the behavior of a system can be learned by training a neural network to predict the derivatives of the variables based on the current state. The neural network takes the current state as input and outputs the rate of change, which is then integrated to obtain the next state

of the system.
In other words, a neural differential equation is a differential equation using a neural network to parameterise the vector field.

Let's give a formal explanation of *neural ordinary differential equation* with a canonical example:

$$y(0) = y_0 \qquad \frac{dy}{dt}(t) = f_\theta(t, y(t)) \tag{1.1}$$

where θ represents a vector of learnt parameters, i.e. $f_\theta : \mathbb{R} \times \mathbb{R}^{d_1 \times d_2 \cdots \times d_k} \to \mathbb{R}^{d_1 \times d_2 \cdots \times d_k}$ is any standard neural architecture and any $y : [0, T] \to \mathbb{R}^{d_1 \times d_2 \cdots \times d_k}$ is the solution.
For many applications f_θ will just be a simple feedforward network.
In this general case the approach is proceed by taking $y(0) = y_0$ as the initial condition of the neural ODE and evolve the ODE until some time T, i.e. a hidden state at any time index $y(t)$ can be obtained by solving the initial value problem (1.1).
To solve the IVP, a black-box differential equation solver can be employed and the hidden states can be computed with the desired accuracy.
Here is the computation graph of the simple neural ODE:

Figure 1.1: Computation graph of simple neural ode

The structure, of course, will be in function of the task we are dealing with, i.e. the activation function, the affine transformation and the black-box will depend on the goal of the network. The parameters of the model are θ. The computation graph may be backpropagated through and trained via stochastic gradient descent in the usual way.
Globally, we can observe we are dealing with a neural network f_θ, embedded in a differential equation for y, embedded in a neural network (the overall computation graph).

NDE represents a unified, continuous computational entity. This eliminates the need to determine the number of layers required to approximate a particular function. Instead, it necessitates specifying the desired level of accuracy, allowing the network to autonomously fine-tune itself within that defined margin of error.
Rather than viewing the network as discrete building blocks, we perceive it as a continuous function. This perspective offers advantages such as quicker testing times at the expense of longer training durations. It introduces a precision-speed tradeoff, delivering enhanced accuracy in tasks like time-series prediction, optimizing the overall process, and computing gradients with consistent memory overhead.

In summary, Neural Differential Equations provide a compelling approach that combines the best aspects of both neural networks and classical differential equations. The neural network-like structure of NDEs offers the advantages of high-capacity function approximation and ease of trainability.

On the other hand, the differential equation-like structure of NDEs brings several benefits such as strong priors on the model space, memory efficiency, and a well-established literature that provides a deep theoretical understanding.

Compared to the classical differential equation literature, NDEs exhibit unparalleled modeling capacity. They allow for more flexible and expressive representations, enabling the modeling of complex phenomena.
In contrast, when compared to the modern deep learning literature, NDEs provide a coherent and principled framework that offers insights into what constitutes a good model. This theoretical foundation helps guide the design and interpretation of NDEs, allowing a deeper understanding of their capabilities and limitations.

Since they provide a powerful framework for modeling complex systems, NDEs have found applications in various fields, including physics, biology, finance, control systems, and more. In a broader context, these structures are highly appropriate for time-series data, where time assumes a critical role, and the primary objective is to understand how data evolves over it.

1.2 Link between NDEs and Residual Neural Networks

In the previous section we have just seen how neural differential equations may be approached via traditional mathematical modelling. They may also be arrived at via modern deep learning.
The goal of this part is to explain the correlation between NDEs and Residual Neural Network, which will be the type we will treat in our model.
The formulation of a residual network is the following:

$$y^{(k+1)} = y^{(k)} + f_\theta(k, y^{(k)}) \tag{1.2}$$

where $f_\theta(k, \cdot)$ is the $k - th$ residual block.
Now recalling the neural ODE

$$\frac{dy}{dt}(t) = f_\theta(t, y(t)$$

and discretizing it via the explicit Euler method at times t_k uniformly separated by Δt we obtain:

$$\frac{y(t_{k+1}) - y(t_k)}{\Delta t} \approx \frac{dy}{dt}(t_k) = f_\theta(t_k, y(t_k))$$

so that

$$y(t_{k+1}) = y(t_k) + \Delta t f_\theta(t_k, y(t_k))$$

Absorbing Δt into the f_θ, we recover the formulation of equation (1.2).

In this way we observed that neural ODEs are the continuous limit of residual networks. Starting from this now we are able to make other connections, moreover it is possible to work with discretization version of dynamics with ResNet and considering this link between the two fields.
These connections are really meaningful since many of the most effective and popular deep learning architectures resemble differential equations.

We can further expand our understanding of the ResNets architecture, and then we can introduce a specific subclass of ResNets for a more in-depth analysis.

1.3 Residual Neural Networks

The process of a ResNet can be shortly summarized as follows.
Given input data, also known as measurements, the ResNet architecture propagates them through layers and neurons to produce a final state using precise dynamics determined by the weights and biases parameters. This final state is commonly referred as the output or prediction of the network, which is expected to solve a specific learning task.

The success of a ResNet relies on selecting appropriate parameters, namely the weights and biases, which must be determined optimally.
This optimization procedure, known as training, involves fine-tuning the parameters based on a reference dataset which consists of input measurements along with corresponding target data, i.e. the desired output for each input.
During training, the objective is to compute optimal weights and biases by minimizing a suitable distance, also known as a *loss* or *cost function*, between the network's predictions and the given targets. The choice of the loss function depends on the specific learning task, such as classification or regression, and can include metrics like mean squared error, cross-entropy, or others.
To perform the training of ResNets, back-propagation algorithms based on the stochastic gradient descent *(SGD)* method are commonly used. The SGD algorithm iteratively adjusts the weights and bias of the network based on the gradients of the loss function with respect to these parameters. This process involves calculating the gradients through the network layers using the chain rule of calculus, and then updating the parameters in the direction that minimizes the loss.

By iteratively adjusting the parameters based on the reference dataset, ResNets can learn complex patterns and relationships within the data, enabling them to generalize well to unseen inputs and make accurate predictions on new data.
The trained ResNet model can then be deployed and used for various tasks,

such as image recognition, object detection, or any other application that requires learning from data. More in detail, in this thesis we will deal with classification and regression problems.

Overall, ResNets architecture provides a powerful and effective framework for addressing optimal control problems by mitigating the challenges associated with training deep neural networks. Their ability to propagate gradients, handle complex functions, and leverage residual learning makes them well-suited for learning optimal control policies and approximating value in various control domains.
This is possible thanks the *skip connections* which are crucial for building deeper and more efficient neural networks that can effectively learn from complex data. More in detail, observing figure (1.2), we can observe that the skip connections in deep residual neural networks facilitate a continuous information flow, allowing them to be naturally associated with neural differential equations, where the network's behavior is described as a continuous evolution governed by mathematical equations.
Residual Networks exhibit high efficiency when dealing with networks of significant depth. The depth of a neural network plays a critical role in its architecture, but training deeper networks poses challenges. However, the introduction of the residual learning framework addresses these difficulties and allows for the construction of substantially deeper networks, avoiding for example the vanishing gradient problem.
Consequently, ResNets exhibit enhanced performance across a wide range of tasks.

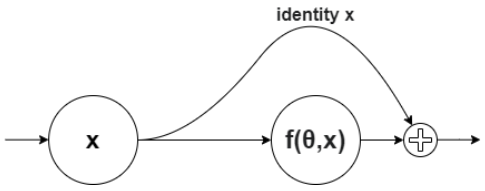


Figure 1.2: ResNet skip connection: at each layer the output is the input of the previous layers plus its function.

As a remark, a notable advantage of ResNets is their ability to achieve significantly greater depth compared to conventional networks, while maintaining a similar number of parameters (weights and biases). This means that ResNets can accommodate a larger number of layers without exponentially increasing the complexity of the model. This characteristic contributes to their enhanced performance and efficiency in learning complex representations.

As previously indicated, within the context of the thesis, we will be focusing on a distinct category of ResNet. Let's proceed to introduce this specific class.

1.3.1 Simplified Residual Neural Networks

Simplified Residual Neural Networks *(SimResNets)* compose a specific class of ResNets with the following assumption

Assumption 1 (SimResNet)**.** *The number of neurons in each layer is fixed and determined by the dimension of the input data.*

This structure can be well-suited for deriving neural differential equations due to their specific architectural characteristics. Here are a few reasons why these simplified ResNets are suitable for such derivations:

1. *Stable gradient flow*: the fixed number of neurons ensures consistent gradient flow throughout the network, as the skip connections allow the gradients to bypass a fixed number of layers at a time. This stability in gradient propagation helps in faster convergence during training.

2. *Control over network complexity*: by fixing the number of neurons in each layer, simplified ResNets provide control over the overall complexity of the network. This can be advantageous when deriving neural differential equations, as it allows for a more controlled exploration of the parameter space and the dynamics of the system.

3. *Analytical tractability*: Simplified ResNets with fixed neuron counts can lead to neural differential equations that are more amenable to mathematical analysis. The fixed architecture allows for clearer interpretations of the network dynamics and facilitates the derivation of mathematical equations that describe the behavior of the system. This analytical tractability can be particularly beneficial when studying the properties and characteristics of the resulting equations.

4. *Relationship to traditional differential equations*: Simplified ResNets with fixed neuron counts often resemble traditional differential equations in their structure. This similarity arises from the sequential nature of the network, where the output of each layer depends on the previous layer's output. By constraining the network architecture, the resulting equations may exhibit more direct correspondences to known differential equation models, making them more interpretable and facilitating the application of existing mathematical techniques.

5. *Cost-effective deployment*: the fixed-size layers in SimResNet can make it more feasible for deployment on resource-constrained devices, as the model size is predictable and can be optimized for specific hardware platforms.

Overall, simplified ResNets with fixed neuron counts offer advantages in terms of memory, control, tractability and interpretability. These characteristics make them well-suited for deriving neural differential equations and studying the dynamics of complex systems in a more analytically tractable and interpretable manner.

Even though SimResNet has a constraint on the number of nodes, it can still represent complex functions by stacking multiple layers with residuals. By increasing the depth of SimResNet, so adding more layers, it can approximate more intricate and complex functions. In other words, SimResNets have been proved to satisfy the universal approximating theorem for different class of functions (7) (8) and also for probability distributions (9).

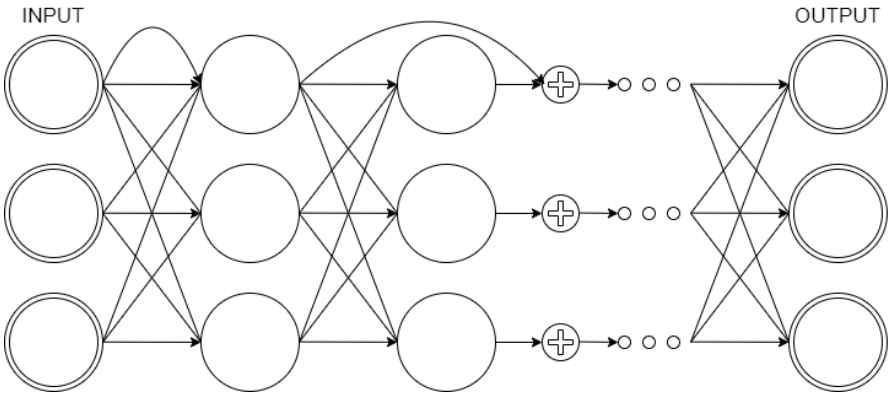


Figure 1.3: SimResNet structure with fixed 3 neurons: the *skip connection* is represented only for the first input but the skipped propagation holds for all the other ones too.

Chapter 2

Mean-field analysis

In this chapter, we will explore the intriguing relationship between a Simplified Neural Network and the mean-field formulation, obtained in the limit of infinitely many input data. By delving into this connection, we aim to uncover valuable insights into the inner workings of both systems and shed light on their potential synergies.

Moreover, we will delve into the concept of mean-field formulation aiming to reveal fascinating parallels and potential analogies between these seemingly distinct concepts.

To delve further into the specifics, our investigation will unveil a direct link to a specialized class of partial differential equations known as Hyperbolic Vlasov-type equations. For a more profound exploration of this latter subject, please refer to *Appendix A*.

Finally, armed with the insights gained from this connection, with a mean-field description of the SimResNet it will be possible to characterise the type of distribution of the equilibrium solution that we can obtain in the feature space, which will then be approximated by the SimResNet.

2.1 Mean-field formulation of SimResNets

Let assume to treat with data of d features, i.e. $\mathbf{x}_i \in \mathbb{R}^d$, and denote input values with $\mathbf{x}_i^{(0)} \in \mathbb{R}^d$. Considering N data and a network composed by L layers, we call data with the following diciture $\mathbf{x}_i^{(\ell)} \in \mathbb{R}^{d_\ell}$ with $i = 1, \dots, N$ and $\ell = 1, \dots, L$.
We denote with $\ell = 0$ and $\ell = L + 1$ respectively the input layer and the output one. Denoting with N_ℓ the number of neurons in each layers, which corresponds to the dimensions of considered data, we interpret with $\mathbf{x}_i^{(\ell)} \in \mathbb{R}^{N_\ell}$ the propagation of the i-th input signal through the network at the ℓ-th layer. Recalling (1.2) and given the input signal a ResNet defines precise microscopic dynamics for the propagation of each i-th data in this way:

$$\begin{cases} \mathbf{x}_i^{(\ell+1)} &= \mathbf{A}^{(\ell)}\mathbf{x}_i^{(\ell)} + \Delta t \sigma\left(\boldsymbol{\omega}^{(\ell)}\mathbf{x}_i^{(\ell)} + \mathbf{b}^{(\ell)}\right), \quad \ell = 0, \dots, L \\ \mathbf{x}_i^{(0)} &= \mathbf{x}_i^0 \end{cases} \tag{2.1}$$

where $\boldsymbol{\omega}^{(\ell)} \in \mathbb{R}^{N_{\ell+1} \times N_\ell}$ represents the weights and $\mathbf{b}(\ell) \in \mathbb{R}^{N_{\ell+1}}$ the biases.

These define the parameters of the network to be optimized.
$\sigma(\cdot) : \mathbb{R} \to \mathbb{R}$ denotes the activation function which is applied component-wise and supposed to be at least differentiable for the purpose of the model.
The matrix $\mathbf{A}^{(\ell)} \in \mathbb{R}^{N_{\ell+1} \times N_\ell}$ is a deterministic matrix which reduces to an identity matrix in the framework of the model, while Δt is a multiplicative factor regarding the time.
During the training procedure of a neural network the goal is to minimize the loss between the predicted output of $\mathbf{x}_i$ and its true target, this action can be very computationally expensive in the case $N, L \to \infty$ and specific optimization algorithms can be required to solve the minimization problem.
A consistent alternative is to formulate the traning process as a continuous optimization problem in such a way there is indipendence with respect to L and N.
Observe that in (2.1) the layers define a discrete structure within the ResNet: we interprete the layers as discrete times where the propagation of the input signal through the dynamics of the network is evaluated. To this end the parameter Δt is seen as the time step of the time discretization.

So now the goal is to compute the continuous limit of (2.1): to do this we need to add a fundamental condition to the neural network.
In order to study the minimization problem in the time continuous limit we need to deal with Simplified Residual Neural Network *(SimResNet)*, i.e. as said before, we are assuming that the number of neurons is identical in each layers. In particular this number is defined by the size of the input signal, it means that if we are considering data living in $\mathbb{R}^{\bar{d}}$, it holds $N_\ell = N = \bar{d} \quad \forall \ell = 1, \dots L+1$.

We recall that a similar assumption underlies the derivation of neural differential equations. By assuming that each layer in SimResNet contains the same number of neurons, we ensure that the network's capacity is consistent across layers. This means that the network's expressiveness and complexity remain balanced throughout the architecture.
This consistency is crucial in the mean field limit, as it allows us to take the limit of an infinitely large number of neurons, leading to the continuous representation of the network's behavior. When each layer contains the same number of neurons, we can confidently apply this limit to the entire network.
If we don't have this assumption, the derivation of neural differential equations becomes more challenging, and the direct application of the mean-field limit may not be feasible.
Without the assumption of a fixed neuron count, the network's capacity would vary across layers, resulting in imbalances in expressiveness and complexity. As a consequence, it would be difficult to take a straightforward mean-field limit that considers the behavior of the entire network as the number of neurons tends to infinity.

So, with this assumption the ResNet (2.1) can be seen as an explicit Euler discretization of an underlying differential equation.
From (2.1), considering $\Delta t \to 0^+$ and $L \to \infty$, we obtain the following *contin-*

uous structure of SimResNet:

$$\begin{cases} \frac{d}{dt}\mathbf{x}_i(t) & = \sigma\left(\boldsymbol{\omega}(t)\right)\mathbf{x}_i(t) + \mathbf{b}(t)\right), \quad t > 0 \\ \mathbf{x}_i^{(0)} & = \mathbf{x}_i^0 \end{cases} \tag{2.2}$$

for each $i = 1, \ldots, N$, where now $\boldsymbol{\omega}(t) \in \mathbb{R}^{\bar{d}\times\bar{d}}$ is the weights' matrix and $\mathbf{b}(t) \in \mathbb{R}^{\bar{d}}$ represents the bias.
Observe that this structure is the *neural differential equation* associated to the SimResNet.

Moreover, thanks the *Picard-Lindelőf Theorem* (see *Appendix B*) the existence and uniqueness of solution for the given system are ensured, as long as the activation function σ adheres to Lipschitz conditions and the functions $t \mapsto \boldsymbol{\omega}(t)$ and $t \mapsto \mathbf{b}(t)$ remain continuous.

Evidently, within this system, the computational and memory requirements of the neural network continue to escalate with the increasing dimensionality N of the dataset. Moreover, developing a network with infinite layers is impractical.
To mitigate this issue, an approach involves introducing a statistical interpretation of the neural network in the scenario where an infinite amount of data is available. To summarize, this kind of *simplified* structure allows us to reach the continuous limit and study the mean-field version of the problem, which is analyzed in the next section.

2.2 Analysis of the mean-field limit

In the previous section we obtained the *continuous SimResNet* (or *Neural Differential equation associated to the SimResNet*). The goal of this section is to compute the corresponding mean-field limit with respect the number of measurement N.

In this way we are able to describe the dynamic of (2.2) by an Hyperbolic Vlasov-type PDE of the form:

$$\begin{cases} \partial_t f(t,\mathbf{x}) + \nabla_{\mathbf{x}} \cdot \big(\sigma(\boldsymbol{\omega}(t)\mathbf{x} + \mathbf{b}(t))f(t,\mathbf{x})\big) = 0, \quad t > 0 \\ f(0,\mathbf{x}) = f_0(\mathbf{x}), \qquad \int_{\mathbb{R}^d} f_0(\mathbf{x})d\mathbf{x} = 1 \end{cases} \tag{2.3}$$

where $f(t,\mathbf{x}) : \mathbb{R}_0^+ \times \mathbb{R}^d \to \mathbb{R}_0^+$ is the probability distribution function, i.e. it gives the probability of finding a particle in position $\mathbf{x}$ at time t. Check *Appendix A* to get more information about this kind of equation.
Moreover we set the conditions assuming that initial distribution is well known and it is a probability density.
A crucial observation is that in the mean-field limit any information on the network output of a precise measurement is lost since we are dealing with a statistical information on the neural network propagation.

So we obtained the mean-field limit of the NDE, which represents an hyperbolic Vlasov-type equation with neural network-like terms.

Here's the explanation of the different components of the equation (2.3):

- $f(t, \mathbf{x})$: this represents the *probability distribution function* or *density function* defined on the domain of time t and space $\mathbf{x}$. It describes the density of particles or the probability of finding a particle at a given time and position.
- $\partial_t f(t, \mathbf{x})$: this term represents the partial derivative of the distribution function with respect to time. It describes how the distribution function changes over time.
- $\nabla_{\mathbf{x}}$: this is the spatial gradient operator with respect to the position vector $\mathbf{x}$. It represents the divergence of the spatial component of the equation.
- $\sigma(\boldsymbol{w}(t)\mathbf{x}+\mathbf{b}(t))$: this term involves a neural network-like structure. Here, $\sigma(\cdot)$ represents an activation function, $\boldsymbol{\omega}(t)$ denotes weight parameters that can vary with time and $\mathbf{b}(t)$ represents bias parameters which is time dependent too. The term $\boldsymbol{w}(t)\mathbf{x} + \mathbf{b}(t)$ represents the input to the activation function, where the weights and biases can modify the input before applying the activation function.
- $\nabla_{\mathbf{x}} \cdot \big(\sigma(\boldsymbol{w}(t)\mathbf{x} + \mathbf{b}(t))f(t, \mathbf{x})\big)$: this term computes the divergence of the vector field $\sigma(\boldsymbol{w}(t)\mathbf{x} + \mathbf{b}(t))f(t, \mathbf{x})$. It represents the *spatial variation* or *transport* of the neural network-like term, taking into account the distribution function $f(t, \mathbf{x})$.
- $= 0$: this equation states that the time derivative of the distribution function plus the divergence of the neural network-like term is equal to zero. This implies that the distribution function $f(t, \mathbf{x})$ satisfies this equation, capturing the evolution of the probability distribution over time and space.

In summary, the equation combines the concepts of the hyperbolic Vlasov equation with a neural network-like structure: it describes the evolution of the distribution function under the influence of the neural network-like term.

More in detail, in the next section will be proven the well-posedness of the Hyperbolic Vlasov-type equation (2.3), the existence and uniqueness of weak solution, the continuous dependence on the initial condition and on the paramaters, and finally the convergence of the continuous SimResNet (2.2) to (2.3) as $N \to \infty$.

Well-posedness of weak solution

In this section we will discuss about the properties of the mean-field limit of NDE (2.3), in order to do that we need to define some mathematical space and tools. Some of them can be found in *Appendix B*.

To achieve the convergence, we employ the framework of probability measures in the space $\mathcal{P}_p(\mathbb{R}^d)$. Specifically, we utilize the 1-Wasserstein distance to quantify it.
The 1-Wasserstein distance is a metric that measures the dissimilarity between two probability measures based on their transportation cost.
Through this approach, we will be able to establish the convergence of the solution of (2.2) to the mean field equation's solution (2.3) in the space of probability measures $\mathcal{P}_1(\mathbb{R}^d)$, providing a solid mathematical foundation for understanding the behavior of the system as we increase the number of neurons (N) in the SimResNet model.

Definition 1 (1-Wasserstein distance)**.** *Let μ and ν two probability measures on $\mathbb{R}^d$. Then the 1-Wasserstein distance is defined by*

$$W(\mu,\nu) := \inf_{\pi\in\mathcal{P}^*(\mu,\nu)} \int_{\mathbb{R}^d}\int_{\mathbb{R}^d} |\xi-\eta| d\pi(\xi,\eta) \tag{2.4}$$

where $\mathcal{P}^*$ is the space of probability measures on $\mathbb{R}^d\times\mathbb{R}^d$ such that the marginals are μ and ν, i.e.

$$\int_{\mathbb{R}^d} d\pi(\cdot,\eta) = d\mu(\cdot), \qquad \int_{\mathbb{R}^d} d\pi(\xi,\cdot) = d\nu(\cdot).$$

First of all we can notice that the microscopic system (2.2) describing a neural differential equation can be recast as an autonomous system using the auxiliary variables $\tau_i = \tau_i(t) \in \mathbb{R}$ for $i = 1,\dots,N$:

$$\begin{cases} \frac{d}{dt}x_i(t) = \sigma\left(\omega(\tau_i(t))x_i(t) + b(\tau_i(t))\right), & x_i(0) = x_i^0 \\ \frac{d}{dt}\tau_i(t) = 1, & \tau_i(0) = 0 \end{cases} \tag{2.5}$$

In the following, the right hand side of (2.5) will be compactly denoted using the function

$$\begin{aligned} G\colon \mathbb{R}^{d+1} &\to \mathbb{R}^{d+1} \\ (x,\tau) &\mapsto \big(\sigma(\omega(\tau)x + b(\tau)), 1\big)^T \end{aligned} \tag{2.6}$$

Let us give a formalization of a weak solution.

Definition 2 (Weak solution)**.** *Let $T > 0$ be fixed. Assume that $F_0 \in \mathcal{P}_1(\mathbb{R}^{d+1})$. We say that the time dependent measure $F_t \in C([0,T];\mathcal{P}_1(\mathbb{R}^{d+1}))$ is a weak solution to the mean-field equation*

$$\partial_t F_t + \nabla_x \cdot \big(\sigma(\omega(\tau)x + b(\tau))F_t\big) + \partial_\tau F_t = 0 \tag{2.7}$$

with initial condition F_0 if for all $\phi = \phi(x,\tau) \in C_0^\infty(\mathbb{R}^{d+1})$ and for all $t \in [0,T]$ the following equality holds:

$$\int_{\mathbb{R}^{d+1}} \phi(x,\tau)dF_t(x,\tau) = \int_{\mathbb{R}^{d+1}} \phi(x,\tau)dF_0(x,\tau) + \\ + \int_0^t \int_{\mathbb{R}^{d+1}} \nabla_{(x,\tau)}\phi(x,\tau) \cdot G(x,\tau)dF_s(x,\tau)ds$$

Taking it a step further, we can anticipate that *existence* and *uniqueness* of weak solution F_t of the mean-field equation (2.3) is obtained under the following assumptions *(See Proposition 1 below)*:

(A1) $\sigma \in C^{0,1}(\mathbb{R}^d), \quad \omega, b \in C^{0,1}(\mathbb{R})$; [1]

(A2) $|\sigma(x)| \leq C_0, \quad \forall x \in \mathbb{R}^d$

Remark 1. *We observe that assumption (A2) requires that the activation function σ is bounded. This property is verified for some choices of the activation function, e.g. if σ is the hyperbolic tangent function or the sigmoid function, but not in general. These assumptions likely ensure that the equations are well-behaved and the solutions are reasonable from a mathematical perspective.*

Now we can move on the next proposition which asserts that the solution of the particle dynamics system (2.5)-(2.6) converges to the solution F_t of the mean-field equation as the number of particles tends to infinity.

Proposition 1. *Let $F_0 \in \mathcal{P}_1(\mathbb{R}^{d+1})$ be given and let $T > 0$.*
Then, under the assumption (A1) and (A2), there exists a unique solution $F_t \in C([0,T];\mathcal{P}_1(\mathbb{R}^{d+1}))$ of the mean-field equation (2.7)*.*
In particular $F_t = \Phi_t \# F_0$ and F_t is continuously dependent on the initial data F_0 with respect to the 1-Wasserstain distance. Furthermore, the solution of the dynamical system (2.5)-(2.6) *converges to F_t in Wasserstain for $N \to \infty$.*

In summary, the proposition deals with the existence, uniqueness, and convergence of solutions in a mean-field context involving the Wasserstain distance under the described assumptions. The proof of this result can be found in *Appendix B.*

The previous proposition shows that the mean-field limit can be obtained provided that the controls $\omega, b \in C^{0,1}(\mathbb{R})$. If the control functions were less regular, the behavior of the large system might not be accurately described by the mean-field equation, and more complex interactions or phenomena could emerge.
This regularity ensures that the interactions among particles are well-behaved

[1] The notation $C^{0,1}$ is used to denote the space of functions that are continuous (C^0) and have bounded first derivatives (C^1).

and allows the mean-field equation to provide a valid description of the system's behavior as the number of particles increases.

As further result we establish the continuous dependence on the functions $\omega, b \in C^{0,1}(\mathbb{R})$.

Proposition 2. *Let $F_0 \in \mathcal{P}_1(\mathbb{R}^{d+1})$ be given and let $T > 0$. Then, under the assumptions (A1) and (A2), the unique solution $F_t \in C([0,T];\mathcal{P}_1(\mathbb{R}^{d+1}))$ of the mean-field equation* (2.7) *is continuosly dependent on (ω, b).*

This result, which might not be significant at first glance, actually has three important practical implications:

- *Robustness*: the continuity of the solution with respect to the control functions suggests that the behavior of the system is not drastically affected by small variations in the control functions. This is valuable in practical applications, where small uncertainties or inaccuracies in modeling the control functions are common.

- *Sensitivity Analysis*: the continuity property allows for a sensitivity analysis. During simulations it allowes to assess how changes in the control functions impact the system's behavior. This is essential for making informed decisions in designing and controlling systems.

- *Optimization*: in scenarios where control functions need to be optimized for desired system behavior, the continuity property can be used to design algorithms that iteratively improve control functions while ensuring gradual changes in the solution.

Proof. Denote by $\Phi^{(\omega,b)}$ the flow defined by equation (B.1) with given (ω, b). Then, for any (ω, b) fulfilling $(A1)$ and ω fulfilling $(A2)$ the assumptions of *Appendix B* (7) are satisfied and we obtain for $F_0 \in \mathcal{P}_1(\mathbb{R}^{d+1})$

$$W_1(\Phi^{(\omega,b)}\#F_0), \Phi^{(\bar{\omega},\bar{b})}\#F_0) \leq \frac{exp(Lt-1)}{L}||(\omega,b)-(\bar{\omega},\bar{b})||_{C^0(0,T)}, \qquad (2.8)$$

where $L = max\{L_{G(\omega,b)}, L_{G(\bar{\omega},\bar{b})}\}$ is the maximum of the Lipschitz constants of G defined by equation (2.6). □

The following proposition will establish a connection between weak solutions of the mean-field equation given by equation (2.7) and weak solutions of the other form of the mean-field equation represented by equation (2.3). More in detail, the proposition will show that if certain conditions are met, a solution of one form of the equation corresponds to a solution of the other form.

Proposition 3. *If a weak solution $F_t \in C([0,T];\mathcal{P}_1(\mathbb{R}^{d+1}))$ of* (2.7) *fulfills*

$$dF_t(x,\tau) = df_t(x)\delta(\tau - t) \qquad (2.9)$$

with $f_t \in C([0,T];\mathcal{P}_1(\mathbb{R}^d))$, and if $dF_0(x,\tau) = df_0(x)\delta(\tau)$ with $f_0 \in \mathcal{P}_1(\mathbb{R}^d)$, then, under the assumptions $(A1)$ and $(A2)$, f_t is a weak solution of the mean-field equation (2.3) *with initial condition f_0.*

Observations:

- The weak solution F_t is represented in a particular form, where it has a density function $f_t(x)$ that evolves over time t.
- The structure $dF_t(x,\tau) = df_t(x)\delta(\tau - t)$ implies that at every moment t, the density $f_t(x)$ describes the probability distribution of particles at that specific time t.
- Similarly, the initial condition $dF_0(x,\tau) = df_0(x)\delta(\tau)$ implies that the initial density $f_0(x)$ represents the initial probability distribution of particles at time $t = 0$.

Proof. Using the assumption on F_t and F_0 in definition 2 we find for all $\phi = \phi(x) \in C_0^\infty(\mathbb{R}^d)$:

$$\int_{\mathbb{R}^d} \phi(x) df_t(x) = \int_{\mathbb{R}^d} \phi(x) df_0(x) + \int_0^t \int_{\mathbb{R}^d} \nabla_x \phi(x) \cdot \sigma(\omega(t)x + b(t)) df_s(x) ds$$

which is exactly the weak form of the mean-field equation (2.3) with initial condition f_0. □

As remark, this proposition shows a relationship between two forms of the mean-field equation. It suggests that if you have a solution that captures the "snapshot" of particle distribution at each time instant, this solution corresponds to a solution of the traditional form of the mean-field equation, where the distribution evolves smoothly over time. This provides insight into the connection between these two formulations and their equivalence under certain conditions.

Mean-field controllability system

In this section, the controllability property of the mean-field PDEs is presented. Control challenges for PDEs manifest in various scenarios: a central concern is controllability, wherein we investigate the feasibility of directing the PDE's solution to a desired target state through control actions.
Drawing parallels to neural networks' continuous formulation, we can perceive the training phase as a controllability issue.
This *controllability* corresponds to the following interpretation:

Definition 3. *Let $f_0, g \in \mathcal{P}_1(\mathbb{R}^d)$ be given. Let $T > 0$ fixed. We say that the mean-field equation (2.3) is controllable if there exist $\omega \in C^{0,1}([0,T];\mathbb{R}^{d\times d})$ and $b \in C^{0,1}([0,T];\mathbb{R}^d)$ such that $(\Phi_T \# f_0) = g$ where $\Phi_t : \mathbb{R}^d \to \mathbb{R}^d$ is the Lipschitz continuous characteristic flow of (2.3)*

Now, the following result characterizes the steady state solution of (2.3). More in particular, it illustrates that if initial and terminal states are a sum of weighted Dirac measures the system is trivially controllable with parameters $(\omega^\infty, b^\infty)$ expressed by the following proposition.

Proposition 4. *Let $f : \mathbb{R}_o^+ \times \mathbb{R}^d \to \mathbb{R}_o^+$ be the compactly supported weak solution of the mean-field equation* (2.3)*. Assume that the activation function $\sigma : \mathbb{R} \to \mathbb{R}$ has disjunct zeros z_k, $k = 1, \dots, K$ for some $K > 0$. Let $b^\infty = lim_{t\to\infty} b(t)$ and $\omega^\infty = lim_{t\to\infty}\omega(t)$ exists and finite.*
Then

$$f^\infty(x) = \sum_{i=1}^{n^d} \rho_i \delta(x - y_i) \tag{2.10}$$

is a steady state solution of (2.3) *in sense of distributions provided that $\boldsymbol{y}_i$ for $i = 1, \dots, n^d$ are disjunct solutions of $\omega^\infty y_i + b^\infty = z_k$ for some k and $\rho_i \in [0, 1]$ such that $\sum_{i=1}^{n^d} \rho_i = 1$.*

Proof. For any test function $\phi(\mathbf{x}) \in C_0^\infty(\mathbb{R}^d)$ the steady state $f^\infty(x)$ of the weak form of (2.3) solves

$$\int_{\mathbb{R}^d} \nabla_x \phi(x) \sigma(\omega^\infty x + b^\infty) g^\infty(x) = 0 \tag{2.11}$$

If $f^\infty(x)$ is defined as (2.10), then equation (2.11) is satisfied only if y_i is the solution to the system $\omega^\infty y + b^\infty = z$, with z any disposition of d elements with repetition of the n zeros of the activation function σ. □

The proposition concludes that under these conditions, the function $f^\infty(x)$ defined by equation (2.10) is a steady state solution of the mean-field equation. This means that $f^\infty(x)$ represents a distribution that remains unchanged over time, even though the system's individual particles are affected by the time-varying controls $\omega(t), b(t)$.

Moreover, the previous proposition has a direct impact on the selection of the activation function. Unless $\sigma(\cdot)$ possesses more than one zero, the steady state of the mean-field equation is represented by a single Dirac delta function. This situation arises when opting for activation functions such as the identity or the hyperbolic tangent.
On the other hand, by choosing the growing cosine unit activation function, for example, an interesting alternative emerges. In this case, the steady state of the mean-field equation consists of a linear combination of Dirac delta functions. This choice allows for a richer representation, capturing multiple peaks in the steady state distribution.
In summary, the proposition highlights how the activation function significantly influences the nature of the steady state in the mean-field equation. The presence or absence of multiple zeros in the activation function dictates whether the system settles into a single Dirac delta or a more complex linear combination of Dirac delta functions at equilibrium.[2]

More in detail, we can compute a good candidate for the weak solution of (2.3) which satisfies all the results proved above.

[2]However, while a single zero in the activation function typically suggests a single simpler steady state, real-world systems can exhibit unexpected and nuanced behavior, and careful analysis is essential to understand such phenomena.

The derivation of (2.3) from (2.2) follows some steps approaching with Liouville's theorem: first of all we can prove that an empirical distribution density is the weak solution of the hyperbolic Vlasov-type equation (2.3) and so it is possible to prove its convergence in a specific probability space.
Therefore, defining this *empirical distribution density*

$$f^N(x,t) := \frac{1}{N}\sum_{i=1}^{N}\delta(x, x_i(t)) \tag{2.12}$$

where

$$\delta(x, x_i(t)) = \begin{cases} 1 & if \quad x_i(t) = x \\ 0 & otherwise \end{cases} \qquad \forall i = 1, \dots, N$$

we can give a formal derivation of the evolution of $f(x,t)$ corresponding to the mean-field limit of (2.2) in the number of measurements N.
So, for each choice of *test function* $\phi \in C_0^1(\mathbb{R})$ with compact support and considering the structure of $f^N(x,t)$ we can write:

$$\int_{\mathbb{R}^d}\phi(x(t))f^N(x,t)dx = \frac{1}{N}\sum_{i=1}^{N}\phi(x_i(t))$$

Now, computing the time derivative of both sides and expanding the right one we obtain:

$$\begin{aligned}
\frac{d}{dt}\int_{\mathbb{R}^d}\phi(x(t))f^N(x,t)dx &= \frac{d}{dt}\left(\frac{1}{N}\sum_{i=1}^{N}\phi(x_i(t))\right) \\
&= \frac{1}{N}\sum_{i=1}^{N}\nabla_x\phi(x_i(t))\cdot\dot{x}_i(t) \\
&= \frac{1}{N}\sum_{i=1}^{N}\nabla_x\phi(x_i(t))\cdot\sigma\big(\omega(t)x_i(t)+b(t)\big) \\
&= \int_{\mathbb{R}^d}\nabla_x\phi(x(t))\cdot\sigma\big(\omega(t)x(t)+b(t)\big)\cdot f^N(x,t)dx
\end{aligned}$$

Now, considering $\nabla_x\phi(x(t))$ as primitive and $\sigma\big(\omega(t)x(t)+b(t)\big)\cdot f^N(x,t)$ as derivative we can integrate by parts:

$$\begin{aligned}
\int_{\mathbb{R}^d}\nabla_x\phi(x(t))\cdot\sigma\big(\omega(t)x(t)+b(t)\big)\cdot f^N(x,t)dx = \\
= \big[\phi(x(t))\cdot\sigma(\omega(t)x(t)+b(t))\cdot f^N(x,t)\big] + \\
-\int_{\mathbb{R}^d}\phi(x(t))\nabla_x\cdot\big(\sigma(w(t)x(t)+b(t))\cdot f^N(x,t)\big)dx = \\
= 0 - \int_{\mathbb{R}^d}\phi(x(t))\nabla_x\cdot\big(\sigma(w(t)x(t)+b(t))\cdot f^N(x,t)\big)dx
\end{aligned}$$

Combining left hand-side and right one, we obtain:

$$\frac{d}{dt}\int_{\mathbb{R}^d}\phi(x(t))f^N(x,t)dx = -\int_{\mathbb{R}^d}\phi(x(t))\nabla_x\cdot(\sigma(w(t)x(t)+b(t))\cdot f^N(x,t))dx$$

$$\implies \int_{\mathbb{R}^d}\phi(x(t))\big[\partial_t f^N(x,t)+\nabla_x\cdot(\sigma(w(t)x(t)+b(t))\cdot f^N(x,t))\big]dx = 0$$

$$\implies \qquad \partial_t f^N(x,t)+\nabla_x\cdot(\sigma(w(t)x(t)+b(t))\cdot f^N(x,t)) = 0$$

Observe that the last equation is exactly (2.3), so, having established $f^N(x,t)$ as a weak solution of it, we can conclude all the results above.

As a remark, it is important to remember the power of the link between mean field equation and neural network in dealing with stationary solutions. In particular, the mean-field analysis serves to understand the structure of asymptotic solutions[3], while neural networks provide us with an approximation of the solution of a PDE that has a certain stationary distribution.

Proposition (4) in this context establishes the existence of steady solutions for the mean-field equation, which are distinguished by their constant weights and biases.

Regarding this, a common mistake is to think that this procedure will obviate the need for an optimisation process. Contrary to this belief, the straightforward application of steady state solutions is not universally sufficient. This happens because the steady state solutions are defined by unchanging weights and biases. As a result, they lack the built-in ability to adjust and handle the intricate variations found in real-world data patterns.

However, while these steady states provide fixed solutions to the equation, their inherent rigidity prevents them from effectively accommodating the rich diversity and dynamic patterns found in real data distributions. In other words, the constant nature of weights and biases does not inherently encompass the flexibility required to capture the intricate relationships within complex datasets.

Hence, even when steady state solutions are at hand, a dedicated optimization algorithm remains imperative. Such an algorithm should effectively adapt the constant weights and biases of the network to the target data distribution, ensuring that the network can flexibly adjust its parameters to accurately classify the data. This algorithm will be presented two sections later.
In this manner, the optimization procedure augments the static solutions of the mean-field equation with the capacity to capture the adaptable nature of real-world data distributions.

[3]Let's give a fast example. From a mathematical perspective, particularly when employing the mean-field formulation of neural networks, the process of classification can be conceptualized as the task of driving the distribution of the input data towards a distribution composed of Dirac delta functions.

2.3 Moment properties of the one-dimensional mean-field equation

This section investigates under which conditions steady states might be obtained as solution to the time-dependent mean-field equation. This analysis is based on the moments of $f(t,x)$, focusing on the one-dimensional case. With such restriction (2.3) reduces to

$$\begin{cases} \partial_t f(t,x) + \partial_x\big(\sigma(\omega(t)x + b(t))f(t,x)\big) = 0, & t > 0 \\ f(0,x) = f_0(x), \qquad \int_{\mathbb{R}} f_0(x)dx = 1 \end{cases} \tag{2.13}$$

The moment properties of the mean field equation are crucial for this analysis. Let us break down why these moment properties are important in this context:

- *Characterizing the distribution*: the moment properties provide a way to characterize the distribution of the stochastic variable x in the equation. Moments, such as the mean (first moment) and variance (second moment), describe important statistical properties of the distribution. By understanding these moments, you gain insights into the behavior and spread of the population.

- *Steady States and Equilibrium*: steady states in a dynamical system are states where the system does not change with time. In the context of your equation, a steady state would be a solution $f(t,x)$ that remains constant over time. To investigate when such steady states are possible, you need to understand how the moments of the distribution evolve with time. Steady states are often associated with specific moment values remaining constant.

- *Stability Analysis*: determining the stability of steady states is a key part of understanding the behavior of dynamical systems. By examining moment properties, you can assess whether perturbations from a steady state will cause the system to return to that state (stable) or move away from it (unstable).

In summary, the moment properties of the mean field equation provide critical information about the central tendency, spread, and shape of the distribution, which are crucial for analyzing the behavior and potential steady states of the system described by the mean-field equation.

Definition 4 (k-th moment)**.** *Given $k \geq 0$, the k-th moment of the probability distribution $f(t,x)$ is defined as*

$$m_k(t) := \int_{\mathbb{R}} x^k f(t,x)dx$$

Definition 5 (Variance)**.** *The variance of the probability distribution $f(t,x)$ is defined as*

$$\mathbb{V}(t) = m_2(t) - (m_1(t))^2$$

We describe the behavior of the solution $f(t,x)$, $(t,x) \in \mathbb{R} \times \mathbb{R}$ to the mean-field equation (2.13) using the following criteria:

Definition 6. *We say that the solution $f(t,x), (t,x) \in \mathbb{R}^+ \times \mathbb{R}$, to the mean-field equation* (2.13) *is characterized by*

(i) ***local energy bound*** *if*

$$m_2(0) > m_2(t),$$

holds at a fixed time t;

(ii) ***energy decay*** *if*

$$m_2(t_1) > m_2(t_2),$$

holds for any $t_1 < t_2$, i.e. if the energy is decreasing with respect to time;

(iii) ***local aggregation*** *if*

$$\mathbb{V}(0) > \mathbb{V}(t),$$

holds at a fixed time t;

(iv) ***aggregation*** *if*

$$\mathbb{V}(t_1) > \mathbb{V}(t_2),$$

holds for any $t_1 < t_2$, i.e. if the variance is decreasing with respect to time;

(v) ***concentration*** *or* ***clustering*** *if*

$$\lim_{t\to\infty} \mathbb{V}(t) = 0,$$

i.e. if the variance vanishes in the long time behaviour.

We note that when the first moment is conserved over time, local energy bound (i) is equivalent to local aggregation (iii), and energy decay (ii) is equivalent to aggregation (iv).
In the context of a residual neural network, achieving concentration (v) or aggregation (iv) phenomena implies that we can obtain an output distribution with decreasing variance concerning the input distribution. This is particularly important for classification tasks.

We will analyze these phenomena further, considering the *identity function* as the activation function, and derive conditions on the parameters to observe these behaviors.
A simple computation reveals that the $0-th$ moment is conserved, i.e.
$m_0(t) = \int_{\mathbb{R}} f(t,x)dx = 1$ holds for all times $t \geq 0$, as we expect since (2.3) is a conservation law.
Instead, for $k \geq 1$ the behaviour in time of the corresponding moment is prescribed by the following linear ordinanary differential equation:

$$\frac{d}{dt}m_k(t) = k\big(\omega(t)m_k(t) + b(t)m_{k-1}(t)\big), \qquad m_k(0) = m_k^0 \tag{2.14}$$

Notice that the $k-th$ moment only depends on the $(k-1)-th$ moment. It is then possible to solve the moment equations iteratively with the separation

of variables approach, obtaining

$$m_k(t) = e^{\Phi_k(t)}\left(m_k(0) + k\int_0^t e^{-\Phi_k(s)}b(s)m_{k-1}(s)ds\right) \tag{2.15}$$

where

$$\Phi_k(t) := k\int_0^t \omega(s)ds$$

We will now investigate the conditions on the parameters necessary to observe the phenomena described in Definition 6.

Proposition 5. *Assume that the bias is identical to zero, namely* $b(t) \equiv 0, \forall t \geq 0$.
Under this assumption, we can establish the following:

(a) ***local energy bound*** *if* $\Phi_1(t) < 0$ *at a fixed time* t*;*

(b) ***energy decay*** *if and only if* $\omega(t) < 0$ *for all time* $t > 0$*;*

(c) ***clustering*** *if and only if* $\lim_{t\to\infty} \Phi_1(t) = -\infty$*. In particular the stedy state is distributed as a Dirac delta centered at* $x = 0$*.*

Proof. If $b(t) \equiv 0$ then (2.15) simplifies to

$$m_k(t) = m_k(0)e^{\Phi_k(t)} \tag{2.16}$$

and thus the first and the second moment are identical except the given initial conditions. Then we can easily apply the definitions of energy bound, energy decay and clusterining to prove the statement. □

The previous result suggests that when $b(t) \equiv 0$, we can address clustering problems at the origin, irrespective of the initial first moment. The following corollary establishes equivalence among the phenomena defined above under the hypothesis of Proposition 5.

Corollary 1. *Assume that the bias is identical zero, namely* $b(t) \equiv 0, \forall t \geq 0$. *Then*

(a) if ***local energy bound*** *exists at a time* t *we have* ***local aggregation****;*

(b) if ***energy decay*** *holds we have* ***aggregation****.*

Proof. We start proving the first statement. First we observe that (2.16) is still true, since by hypothesis we assume that the bias is zero. Due to the definition of local aggregation we need to verify that $\mathbb{V}(0) > \mathbb{V}(t)$ for a fixed time t. Using the definition of the variance, local aggregation is implied by

$$m_2(0)(1 - e^{\Phi_2(t)}) > m_1(0)^2(1 - e^{2\Phi_1(t)})$$

For the second part, we observe that aggregation phenomena is verified if $\mathbb{V}(t_1) > \mathbb{V}(t_2)$ for any $t_1 < t_2$. This is equivalent to

$$m_2(t_1) - m_1(t_1)^2 > m_2(t_2) - m_1(t_2)^2 \tag{2.17}$$
$$\iff m_2(t_1) - m_2(t_2) > (m_1(t_1) - m_1(t_2))(m_1(t_1) + m_1(t_2)) \tag{2.18}$$
$$\iff m_2(t_1) > m_1(t_1)^2 \tag{2.19}$$

□

Conservation of the first moment is guaranteed by choosing $b(t) := -\omega(t)m_1(t)$. See the following results.

Proposition 6. *Let the bias be* $b(t) := -\omega(t)m_1(t), \forall t \geq 0$. *Then the first moment* m_1 *is conserved in time and we obtain*

(a) ***local energy bound*** *if* $\Phi_2(t) < 0$ *at a fixed time* t*;*

(b) ***clustering*** *phenomenon if* $\omega(t) < 0$ *holds for all* $t \geq 0$. *In particular the steady state is distributed as a Dirac delta centered at* $x = m_1(0)$.

Proof. The solution formula for the second moment is

$$m_2(t) = e^{\Phi_2(t)}(m_2(0) - m_1(0))^2 + m_1(0)^2 = e^{\Phi_2(t)}\mathbb{V}(0) + m_1(0)^2$$

Then we have local energy bound if

$$\mathbb{V}(0)\big(e^{\Phi_2(t)} - 1\big) < 0,$$

which is satisfied assuming that $\Phi_2(t) < 0$ at a fixed time t. For the second statement we observe that concentration to a delta is also implied by $\Phi_2(t) < 0$ for all times t, so that $m_2(t) \to m_1(t)^2$ for $t \to \infty$. This occurs if $\omega(t) < 0$ for all times. □

Now, let us discuss the impact of the variance on aggregation and concentration phenomena. This is especially important when focusing on a finite time horizon to determine whether $\mathbb{V}(T) \leq V$ for some tolerance $V > 0$ and time $T > 0$. In practical applications, this tolerance level would be determined by the variance of the target distribution and helps prevent overfitting phenomena.

Corollary 2. *If the bias is identical to zero, namely* $b(t) \equiv 0, \forall t \geq 0$, *then the energy at time* $t > 0$ *is below tolerance* $V > 0$ *if*

$$\Phi_2(t) < ln\bigg(\frac{V}{m_2(0)}\bigg),$$

is satisfied. Instead, the variance is below the level $V > 0$ *if*

$$\Phi_2(t) < ln\bigg(\frac{V}{\mathbb{V}(0)}\bigg)$$

holds.
Similarly, if the bias fulfills $b(t) := -\omega(t)m_1(t)$, *then the energy at time* $t > 0$ *is below the level* $V > 0$ *if*

$$\Phi_2(t) < ln\bigg(\frac{V - m_1(0)^2}{\mathbb{V}(0)}\bigg)$$

is satisfied provided that $V > m_1(0)^2$ *holds. Instead, the variance at time* $t > 0$ *is below the level* $V > 0$ *if*

$$\Phi_2(t) < ln\bigg(\frac{V}{\mathbb{V}(0)}\bigg),$$

is satisfied provided that $V > 0$ *holds.*

Remark 2. *In general it is not possible to obtain a closed moment model in the case of the sigmoid $\sigma_S(x)$ or hyperbolic tangent $\sigma_T(x)$ activation function. Nevertheless one might approximate both activation functions by the linear part of their series expansion:*

$$\sigma_S(x) \approx \frac{1}{2} + \frac{x}{4}, \quad \sigma_T(x) \approx x$$

As explained before, the following section will be related to the algorithm used to adapt the constant weights and biases of the network in order to make the model flexibly fit diffrent data.

2.4 Forward re-training algorithm

In this section we aim to build a Forward re-training algorithm in order to find out the constant weights and biases related to the steady solutions.
The particularity of this approach is the fact that the training procedure is based on the motion of each single particle. In other words, we consider the kinetic version of the system.
More in detail, we introduce a training algorithm for the weights and biases of the SimResNet using a formulation as *optimal control problem*.
In this way, computing the sensitivity quantity of the loss function and approaching with Stochastic Gradient Descent, we will get the updating rules of parameters in the training process.

We will consider different version of *OCP*, which will be tested later.

Classification problem - one dimensional case

Given target values $y_j \sim \xi(x), \quad j = 1\ldots., N$ and N_T the last time step (which is represented by the last layer of the network), we aim to minimize the following loss function:

$$L(N_T) = \frac{1}{N} \sum_{j=1}^{N} \frac{1}{2} \|x_j^{(N_T)} - y_j\|_2^2 \tag{2.20}$$

We obtain the kinetic version of the *optimal control problem*:

$$\begin{aligned} &\arg\min_{\omega,b} L(N_T; \omega, b) \\ &\quad s.t. \quad x_j^{(n+1)} = x_j^{(n)} + h \cdot \sigma(\omega^{(n)} x_j^{(n)} + b^{(n)}) \\ &\qquad x_j^{(o)} \in \mathbb{R}^d \end{aligned}$$

where $\omega \in \mathbb{R}^{d \times d}, b \in \mathbb{R}^d$ and h is the time step.
To study the optimality condition we have to consider the Lagrangian including

lagrangian multipliers $\lambda_j^{(n)}$, obtaining the following:

$$\begin{aligned}\mathcal{L}(x,\lambda,\omega,b) =& \frac{1}{N}\sum_{j=1}^{N}\frac{1}{2}\|x_j^{(N_T)}-y_j\|_2^2+ \\ &-\frac{h}{N}\sum_{n=0}^{N_T-1}\sum_{j=1}^{N}\lambda_j^{(n+1)}\cdot\big(x_j^{(n+1)}-x_j^{(n)}-h\cdot\sigma(w^{(n)}x_j^{(n)}+b^{(n)})\big)\end{aligned} \tag{2.21}$$

Observe that, since the first term includes only final information, we can obtain optimality conditions by deriving only in the case $n \neq N_T$. Let's first extend the Lagrangian at time $n = \{n-1, n, n+1\}$ in order to compute derivatives:

$$\begin{aligned}\mathcal{L}(x,\lambda,\omega,b) = & \frac{1}{N}\sum_{j=1}^{N}\|x_j^{(N_T)}-y_j\|_2^2+ \\ &-\frac{h}{N}\big[\cdots+\lambda_j^{(n)}\cdot\big(x_j^{(n)}-x_j^{(n-1)}-h\cdot\sigma(w^{(n-1)}x_j^{(n-1)}+b^{(n-1)})\big)+ \\ &+\lambda_j^{(n+1)}\cdot\big(x_j^{(n+1)}-x_j^{(n)}-h\cdot\sigma(w^{(n)}x_j^{(n)}+b^{(n)})\big)+ \\ &+\lambda_j^{(n+2)}\cdot\big(x_j^{(n+2)}-x_j^{(n+1)}-h\cdot\sigma(w^{(n+1)}x_j^{(n+1)}+b^{(n+1)})\big)+\dots\big]\end{aligned}$$

So we can compute:

$$\begin{aligned}\partial_{x_j^{(n)}}\mathcal{L} &= \overbrace{\partial_{x_j^{(n)}}L(N_T)}^{=0}-\frac{h}{N}\big(\lambda_j^{(n)}-\lambda_j^{(n+1)}-\lambda_j^{(n+1)}\cdot h\cdot\sigma'(\omega^{(n)}x_j^{(n)}+b^{(n)})\omega^{(n)}\big) \\ &= \frac{h}{N}\big(\lambda_j^{(n+1)}+\lambda_j^{(n+1)}\cdot h\cdot\sigma'(\omega^{(n)}x_j^{(n)}+b^{(n)})\omega^{(n)}-\lambda_j^{(n+1)}\big)=0\end{aligned}$$

This implies:

$$\lambda_j^{(n)} = \lambda_j^{(n+1)}+\lambda_j^{(n+1)}\cdot h\cdot\sigma'(\omega^{(n)}x_j^{(n)}+b^{(n)})\omega^{(n)} \tag{2.22}$$

which requires a final point[4] set as $\lambda_j^{(N_T)} = |x_j^{(N_T)}-y_j|$.
$\sigma'(z)$ is the derivative of the activation function which is assumed to be differentiabile, e.g. sigmoid or hyperbolic tangent function.
Furthermore, we compute the sensitivity quantities:

$$\partial_{\omega^{(n)}}\mathcal{L} = \frac{h}{N}\sum_{j=1}^{N}\lambda_j^{(n+1)}\cdot h\cdot\sigma'(\omega^{(n)}x_j^{(n)}+b^{(n)})\cdot x_j^{(n)}$$

$$\partial_{b^{(n)}}\mathcal{L} = \frac{h}{N}\sum_{j=1}^{N}\lambda_j^{(n+1)}\cdot h\cdot\sigma'(\omega^{(n)}x_j^{(n)}+b^{(n)})$$

[4]This quantity has been derived computing $\partial_{x_j^{(N_T)}}L(N_T;\omega,b)$

obtaining the following parameters' updates:

$$\omega^{(n+1)} = \omega^{(n)} - \gamma \frac{h}{N} \sum_{j=1}^{N} \lambda_j^{(n+1)} \cdot h \cdot \sigma'(\omega^{(n)} x_j^{(n)} + b^{(n)}) \cdot x_j^{(n)} \tag{2.23}$$

$$b^{(n+1)} = b^{(n)} - \gamma \frac{h}{N} \sum_{j=1}^{N} \lambda_j^{(n+1)} \cdot h \cdot \sigma'(\omega^{(n)} x_j^{(n)} + b^{(n)}) \tag{2.24}$$

where γ is the chosen stepsize for the gradient descent method.

Finally, in this framework, we can obtain the following algorithm to reach optimal parameters values:

Algorithm 1: Forward update of the parameters of the SimResNet

Initialize $\omega^{(0)}$ and $b^{(0)}$ randomly
for $iter \leftarrow 1$ **to** n_iter **do**
Propagate training data with the current parameters
Compute the loss
Compute the new updates following

$$\lambda_j^{(n)} = \lambda_j^{(n+1)} + \lambda_j^{(n+1)} \cdot h \cdot \sigma'(\omega^{(n)} x_j^{(n)} + b^{(n)}) \omega^{(n)}$$

$$\omega^{(n+1)} = \omega^{(n)} - \gamma \frac{h}{N} \sum_{j=1}^{N} \lambda_j^{(n+1)} \cdot h \cdot \sigma'(\omega^{(n)} x_j^{(n)} + b^{(n)}) \cdot x_j^{(n)}$$

$$b^{(n+1)} = b^{(n)} - \gamma \frac{h}{N} \sum_{j=1}^{N} \lambda_j^{(n+1)} \cdot h \cdot \sigma'(\omega^{(n)} x_j^{(n)} + b^{(n)})$$

end

Regression problem - two dimensional case

In this case we aim to minimize the following loss function:

$$L(N_T;\omega,b) = \frac{1}{n*}\sum_{i=1}^{n^*}\frac{1}{2}||y_i - m^{(N_T)}x_i - q^{(N_T)}||^2 \tag{2.25}$$

Since now we are considering the propagation of two paramters, i.e. the slope m and the intercept q of the model, the relative *optimal control problem* is the following:

$$\begin{aligned}
&\arg\min_{\omega,b} L(N_T;\omega,b)\\
&\quad s.t. \quad m^{(n+1)} = m^{(n)} + h\cdot\sigma(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)})\\
&\qquad\quad q^{(n+1)} = q^{(n)} + h\cdot\sigma(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)})
\end{aligned}$$

where now $\omega = (\omega_{11},\omega_{12},\omega_{21},\omega_{22}) \in \mathbb{R}^4$ and $b = (b_1,b_2) \in \mathbb{R}^2$.

As before, we can compute the Lagrangian in order to find out the optimality conditions, in this case we need to consider two different *lagrangian multipliers* $\lambda^{(n)}$ and $\mu^{(n)}$:

$$\begin{aligned}
\mathcal{L}(\lambda,\mu,\omega,b) = &\frac{1}{n*}\sum_{i=1}^{n^*}\frac{1}{2}||y_i - m^{(N_T)}x_i - q^{(N_T)}||^2 +\\
&-\frac{1}{N_T}\sum_{n=0}^{N_T-1}\lambda^{(n+1)}\cdot(m^{(n+1)} - m^{(n)} - \sigma(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)})) +\\
&-\frac{1}{N_T}\sum_{n=0}^{N_T-1}\mu^{(n+1)}\cdot(q^{(n+1)} - q^{(n)} - \sigma(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)}))
\end{aligned}$$

So we can compute the sensitivities quantity:

$$\begin{aligned}
\partial_{m^{(n)}}\mathcal{L} = &-\lambda^{(n)} + \lambda^{(n+1)} + \lambda^{(n+1)}\sigma'(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)})\cdot\omega_{11}^{(n)} +\\
&+\mu^{(n+1)}\sigma'(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)})\cdot\omega_{12}^{(n)} = 0
\end{aligned}$$

$$\begin{aligned}
\partial_{q^{(n)}}\mathcal{L} = &-\mu^{(n)} + \mu^{(n+1)} + \mu^{(n+1)}\sigma'(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)})\cdot\omega_{22}^{(n)} +\\
&+\lambda^{(n+1)}\sigma'(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)})\cdot\omega_{21}^{(n)} = 0
\end{aligned}$$

This implies:

$$\begin{aligned}
\lambda^{(n)} = \lambda^{(n+1)} &+ \lambda^{(n+1)}\sigma'(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)})\cdot\omega_{11}^{(n)} +\\
&+\mu^{(n+1)}\sigma'(\omega_{12}^{(n)}m^{(n)}\omega_{22}^{(n)}q^{(n)} + b_2^{(n)})\cdot\omega_{12}^{(n)}
\end{aligned} \tag{2.26}$$

$$\mu^{(n)} = \mu^{(n+1)} + \mu^{(n+1)}\sigma'(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)}) \cdot \omega_{22}^{(n)} + \\ + \lambda^{(n+1)}\sigma'(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)}) \cdot \omega_{21}^{(n)} \quad (2.27)$$

which require the respective final points corresponding to the residuals.
In order to update each parameter's update we need to compute derivatives:

$$\partial_{\omega_{11}^{(n)}}\mathcal{L} = \frac{1}{N_T}\sum_{n=0}^{N_T-1} \lambda^{(n+1)}\sigma'(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)}) \cdot m^{(n)}$$

$$\partial_{\omega_{12}^{(n)}}\mathcal{L} = \frac{1}{N_T}\sum_{n=0}^{N_T-1} \mu^{(n+1)}\sigma'(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)}) \cdot m^{(n)}$$

$$\partial_{\omega_{21}^{(n)}}\mathcal{L} = \frac{1}{N_T}\sum_{n=0}^{N_T-1} \lambda^{(n+1)}\sigma'(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)}) \cdot q^{(n)}$$

$$\partial_{\omega_{22}^{(n)}}\mathcal{L} = \frac{1}{N_T}\sum_{n=0}^{N_T-1} \mu^{(n+1)}\sigma'(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)}) \cdot q^{(n)}$$

$$\partial_{b_1^{(n)}}\mathcal{L} = \frac{1}{N_T}\sum_{n=0}^{N_T-1} \lambda^{(n+1)}\sigma'(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)})$$

$$\partial_{b_2^{(n)}}\mathcal{L} = \frac{1}{N_T}\sum_{n=0}^{N_T-1} \mu^{(n+1)}\sigma'(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)})$$

Obtaining in this way the parameters' updates:

$$\omega_{11}^{(n+1)} = \omega_{11}^{(n)} - \gamma\frac{1}{N_T}\sum_{n=0}^{N_T-1} \lambda^{(n+1)}\sigma'(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)}) \cdot m^{(n)} \quad (2.28)$$

$$\omega_{12}^{(n+1)} = \omega_{12}^{(n)} - \gamma\frac{1}{N_T}\sum_{n=0}^{N_T-1} \mu^{(n+1)}\sigma'(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)}) \cdot m^{(n)} \quad (2.29)$$

$$\omega_{21}^{(n+1)} = \omega_{21}^{(n)} - \gamma\frac{1}{N_T}\sum_{n=0}^{N_T-1} \lambda^{(n+1)}\sigma'(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)}) \cdot q^{(n)} \quad (2.30)$$

$$\omega_{22}^{(n+1)} = \omega_{22}^{(n)} - \gamma\frac{1}{N_T}\sum_{n=0}^{N_T-1} \mu^{(n+1)}\sigma'(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)}) \cdot q^{(n)} \quad (2.31)$$

$$b_1^{(n+1)} = b_1^{(n)} - \gamma\frac{1}{N_T}\sum_{n=0}^{N_T-1} \lambda^{(n+1)}\sigma'(\omega_{11}^{(n)}m^{(n)} + \omega_{21}^{(n)}q^{(n)} + b_1^{(n)}) \quad (2.32)$$

$$b_2^{(n+1)} = b_2^{(n)} - \gamma\frac{1}{N_T}\sum_{n=0}^{N_T-1} \mu^{(n+1)}\sigma'(\omega_{12}^{(n)}m^{(n)} + \omega_{22}^{(n)}q^{(n)} + b_2^{(n)}) \quad (2.33)$$

Finally, also in this framework, we can obtain the following algorithm to reach optimal parameters values:

Algorithm 2: Forward update of the parameters of the 2D SimResNet

Initialize $\omega^{(0)}$ and $b^{(0)}$ randomly
Generate SGD_set **for** $iter \leftarrow 1$ **to** n_iter **do**
  Propagate training data with the current parameters
  Compute the loss with respect to SGD_set
  Compute the new updates with formulas (2.26), (2.27), (2.28), (2.29), (2.30), (2.31), (2.32), (2.33)
end

Regression problem - multivariate case

As already explained, one of the main advantages working with neural networks is the easier training procedure when the model deals with data in high dimensional spaces.
In fact, one of the goal of this thesis is to backs up with this fact, conluding with the potential extension in higher dimensional data.
The equation $A^T x = c$ represents a system of linear equations. Each equation corresponds to a row of the matrix A^T multiplied by the vector x, and the result of this multiplication is compared to the corresponding element in the vector c. If these results match, then the vector x lies on the hyperplane defined by the equation $A^T x = c$.
In the context of geometry, a hyperplane is a flat affine subspace of one dimension less than the ambient space it's in: in a two-dimensional space, a hyperplane is a line, while in a three-dimensional space, it's a flat plane.
The normal vector to this hyperplane is given by the coefficients of the matrix A, and c determines the offset of the hyperplane from the origin.
Let's suppose to work with data $x_i \in \mathbb{R}^d \quad i = 1, \dots, N$. The model we are considering aims to find an hyperplane abled to fit all data.
In other words, the goal of this learning process is to train the weights in order to represent data with the propagated hyperplane given by the matrix $A^{(N_T)}$ (including the scalar c) which is the output of the network.

For computational purposes, we will consider the vector A as follows

$$A = [a_1, a_2, \dots a_d, a_{d+1}] \in \mathbb{R}^{d+1}$$

where a_{d+1} corresponds to the coefficient c.
For the same reason, the vector x is

$$x = [x_1, x_2, \dots x_d, -1] \in \mathbb{R}^{d+1}$$

in order to satisfy the hyperplane equation.

The *optimal control problem* can be explained by:

$$
\begin{aligned}
&\arg\min_{\omega,b} \frac{1}{n^*}\sum_{k=1}^{n^*}\frac{1}{2}\|A^{(N_T)}x_k\|_2^2 \\
s.t\quad & a_1^{(n+1)} = a_1^{(n)} + h\cdot\sigma(\omega_{11}^{(n)}a_1^{(n)} + \omega_{21}^{(n)}a_2^{(n)} + \cdots + \omega_{d+1,1}^{(n)}a_{d+1}^{(n)} + b_1^{(n)}) \\
& a_2^{(n+1)} = a_2^{(n)} + h\cdot\sigma(\omega_{12}^{(n)}a_1^{(n)} + \omega_{22}^{(n)}a_2^{(n)} + \cdots + \omega_{d+1,2}^{(n)}a_{d+1}^{(n)} + b_2^{(n)}) \\
& \quad\vdots \\
& a_d^{(n+1)} = a_d^{(n)} + h\cdot\sigma(\omega_{1d}^{(n)}a_1^{(n)} + \cdots + \omega_{dd}^{(n)}a_d^{(n)} + \omega_{d+1,d}^{(n)}a_{d+1}^{(n)} + b_d^{(n)}) \\
& a_{d+1}^{(n+1)} = a_{d+1}^{(n)} + h\cdot\sigma(\omega_{1,d+1}^{(n)}a_1^{(n)} + \cdots + \omega_{d,d+1}^{(n)}a_d^{(n)} + \omega_{d+1,d+1}^{(n)}a_{d+1}^{(n)} + b_{d+1}^{(n)}) \\
& A^o \in \mathbb{R}^{d+1}
\end{aligned}
$$

Which in compact form is:

$$
\begin{aligned}
&\arg\min_{\omega,b} \frac{1}{n^*}\sum_{k=1}^{n^*}\frac{1}{2}\|A^{(N_T)}x_k\|_2^2 \\
s.t\quad & A^{(n+1)} = A^{(n)} + h\cdot\sigma(W^{(n)} * A^{(n)} + b^{(n)}) \\
& A^o \in \mathbb{R}^{d+1}
\end{aligned}
$$

where $W \in \mathbb{R}^{d+1\times d+1}$ is the weights' matrix, $b \in \mathbb{R}^{d+1}$ is the bias vector.

So we can show the general network architecture, which depends on the given dimensionality:

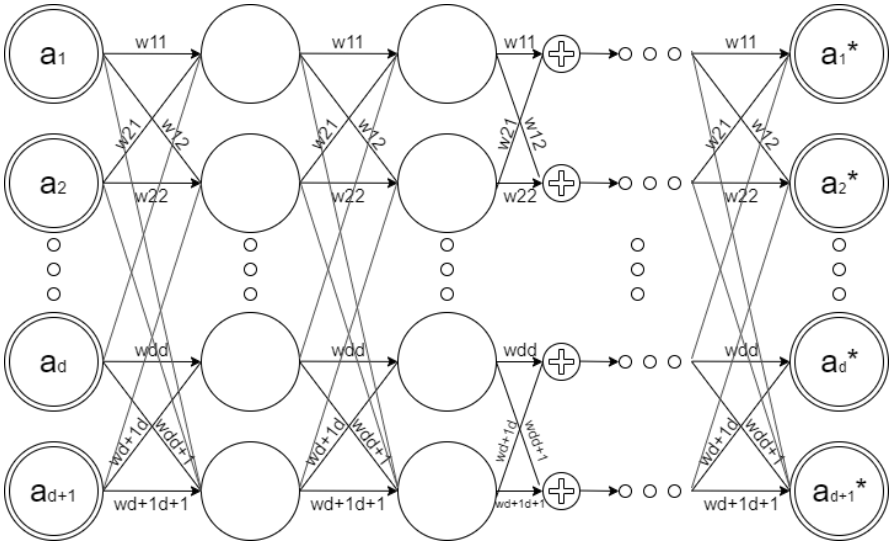


Figure 2.1: Architecture of the general multivariate case

The precise derivation of the sensitivity measure, encompassing parameter updates and the optimality conditions via the Lagrangian approach, can be explored directly within the Matlab code provided in *Appendix E*.

Chapter 3

Boltzmann-type formulation of SimResNet

In this section, we introduce a Boltzmann-type neural network model and explore its asymptotic limit, which leads to the formulation of a Fokker-Planck equation. To get a preliminary paronamic on the equation observe the *Appendix C*.

Our primary objective is to investigate the Fokker-Planck equation's capability to describe specific target distributions, depending on the selection of parameters and activation functions. We propose a modified version of the SimResNet, incorporating noisy dynamics, motivated by stochastic neural networks that incorporate uncertainty in activation outputs.
The long-term behavior of Boltzmann-type equations can be conveniently studied in the grazing limit regime, naturally leading to a Fokker-Planck equation which allows us to derive non-trivial steady-state distributions, and its steady states depend on various network parameters. These parameters need to be carefully chosen to best fit the desired target distribution.
In essence, this modification of the microscopic dynamics, inspired by stochastic residual neural networks, results in a Fokker-Planck formulation of the network. This formulation shifts the paradigm from traditional network training to the task of fitting a distribution: instead of training the network, we impose conditions on the network's parameters and the choice of activation functions to match a specific distribution.
This novel perspective offers a promising avenue for tackling problems where distribution fitting is more relevant than traditional training paradigms.

We established that the mean-field equation can be obtained as a suitable asymptotic limit of the Boltzmann-type space homogeneous kinetic equation. This means that we can derive the mean-field equation by defining instantaneous microscopic interactions that emerge from the continuous dynamics of the system governed by the Boltzmann-type kinetic equation, this transition is made possible by the convergence of the system's behavior in the asymptotic limit, where the detailed particle interactions are replaced by averaged, mean-field interactions.

We show that this is true also for the mean-field limit (2.3) of the neural differential equation (2.2) by suitably defining instantaneous microscopic interactions emerging from the continuous dynamics.
In the case of one-dimensional measurements, we can derive the mean-field equation from a linear Boltzmann-type equation. The system of ordinary differential equations (2.2) can be reformulated as an interaction rule:

$$x^* = x + \sigma(\omega(t)x + b(t)) \tag{3.1}$$

Here, the terms x^* and x represent the post- and pre-collision states, respectively.
It's important to note that even though this equation appears to be independent of other particles or measurements, the interactions are actually embedded in the parameters ω and b: these parameters are chosen based on the entire dataset, incorporating the collective behavior of all measurements.

In this way, by employing an explicit Euler discretization with a time step of $\Delta t = 1$, we obtained the aforementioned interaction rule from equation (2.2).

We aim now in deriving an evolutionary model resorting to the law (3.1) derived previously: to that aim, let us observe that the variation of the density $f(t, x)$ obeys a Boltzmann-type equation (10) which weak form corresponding to (3.1) reads

$$\frac{d}{dt}\int_{\mathbb{R}} \Phi(x) f(t,x) dx = \frac{1}{\tau}\int_{\mathbb{R}} \Big(\Phi(x^*) - \Phi(x)\Big) f(t,x) dx \tag{3.2}$$

where τ represents the interaction rate, i.e. quantifies how frequently interactions occur, and $\Phi \in \mathbb{C}_0^\infty(\mathbb{R})$ is a test function. See *Appendix C* for the entire derivation.
The weak form of the Boltzmann-type equation corresponds to the integral form of the governing equation. It is derived by considering the action of the equation on a test function, and it is often used when dealing with distribution functions and particle interactions in a probabilistic sense.

The Boltzmann-type description (3.2) offers a valuable advantage, namely, the ability to explore various asymptotic scales beyond the mean-field regime[1]. These distinct scales can yield equations exhibiting non-trivial steady states, which can be analytically characterized. Notably, these steady states are contingent upon the parameters ω and b.
This opens up a novel perspective on training, wherein the selection of network parameters and activation functions can be guided by the objective of matching the target distribution.

To investigate steady state profiles when dealing with arbitrary activation functions, we adopt a specific strategy: we introduce stochasticity into the microscopic interaction rule (3.1) and subsequently apply a grazing collision

[1]The Boltzmann-type description's versatility in capturing diverse interactions (collision dynamics, diffusion processes, and more), accommodating noise, and permitting continuum limits makes it a powerful tool for investigating a wide range of asymptotic scales beyond the mean-field regime

limit, i.e. a mathematical approximation that simplifies the analysis of systems with infrequent and nearly tangent interactions. This limit transforms the dynamics into a Fokker-Planck-type equation, from which it is possible to reckon explicitly the stationary solution.

3.1 Fokker-Planck description of the SimResNet

As explained before, the idea is to add noisy interaction in (3.1) in order to recover a broader class of steady states, in addition to the trivial ones described by the original mean-field equation.
Including the random variable $\eta \sim \mathcal{N}(0, \nu^2)$, the diffusion function $K(x)$ and considering ϵ the small parameter which wheights the strenght of the interactions, the interacting rule modifies as follow:

$$x^* = x + \epsilon\sigma(\omega(t)x + b(t)) + \sqrt{\epsilon}K(x)\eta \tag{3.3}$$

Observe that for $K(x) = 0$ it corresponds to (2.1) with $\epsilon = \Delta t$ and to (3.1) with $\epsilon = 1$.

This kind of structure is well known as *Euler-Maruyama*[2]: the deterministic part of the stochastic differential equation (the part without noise) is approximated using a standard Euler update, while the stochastic part (the part involving random noise) is simulated by adding random increments at each time step.

From a neural network perspective, this modeling approach draws inspiration from stochastic neural networks that incorporate stochastic output layers. Stochastic neural networks introduce random variations into the network's computations: one common method to represent these variations is by employing stochastic output layers, which effectively capture and quantify uncertainty in the network's activations.

One notable advantage of such stochastic neural networks is their utility in optimization procedures. In particular, the inherent randomness in these networks can help them escape local minima during training, contributing to more effective optimization.

In alignment with this concept, we extend our modeling approach by introducing uncertainty not only in the network's activations but also in its forward propagation. Consequently, our modified framework deviates from the deterministic interactions described by (3.1), allowing for uncertainty in the computation of certain states during forward propagation.

[2]The "Maruyama" part of the name is in honor of the Japanese mathematician Ryogo Maruyama, who contributed significantly to the development and analysis of numerical methods for SDEs.

As previously mentioned, our objective is to apply the grazing collision limit by rescaling time ($t = \epsilon t$) with $\epsilon \to 0$. This transformation leads to the following Fokker-Planck equation, which describes the evolution of the scaled probability distribution $f(t, x)$:

$$\partial_t f(t,x) + \partial_x[\mathcal{B} f(t,x) - \mathcal{D}\partial_x f(t,x)] = 0 \tag{3.4}$$

Here, the *interaction operator* $\mathcal{B}$ and the *diffusive operator* $\mathcal{D}$ take the forms:

$$\mathcal{B} = \sigma(\omega(t)x + b(t)) - \frac{\nu^2}{2}\partial_x K^2(x), \quad \mathcal{D} = \frac{\nu^2}{2}K^2(x)$$

In summary, our approach draws inspiration from *stochastic neural networks*, introducing uncertainty into both activations and forward propagation. This novel perspective offers unique advantages and leads to a Fokker-Planck equation that characterizes the evolution of the scaled probability distribution as the system evolves.
The deep extraction of (3.4) from (3.2) through the stochastic interaction rule (3.3) can be studied in the following.

First, some intermediate results are calculated from (3.3):

$$\begin{aligned}
\mathbb{E}[x^* - x] &= \mathbb{E}[\epsilon\sigma(\omega(t)x + b(t)) + \sqrt{\epsilon}K(x)\eta] \\
&= \mathbb{E}[\epsilon\sigma(\omega(t)x + b(t))] + \mathbb{E}[\sqrt{\epsilon}K(x)\eta] \\
&= \mathbb{E}[\epsilon\sigma(\omega(t)x + b(t))] + \sqrt{\epsilon}K(x)\mathbb{E}[\eta] \\
&= \epsilon\sigma(\omega(t)x + b(t))
\end{aligned}$$

$$\begin{aligned}
\mathbb{E}[(x^* - x)^2] &= \mathbb{E}[\big(\epsilon\sigma(\omega(t)x + b(t)) + \sqrt{\epsilon}K(x)\eta\big)^2] \\
&= \mathbb{E}[\epsilon^2\sigma^2(\omega(t)x + b(t)) + \epsilon K^2(x)\eta^2 + 2\epsilon\sqrt{\epsilon}\sigma(\omega(t)x + b(t))K(x)\eta] \\
&= \mathbb{E}[\epsilon^2\sigma^2(\omega(t)x + b(t))] + \mathbb{E}[\epsilon K^2(x)\eta^2] + \mathbb{E}[2\epsilon\sqrt{\epsilon}\sigma(\omega(t)x + b(t))K(x)\eta] \\
&= \epsilon^2\sigma^2(\omega(t)x + b(t)) + \epsilon K^2(x)\mathbb{E}[\eta^2] + 2\epsilon\sqrt{\epsilon}\sigma(\omega(t)x + b(t))K(x)\mathbb{E}[\eta] \\
&= \epsilon^2\sigma^2(\omega(t)x + b(t)) + \epsilon K^2(x)\nu^2
\end{aligned}$$

Then, to get some insight into the time evolution of the model above we use a standard procedure proceeding with the second order Taylor expansion of $\Phi(x^*)$ centered at x:

$$\Phi(x^*) \approx \Phi(x) + (x^* - x)\Phi'(x) + \frac{(x^* - x)^2}{2}\Phi''(x) + \mathcal{R}(x)$$

Applying mean value and using previous results we can write:

$$\begin{aligned}\mathbb{E}[\Phi(x^*)-\Phi(x)] &\approx \mathbb{E}\big[(x^*-x)\Phi'(x)+\frac{(x^*-x)^2}{2}\Phi''(x)+\mathcal{R}(x)\big]\\ &= \mathbb{E}[(x^*-x)]\Phi'(x)+\frac{\mathbb{E}[(x^*-x)^2]}{2}\Phi''(x)+\mathbb{E}[\mathcal{R}(x)]\\ &= \epsilon\sigma(\omega(t)x+b(t))\Phi'(x)+\frac{\epsilon^2\sigma^2(\omega(t)x+b(t))+\epsilon K^2(x)\nu^2}{2}\Phi''(x)+\mathcal{R}(x)\\ &= \epsilon\bigg(\sigma(\omega(t)x+b(t))\Phi'(x)+\frac{\epsilon\sigma^2(\omega(t)x+b(t))+K^2(x)\nu^2}{2}\Phi''(x)\bigg)+\mathcal{R}(x)\end{aligned}$$

Due to the randomness of (3.2) we aim to study the evolution of the following relation:

$$\frac{d}{dt}\int_{\mathbb{R}}\Phi(x)f(t,x)dx=\mathbb{E}\bigg[\frac{1}{\tau}\int_{\mathbb{R}}\Big(\Phi(x^*)-\Phi(x)\Big)f(t,x)dx\bigg] \tag{3.5}$$

It particular, extending the right side term and considering the rescaling $\frac{1}{\tau}=\frac{1}{\epsilon}$, we can write:

$$\begin{aligned}&\mathbb{E}\bigg[\frac{1}{\tau}\int_{\mathbb{R}}\Big(\Phi(x^*)-\Phi(x)\Big)f(t,x)dx\bigg]=\frac{1}{\tau}\int_{\mathbb{R}}\mathbb{E}\big[\Phi(x^*)-\Phi(x)\big]f(t,x)dx\\ &=\frac{1}{\tau}\int_{\mathbb{R}}\bigg[\epsilon\bigg(\sigma(\omega(t)x+b(t))\Phi'(x)+\frac{\epsilon\sigma^2(\omega(t)x+b(t))+K^2(x)\nu^2}{2}\Phi''(x)\bigg)+\mathcal{R}(x)\bigg]f(t,x)dx\\ &\simeq\frac{1}{\epsilon}\int_{\mathbb{R}}\epsilon\sigma(\omega(t)x+b(t))\Phi'(x)f(t,x)dx+\frac{1}{2\epsilon}\int_{\mathbb{R}}\big(\epsilon^2\sigma^2(\omega(t)x+b(t))+\epsilon K^2(x)\nu^2\big)\Phi''(x)f(t,x)dx\\ &=\int_{\mathbb{R}}\sigma(\omega(t)x+b(t))\Phi'(x)f(t,x)dx+\frac{1}{2}\int_{\mathbb{R}}\big(\epsilon\sigma^2(\omega(t)x+b(t))+K^2(x)\nu^2\big)\Phi''(x)f(t,x)dx\end{aligned}$$

Applying the grazing limit $\epsilon\to 0$ we obtain:

$$\int_{\mathbb{R}}\sigma(\omega(t)x+b(t))\Phi'(x)f(t,x)dx+\frac{1}{2}\int_{\mathbb{R}}K^2(x)\nu^2\Phi''(x)f(t,x)dx$$

Now, starting from (3.5) and passing to the strong form (deriving by parts) we obtain the following relation:

$$\int_{\mathbb{R}}\bigg[\partial_t f\big[t,x)+\partial_x\big[\sigma(\omega(t)x+b(t)f(t,x))\big]-\partial_{xx}\bigg[\frac{\nu^2}{2}K^2(x)f(t,x)\bigg]\bigg]\Phi(x)dx=0$$

which implies that

$$\begin{aligned}&\partial_t f\big(t,x)+\partial_x\big[\sigma(\omega(t)x+b(t)f(t,x))\big]-\partial_{xx}\bigg[\frac{\nu^2}{2}K^2(x)f(t,x)\bigg]=0\\ \iff&\partial_t f\big(t,x)+\partial_x\bigg[\sigma(\omega(t)x+b(t)f(t,x))-\frac{\nu^2}{2}\partial_x\big[K^2(x)f(t,x)\big]\bigg]=0\end{aligned}$$

Finally, we can obtain the extend form of (3.4):

$$\partial_t(f(t,x)) + \partial_x\big[\big(\sigma(\omega(t)x+b(t)) - \frac{\nu^2}{2}\partial_x K^2(x)\big)f(t,x) - \frac{\nu^2}{2}K^2(x)\partial_x f(t,x)\big] = 0$$

The computation of the grazing collision limit offers a substantial advantage: it replaces the classical integral formulation of the Boltzmann collision term with differential operators.
This transition simplifies the characterization of the steady-state solution, denoted as $f^\infty(x)$, for the equation (3.4).
When the target distribution aligns well with a steady-state solution of the Fokker-Planck equation, it becomes possible to readily determine the values of weight parameters, biases, and the activation function.
Due to this reason this streamlined approach presents a significant computational benefit, especially when contrasted with the conventional training of neural networks, which typically involves complex learning processes and backpropagation.

In the forthcoming section, we will delve into the characterization of steady states and present illustrative examples that can be effectively described by the solution to (3.4) in the context of long-term behavior. Our goal is to establish specific conditions governing the parameters and the choice of activation functions that enable an accurate fit. This approach fundamentally eliminates the need for conventional training methods, marking a substantial departure from established practices in neural network development.

3.1.1 Steady state characterization

Leveraging the grazing collision limit it is possible to write an analytical steady state solution of the Fokker-Planck equation (3.4):

$$f^\infty(x) = \frac{C}{K^2(x)} exp\left(\int \frac{2\sigma(\omega^\infty x + b^\infty)}{\nu^2 K^2(x)} dx\right) \tag{3.6}$$

where $C \in \mathbb{R}$ is determined by mass conservation, i.e. $\int_{\mathbb{R}} f^\infty(x)dx = 1$.

The existence and the explicit shape of the steady state is determined by the specific choice of the activation function $\sigma(\cdot)$, of the diffusion function $K(\cdot)$ and of the parameters ω^∞, b^∞.
Let us discuss two classical cases:

(a) Assuming the target $h(x)$ distributed as a Gaussian.
Choosing $\sigma(x) = \sigma_I(x)$ and $K(x) = 1$ we obtain the following

$$f^\infty(x) = C \cdot exp\left(\frac{\omega^\infty}{\nu^2}x^2 + 2\frac{b^\infty}{\nu^2}x\right)$$

which yields a suitable approximation of $h(x)$ provided that $\omega^\infty < 0$ and $b^\infty = 0$.

Moreover, due to mass conservation:

$$C = \frac{\sqrt{-\frac{\omega^\infty}{\nu^2}} exp\left(\frac{(b^\infty)^2}{\omega^\infty \nu^2}\right)}{\sqrt{\pi}}$$

which is defined for $\omega^\infty < 0$.

(b) Assuming the target $g(x)$ distributed as a Inverse Gamma. Choosing $\sigma(x) = \sigma_I(x)$ and $K(x) = x$ we obtain the following

$$f^\infty(x) = \begin{cases} 0 & if \quad x \leq 0 \\ \frac{C}{x^{1+\mu}} exp\big(\frac{\mu-1}{x}\frac{b^\infty}{\omega^\infty}\big) & if \quad x > 0 \end{cases}$$

which yields a suitable approximation of $g(x)$ where $\mu := 1 + \frac{2\omega^\infty}{\nu^2}$ and provided that $\omega^\infty < 0$ and $b^\infty > 0$ in order to obtain the distribution. Moreover

$$C = \frac{\big((1-\mu)\frac{\omega^\infty}{b^\infty}\big)^\mu}{\Gamma(\mu)}$$

where $\Gamma(\cdot)$ denotes the Gamma function.

Chapter 4

Numerical simulations

In this section, we explore two fundamental applications of machine learning algorithms: classification and regression problems. We utilize these applications to evaluate the effectiveness of the weight and bias update algorithm derived from the sensitivity analysis, as discussed in Section (2.4).

Next, we present a series of simulations based on both the mean-field and kinetic approaches, which leverage the statistical interpretation of the neural network process. Our focus lies in tackling classification and regression problems, which are representative of common scenarios encountered in machine learning applications.

Throughout these simulations, we employ the hyperbolic tangent as the chosen activation function, denoted as $\sigma(x) = \sigma_T(x) = \tanh(x)$. This activation function helps us achieve the desired behavior and computational capabilities within the neural network.

4.1 Classification task

Consider a classification problem as follows. We measure a quantity (e.g. the length of a vehicle) and we need to identify, or classify, the type of the object related to that measurement (e.g. *car* or *truck*).
Therefore, in a classification problem, the task of the neural network is to determine the type given a measurement. Keeping in mind the example of classifying vehicles from their length measurement, an experimental data set might look like Table 1 below, with a suitable scalar label for the classifiers.
We consider 50 vehicles with measured length between 2 and 8 obtained as (uniformly distributed realizations).
Considering the resulting table we can display the histogram of frequencies.

Since the classification task is dealing with two values (2 and 8) it means that the output of the mean field equation will be a two Dirac delta distribution located at 2 and 8.
In general, from a mean-field mathematical perspective, classification can be seen as the problem of driving the distribution of the input data to a distribution of Dirac delta functions.

quantity	interval	target
5	2 - 2.6	2
5	2.6 - 3.2	2
4	3.2 - 3.8	2
6	3.8 - 4.4	2
3	4.4 - 5	2
8	5 - 5.6	8
3	5.6 - 6.2	8
6	6.2 - 6.8	8
6	6.8 - 7.4	8
4	7.4 - 8	8

Figure 4.1: Data set for the classification problem

Figure 4.2: Frequencies of train data values

According to the dynamics explained in section (2.1), we can represent the network architecture we want to train in this way:

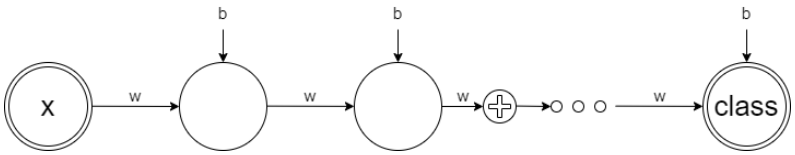


Figure 4.3: Architecture of this classification task. Although they are not displayed, skip connections occur at each step, *"class"*, in this case, is a value between [2,8]

In order to find the best parameters for the model, cross validation has been applied with respect to the number of network layers and the learning rate. The first is crucial in time evolving of the particles and the second one in updating weights and bias.

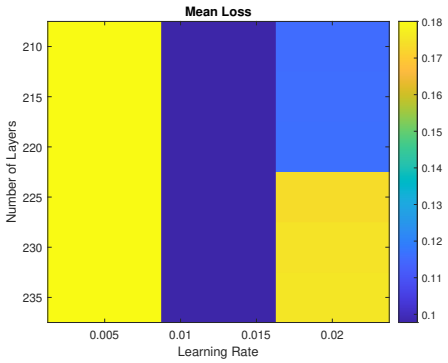


Figure 4.4: Cross validation to obtain best parameters, as we can see the dark blue zone enforce the lowest loss value

The result of cross validation ensures good performance for $L = 220$ and $\gamma = 0.01$.

Dynamics simulation has been computed with the above parameters and we can observe a good behaviour on particle level since there is a good division between values of different clusters, i.e. bigger values than 5 converge to 8 while smaller values converge to 2.

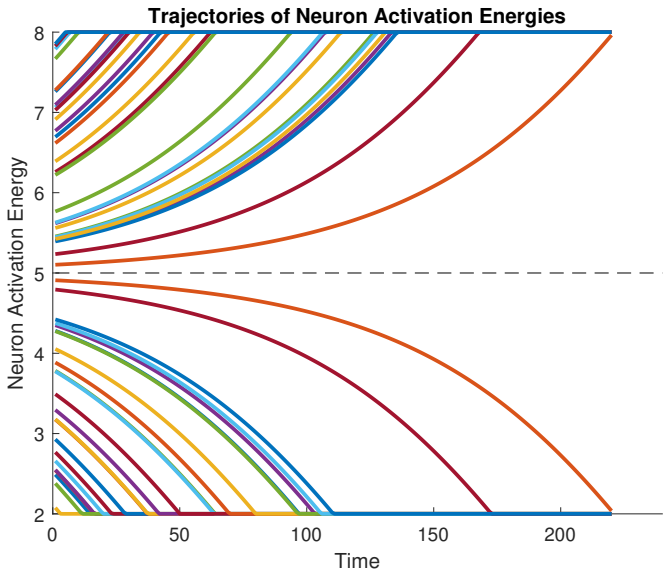
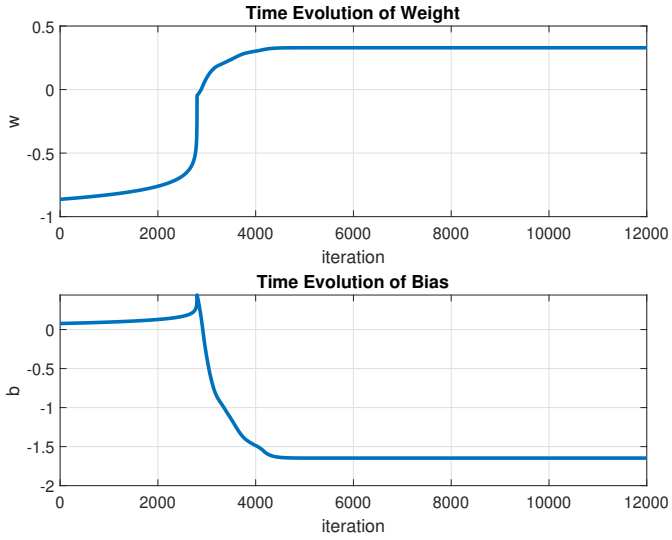


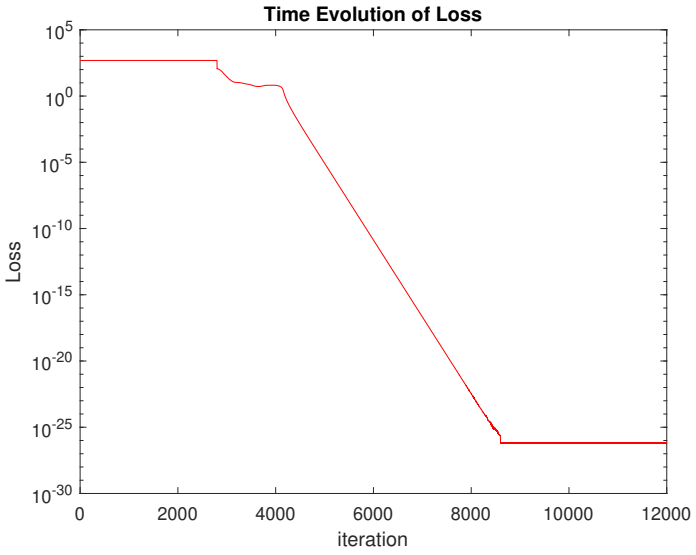
Figure 4.5: Trajectories of neurons, we can observe that more the particle is near to the crucial zone (close to 5) lower is the convergence

Then we can observe the behaviour of weight and bias which find a stabilization after a certain number of iterations.
These two are computed optimally with the goal of minimize the loss of type (2.20), the good result is proved by Figure (4.6b)

We can represent the evolution of the probability density function, observing that the final distribution is spread as a Dirac centered in two points, see Figure (4.7).



(a) Parameters start from an initial random value converging at iteration 4000, w = 0.43, b = -1.6



(b) Time behaviour of Loss function

Figure 4.6: Parameters' stabilization and loss minimization

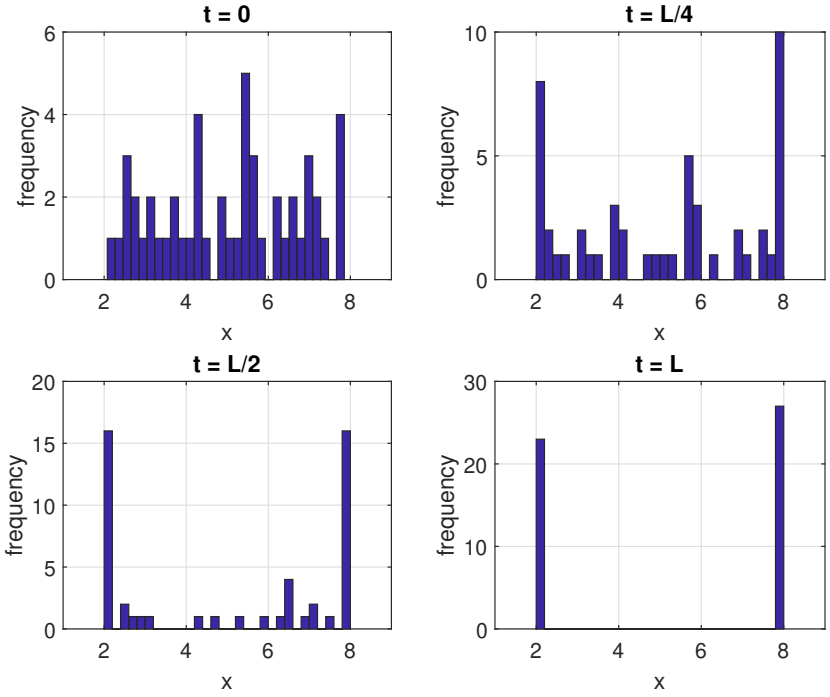


Figure 4.7: Probability density function distribution over time

4.2 Regression task

In this regression problem, we are dealing with data points represented in two dimensions, denoted as (m, q), where m represents the angular coefficient and q represents the intercept of the linear fit $y = mx + q$[1].
Focusing on the mean field equation (2.3) we can represent this dynamics with the following relation:

$$\partial_t f(t,m,q) + \partial_m(\sigma(\omega_{11}m + \omega_{21}q + b_1)f(t,m,q)) + \\ + \partial_q(\sigma(\omega_{12}m + \omega_{22}q + b_2)f(t,m,q)) = 0 \qquad (4.1)$$

which is a two-dimensional version of the Vlasov type equation where $f(t, m, q)$ describes the distribution function over time of the regression parameters.

We may have given measurements $(x_i, y_i) \quad i = 1, \ldots, N$ at fixed locations. These measurements might be disturbed possibly due to measurements errors as in the plot of Figure (4.8a).
They are generated in such a way they can be represented by the regression given by
$(m, q) = (1 + \psi_m, 0 + \psi_q)$, with ψ_m, ψ_q two gaussian distributed values with different mean and variance.

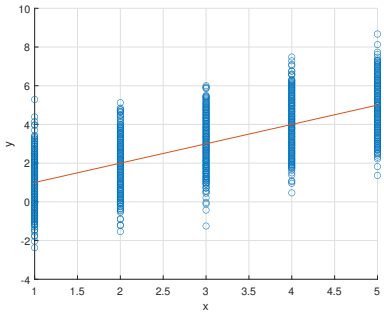
In a regression problem the task of the neural network is to find a linear

[1] m is called *slope* of the linear fit, while q is the intersection of the linear function with the y axis

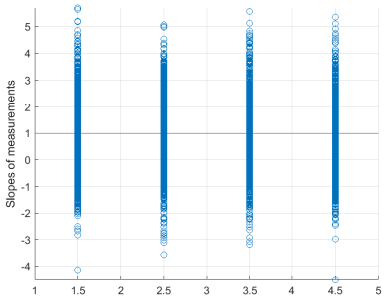
fit $y = mx + q$ of those data points, so the main objective of this problem is to train a neural network to converge the learned values of (m, q) to the values (m^*, q^*) which best fits the measurements.
In particular, these measurements are used to generate an *input data* of slopes and intercepts in the following way: for each pair of clouds of the type (x, y) and $(x + 1, y)$ two random values called (x_1, y_1) and (x_2, y_2) are picked and with the formulas

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$
$$q = y_1 - m \cdot x_1$$

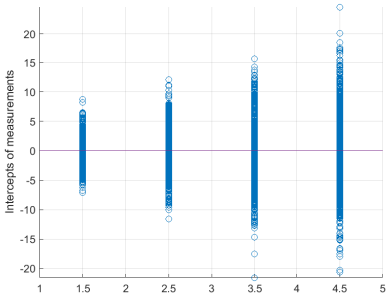
have been generated the set of parameters. See plots of Figure (4.8b)-(4.8c).



(a) Training data (x, y)



(b) Numerical slopes



(c) Numerical intercepts

Figure 4.8: Initial states regarding training dataset, numerical slopes and intercepts.

Figure (4.9) represents the initial distribution of couples (m, q) obtained from (4.8b)-(4.8c) which will be the training data.

The neural network's architecture is designed differently from the previous problem, as we now have two dimensions to consider, which means more weights need optimizing.
The neural network will learn to perform a regression task, where it takes the input data points $(m_k, q_k), \quad k = 1, \ldots, M$ and outputs the corresponding angular coefficient m^* and intercept q^* that best fit the given data measurements. To achieve this, we will employ stochastic gradient descent (SGD) as our op-

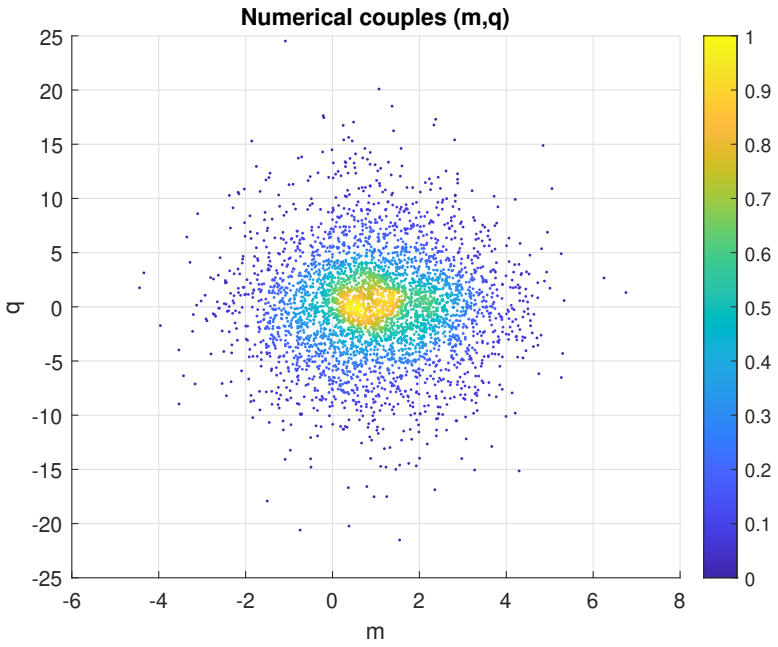


Figure 4.9: Numerical regression coefficients (m, q)

timization algorithm. We will randomly sample a subset of the data of cardinality n^*, i.e. $(x_h, y_h), \quad h = 1, \dots, n^*$, to calculate the loss function.
The loss function is defined as the mean squared error between the actual y_k and the predicted value, which is estimated using the learned $(m^{(N_T)}, q^{(N_T)})$ and the input x_k.
The objective is to minimize this loss function and converge the predicted (m, q) values to the desired couple (m^*, q^*).

Recalling what we saw in section (2.4), the optimal control problem for this regression task is given by:

$$\begin{aligned} &\arg\min_{\omega,b} \frac{1}{n^*} \sum_{k=1}^{n^*} \frac{1}{2} \| y_k - m^{(N_T)} \cdot x_k - q^{(N_T)} \|_2^2 \\ s.t \quad & m^{(n+1)} = m^{(n)} + \sigma(\omega_{11}^{(n)} m^{(n)} + \omega_{21}^{(n)} q^{(n)} + b_1^{(n)}) \\ & q^{(n+1)} = q^{(n)} + \sigma(\omega_{12}^{(n)} m^{(n)} + \omega_{22}^{(n)} q^{(n)} + b_2^{(n)}) \\ & (m, q)^o \in \mathbb{R} \times \mathbb{R} \end{aligned}$$

The network architecture is displayed in figure (4.10), during simulations has been considered a SimResNet with parameters $L = 500$ and $\gamma = 0.01$.
.

We can observe a good propagation behaviour, since all the input couples (m, q) converge in proximity of the desidered value $(m^*, q^*) = (1, 0)$. Moreover, after a certain number of iteration, in Figures (4.11a)(4.11b) we observe the stabilization of weights and biases, which is an important theoretical result.

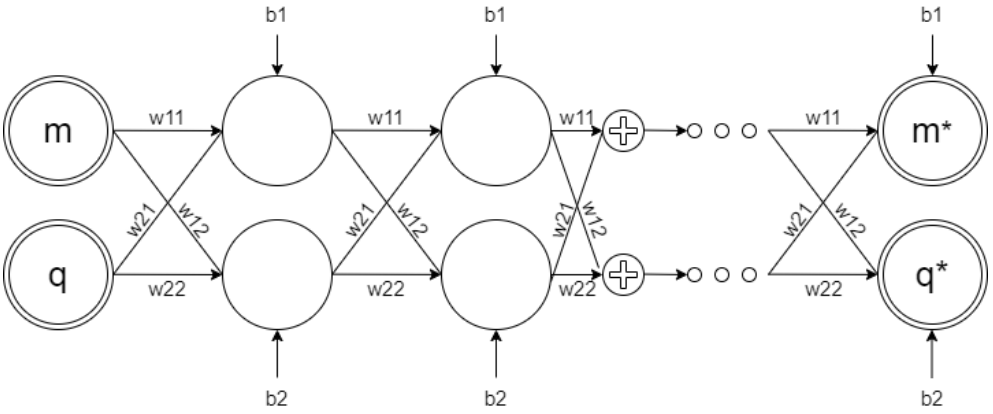
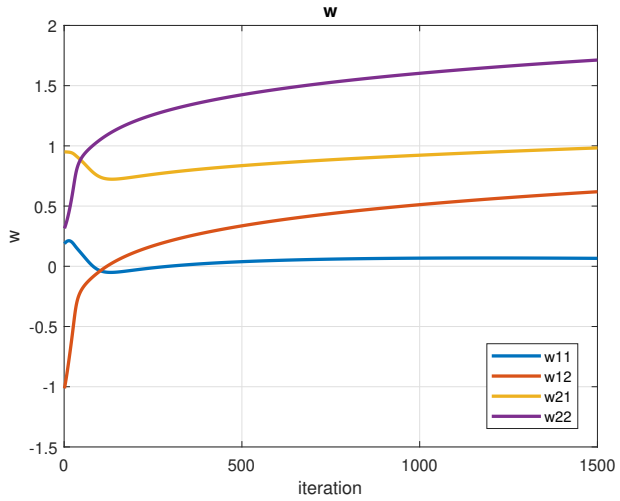
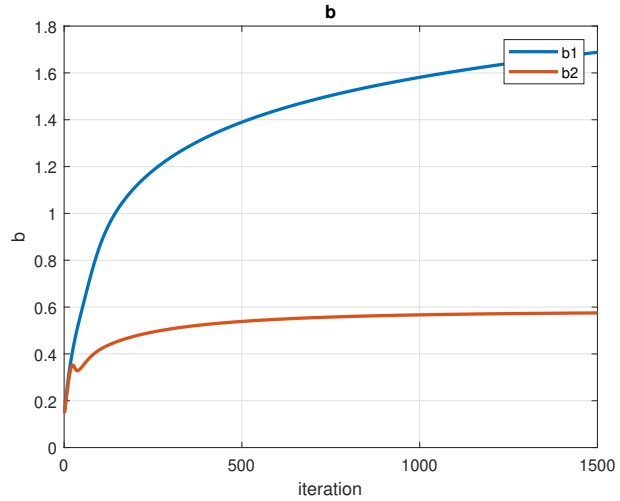


Figure 4.10: SimResNet Architecture of the 2D-regression task.



(a) Evolution of weights



(b) Evolution of biases

Figure 4.11: Caption for the whole figure containing both images

As a result, we obtain the goal of the training procedure minimizing the loss function described above, see Figure (4.12).

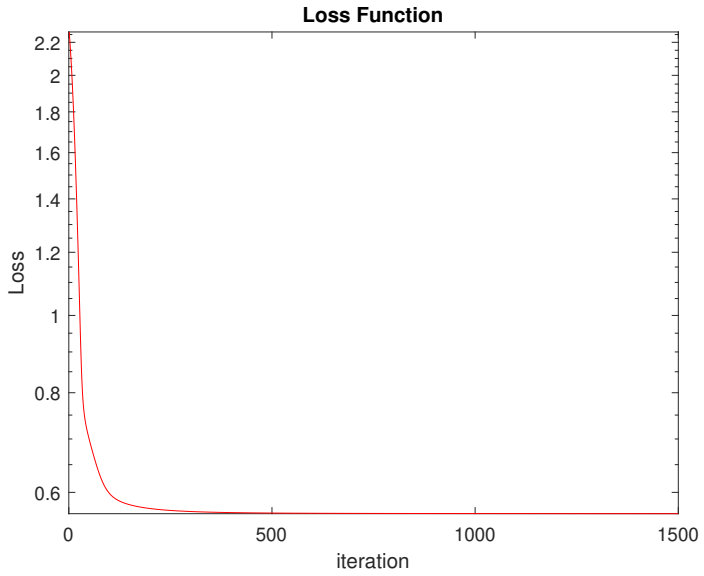


Figure 4.12: Loss function evolution of the Regression problem

Now, in order to visualize the real convergence of the model, the plots in Figure (4.16) represent the evolution of the propagation of couples (m_i, q_i), $i = 1, \ldots, M$ throught the network with parameters $\omega^* = (\omega^*_{11}, \omega^*_{12}, \omega^*_{21}, \omega^*_{22})$ and $b^* = (b^*_1, b^*_2)$.

An intriguing observation is evident: when training a model with a fixed number of layers L, subsequent testing of this model using the same L layers yields a final density distribution that reminds a Dirac distribution. Furthermore, when test data is propagated through a deeper network - specifically, when the model is assessed with an increased value for L — the resultant distribution converges even more profoundly towards a Dirac distribution. This result is important for the main objective of the network, which is to approximate the stationary distribution of the mean-field equation.

In the context of the mean-field equation, deeper layers correspond to prolonged temporal dynamics, askin to observing the system's evolution over extended time intervals.
The convergence towards a Dirac distribution when testing with deeper layers aligns with theoretical expectations. In essence, it implies that the model's predictions are converging towards a stable solution, representative of the system's long-term behavior. This convergence supports the notion that the network is effectively approximating the stationary distribution of the mean-field equation, as increasing the network's depth corresponds to exploring the system's behavior over extended temporal scales.

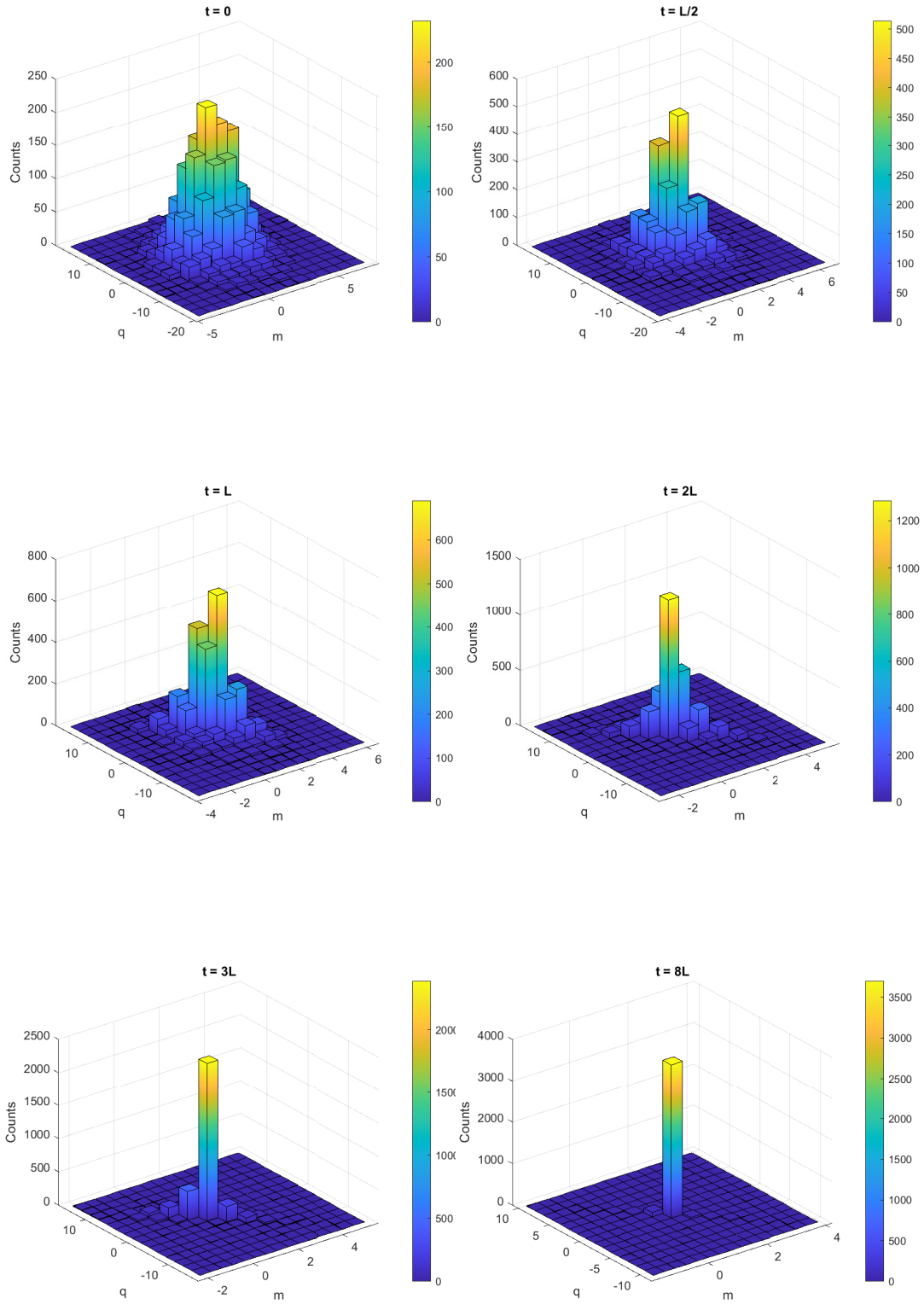


Figure 4.15: Evolution of density distribution function of couples (m, q) with respect to the number of layers.

4.3 Multivariate regression task

As already explained, one of the main advantages working with neural networks is the easier training procedure when the model deals with data in high dimensional spaces.
Despite this, for the sake of clarity, in our simulation we have chosen to set the dimensionality to $d = 3$.

This means that

$$A = [a_1, a_2, a_3, a_4] \in \mathbb{R}^4, \quad b = [b_1, b_2, b_3, b_4] \in \mathbb{R}^4$$

This approach entails the objective of aligning a set of hyperplanes to converge toward a designated target one denoted as $A_{\text{target}} \in \mathbb{R}^4$. More specifically, during the training procedure, the weights are learned based on a loss function that focuses on points deviating from A_{target}.

From a computational perspective, the objective of this task is to generate a set of hyperplanes generated by a collection of points located in the vicinity of the target hyperplane (see Figure (4.18)) and train the SimResNet with them. The ultimate goal is to ensure that, given a generic hyperplane, it converges towards the target one.

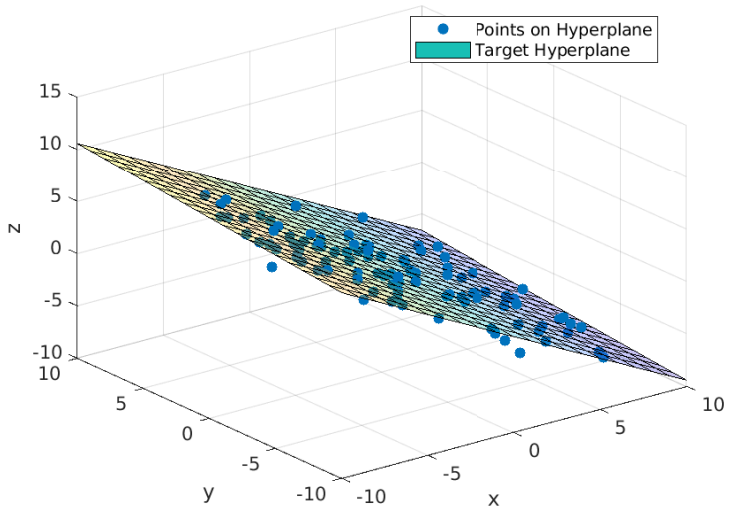


Figure 4.16: Target hyperplane and train points.

In the context of three-dimensional space, we can quantify the distance between two hyperplanes represented by vectors A_1 and A_2, which have the following form:

$$A_1 = [A_1(1), A_1(2), A_1(3), A_1(4)] \in \mathbb{R}^4$$
$$A_2 = [A_2(1), A_2(2), A_2(3), A_2(4)] \in \mathbb{R}^4$$

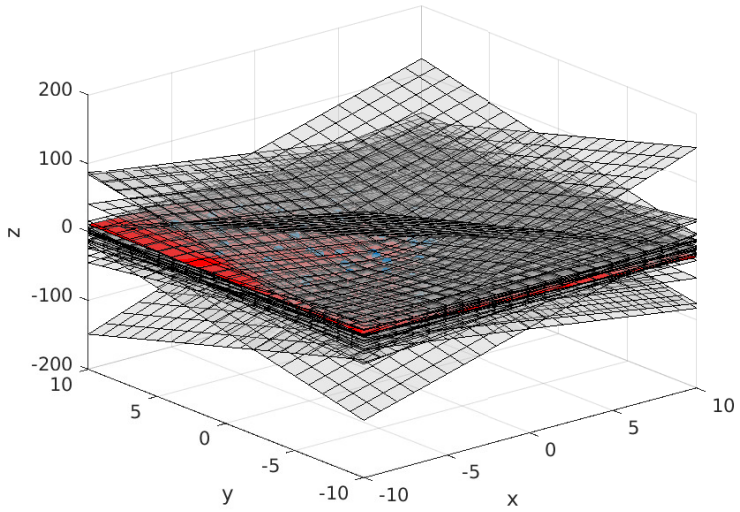


Figure 4.17: Generated hyperplanes by selecting three random points located in the vicinity of the target (red) hyperplane.

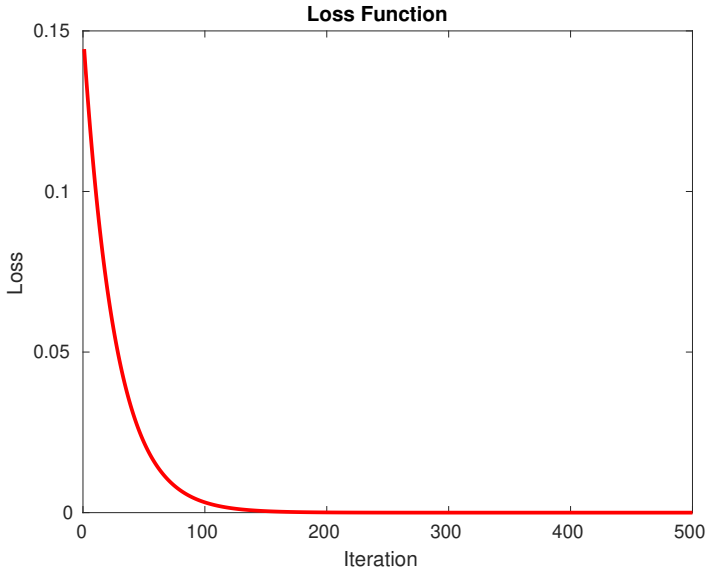


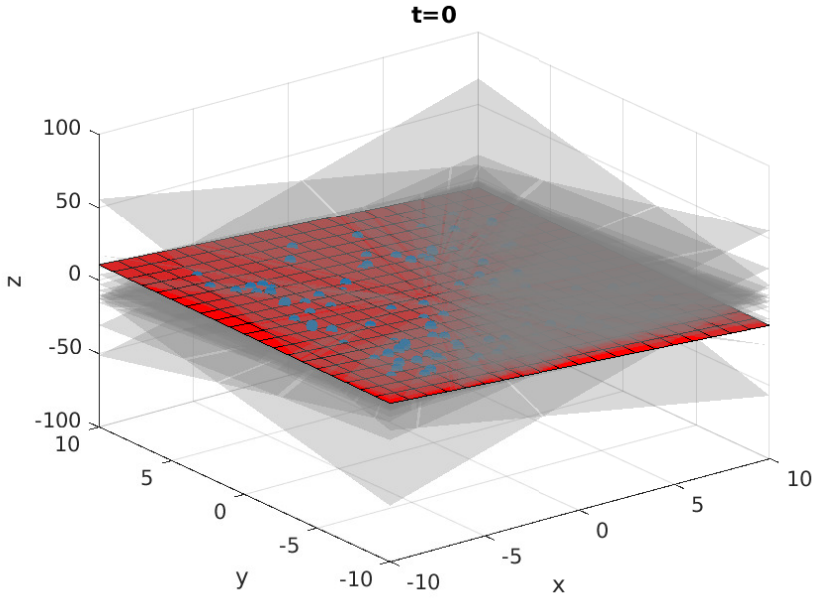
Figure 4.18: Loss during training process of the SimResNet.

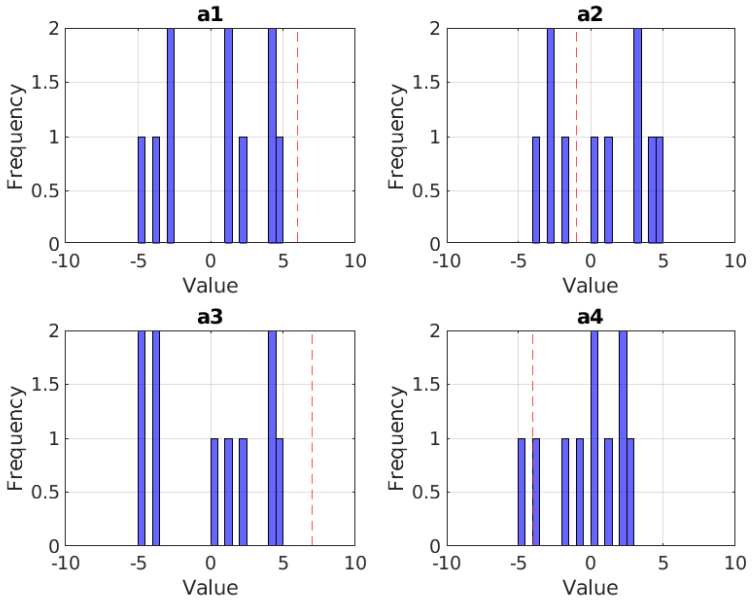
The formula for calculating this distance can be expressed as:

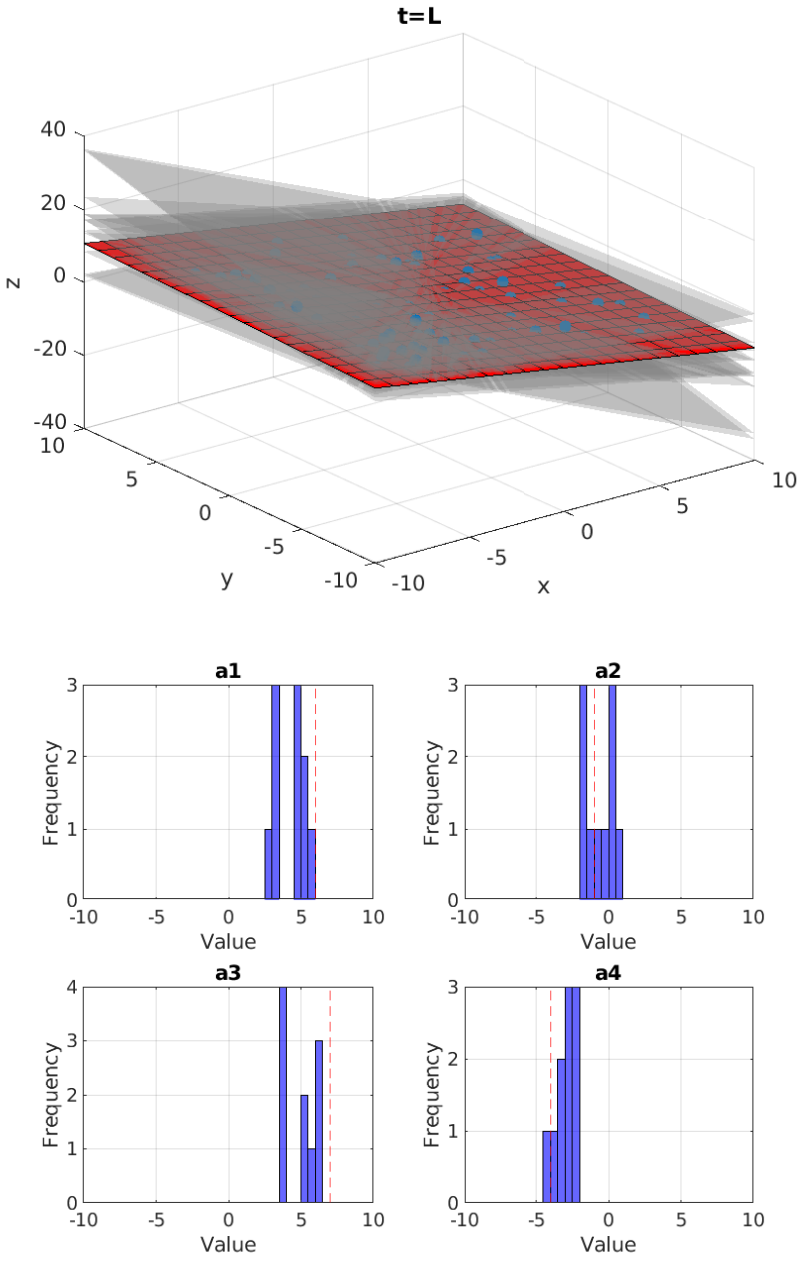
$$d(A_1, A_2) = \frac{|A_2(4) - A_1(4)|}{\sqrt{A_1(1)^2 + A_1(2)^2 + A_1(3)^2}}$$

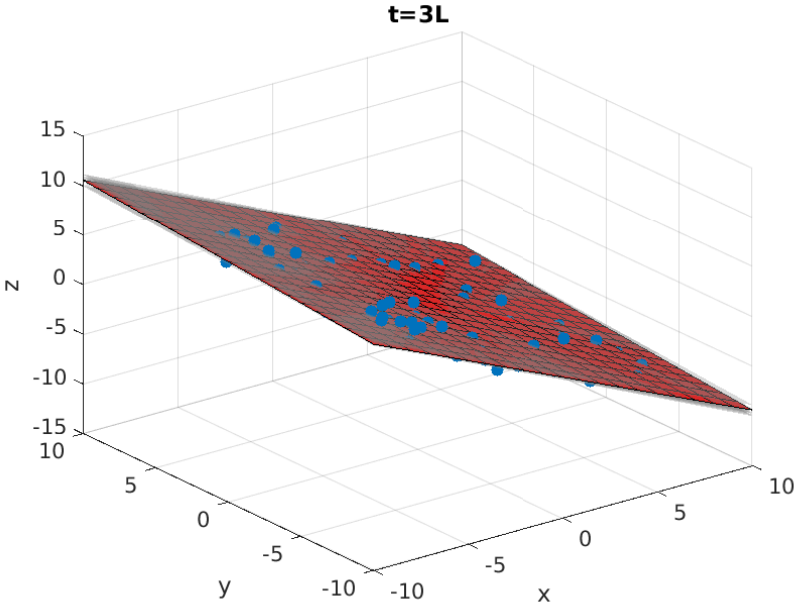
This formula quantifies the distance between the two hyperplanes based on their coefficients, taking into account the specific components of their equations in three-dimensional space.

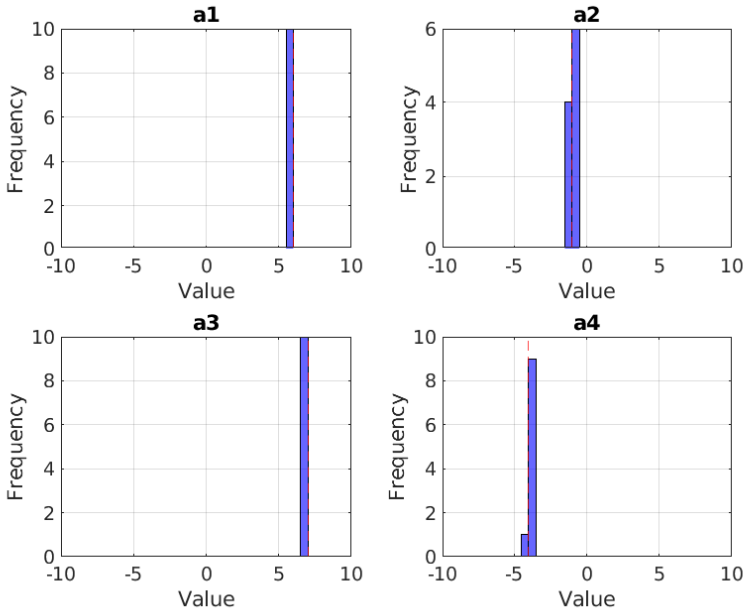
Now, we can analyze the propagation of given test hyperplanes through the network.






t=L
z
y
x
a1
a2
a3
a4
Frequency
Value


t=3L
z
15
10
5
0
-5
-10
-15
y
10
5
0
-5
-10
x
-10
-5
0
5
10


a1
a2
a3
a4
Frequency
Value
0
2
4
6
8
10
-10
-5
0
5
10

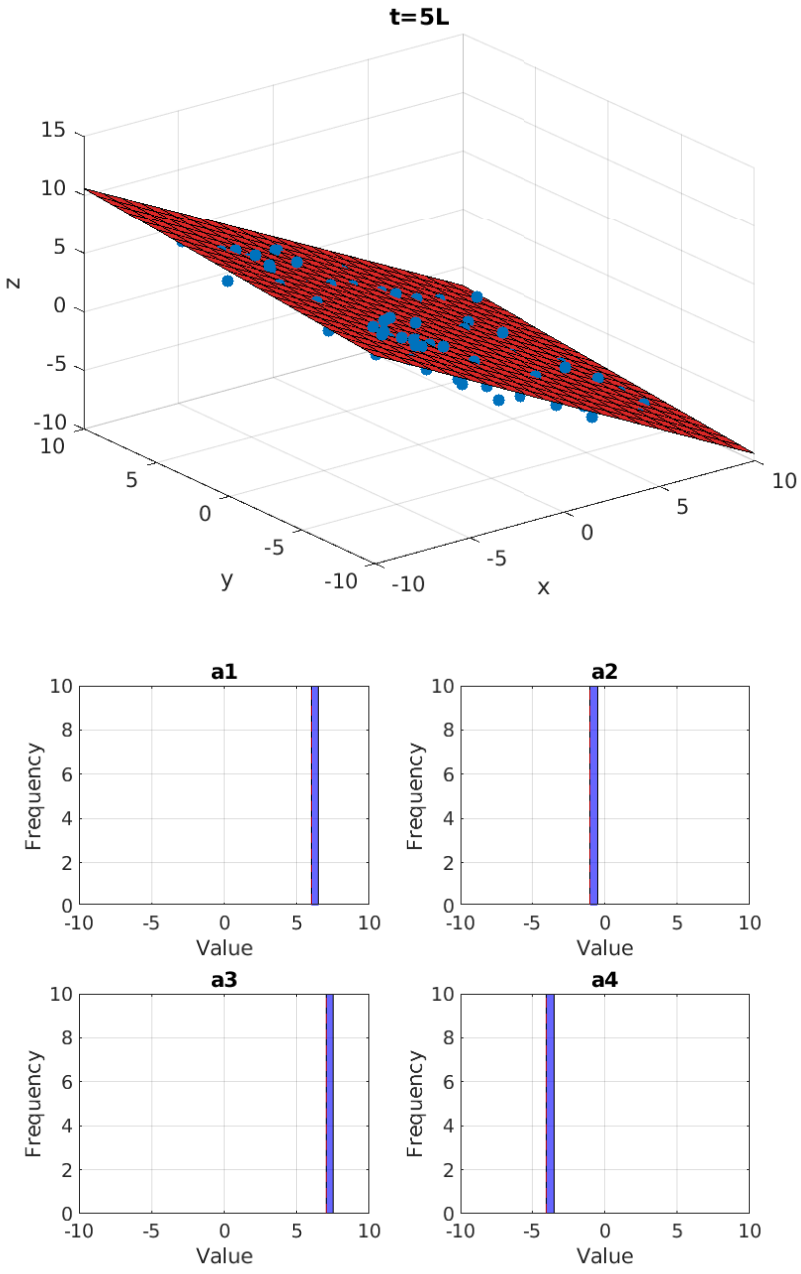


Figure 4.22: Evolution of hyperplanes' evolution with respect to the time propagation

We can observe that the trajectories of the hyperplane's coefficients, namely the vector A, converge to the values given by the vector A_{target}.
In this case, the steady-state solution will be a sum of four dirac, one for each dimension.

As said before, a good measure to capture the convergence is understand how far the propagated results from the target distribution are. The following plot support the fact that the distance between the propagated hyperplanes from A_{target} decreases with the increasing of the propagation time.

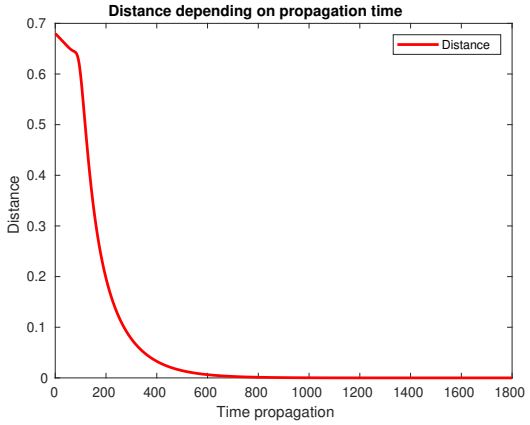


Figure 4.23: Mean distance between propagated hyperplanes and target one depending on time propagation.

4.4 Fokker-Planck equation

Case (a): Normal Gaussian distribution

The purpose of this simulation is to validate the findings presented in Section (3.1.1). Specifically, in the context of example *(a)*, our objective is to demonstrate the feasibility of fitting a normal Gaussian distribution as the desired outcome. We will adhere to the predetermined criteria for parameters, activation functions, and diffusion functions.

Here, we initialize the Fokker-Planck model (3.4) with a uniform distribution in the range $[-1, \frac{1}{2}]$ and allow it to evolve over time.

In Section (3.1.1), we concluded that the Fokker-Planck interpretation of the neural network can approximate a Gaussian distribution in the steady state under certain conditions. These conditions include using the identity activation function σ_I, setting ω^∞ to be less than zero, establishing b^∞ as zero, and employing $K(x)$ as a constant equal to 1. This approach enables us to steer any initial input towards the desired target distribution.

It's worth noting that, conversely, as demonstrated in Section (2.3), the mean-field neural network can only handle clustering tasks when using a hyperbolic tangent or identity activation function, provided that ω^∞ is less than zero and b^∞ is zero. In such cases, the distribution tends to a Dirac delta distribution over time. Consequently, it becomes impractical to fit a Gaussian distribution target using the deterministic (classical) SimResNet model.

In this context, stochastic neural networks with stochastic output layers, leading to a Fokker-Planck kinetic interpretation, exhibit improved capabilities for fitting target distributions other than Dirac delta distributions under the same conditions. In general, we can assert that stochastic neural networks are more effective for fitting target distributions that differ from Dirac delta distributions.

To be more technical, the simulation has been run[2] with the following values:

$$\omega = -1, \quad b = 0, \quad K(x) = 1, \quad \epsilon = 10^{-2}, \quad L = 100$$

According to these values and from the first example in section (3.1.1), the

[2]the data propagation follows the rule given by (3.3)

steady solution has the following form:

$$
\begin{aligned}
f^{\infty} &= \frac{\sqrt{-\frac{\omega^{\infty}}{\nu^2}}exp\left(\frac{(b^{\infty})^2}{\omega^{\infty}\nu^2}\right)}{\sqrt{\pi}}exp\left(\frac{\omega^{\infty}}{\nu^2}x^2 + 2\frac{b^{\infty}}{\nu^2}x\right) \\
&= \frac{\sqrt{-\frac{(-1)}{\nu^2}}exp\left(\frac{(0)^2}{-1\cdot\nu^2}\right)}{\sqrt{\pi}}exp\left(\frac{-1}{\nu^2}x^2 + 2\frac{0}{\nu^2}x\right) \\
&= \frac{\sqrt{\nu^{-2}}}{\sqrt{\pi}}exp\big(-x^2\cdot\nu^{-2}\big)
\end{aligned}
$$

Observe that, in the case we are dealing with a Standard Gaussian distribution target $h(x) \sim \mathcal{N}(0,1)$, the steady solution becomes:

$$f^{\infty} = \frac{1}{\sqrt{\pi}}exp(-x^2)$$

which analytically is exactly a Gaussian distribution centered in 0.

Figure (4.26) displays the evolution of the Fokker-Planck neural network's solution for various propagation time steps. The convergence towards the specified target distribution becomes evident after a certain number of time steps.

Furthermore, to maintain the inherent randomness of the model, the table below summarizes the mean and variance values of the final data distribution based on a specified number of simulations

Time $\times L$	Mean	Variance	Relative Error
0.5	-0.0298	1.0354	0.0326
1	-0.0275	1.0340	0.0308
1.5	-0.0085	1.0248	0.0167
2	-0.0095	1.0102	0.0099
2.5	0.0021	1.0102	0.0062
3	0.0013	1.0036	0.0026

Table 4.1: Average statistical values of final distributions over 50 simulations.

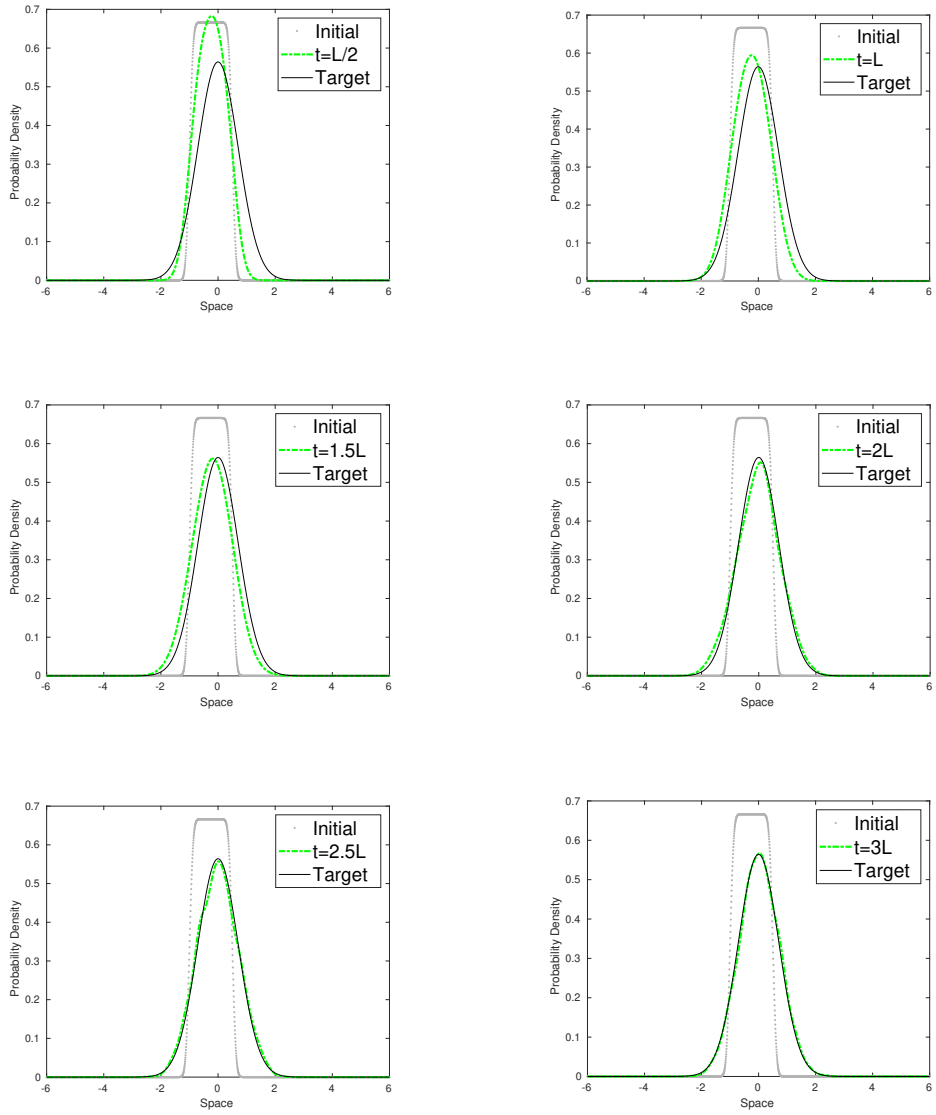


Figure 4.26: Solution of the Fokker-Planck neural network model at different propagation times.

Case (b): Inverse Gamma

Considering example *(b)*, our objective is to demonstrate the feasibility of fitting an inverse Gamma distribution as the desired outcome. As before, we will set the specific criteria for parameters, activation functions, and diffusion functions in order to reach the goal.

Here, we initialize the Fokker-Planck model (3.4) with a uniform distribution in the range $[-5, 30]$ and allow it to evolve over time.

In the theoretical section (3.1.1) we derived a significant conclusion. It was determined that, under specific conditions, the neural network's Fokker-Planck interpretation can effectively approximate an inverse Gamma distribution in the steady state. These conditions entail the use of the identity activation function, denoted as σ_I, the specification of ω^∞ as a negative value, the establishment of b^∞ as a positive value, and the utilization of $K(x)$ in its simplest form, i.e., $K(x) = x$.
By adopting this approach, we are empowered to guide any initial input towards achieving the desired target distribution.

To be more technical, the simulation has been run with the following values:

$$\omega = -0.5, \quad b = 1, \quad K(x) = x, \quad \nu^2 = 0.5, \quad \epsilon = 1^{-3}$$

According to these values the steady solution terms have the following form:

$$\mu = 1 + \frac{2\omega^\infty}{\nu^2} = 1 + \frac{2 \cdot (-0.5)}{0.5^2} = 3$$
$$C = \frac{\left((1-\mu)\frac{\omega^\infty}{b^\infty}\right)^\mu}{\Gamma(\mu)} = \frac{\left((1-3)\frac{-0.5}{1}\right)^3}{\Gamma(3)} = \frac{1}{2}$$

Implying the steady state function:

$$f^\infty(x) = \begin{cases} 0 & if \quad x \leq 0 \\ \frac{1}{2x^4} exp\left(\frac{4}{x}\right) & if \quad x > 0 \end{cases}$$

Figure (4.29) displays the evolution of the Fokker-Planck neural network's solution for various propagation time steps. The convergence towards the specified target distribution becomes evident after a certain number of time steps.

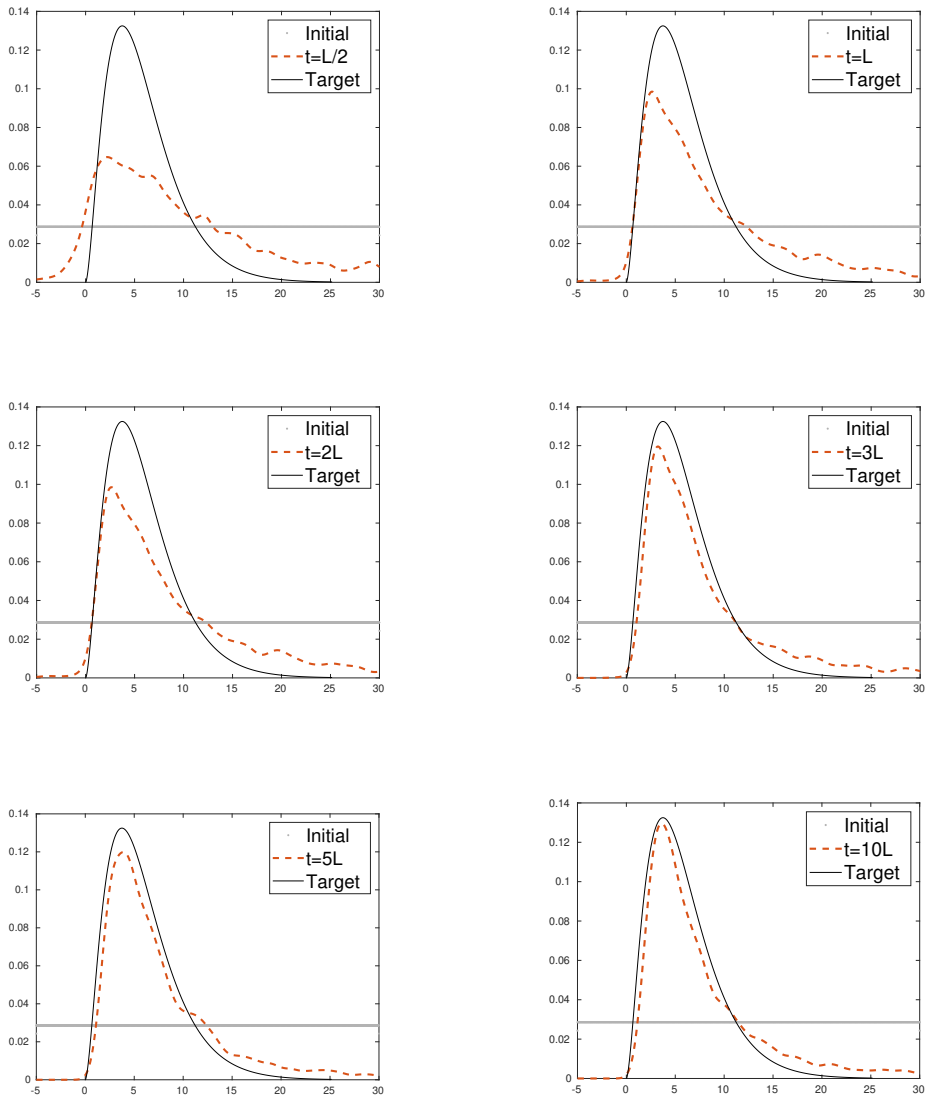


Figure 4.29: Solution of the Fokker-Planck neural network model at different propagation times.

Classification task

One dimensional case

In this final test, we are training a stochastic SimResNet that follows the dynamics described by the Euler-Maruyama interacting rule (3.3). The main objective is to classify a dataset with a specific initial distribution, aiming to achieve a final distribution that adheres to the characteristics outlined by the Fokker-Planck equation.
For these simulations, the following parameters have been set:

$$L = 20, \quad \epsilon = 0.05, \quad \gamma = 0.01, \quad N = 200$$

In this scenario, we have two separate sets of data, both initially distributed with values around -1 and 1. These two sets are assigned different labels, -2 and $+2$, respectively. The goal is to ensure that the SimResNet correctly classifies the data, causing them to converge towards their respective labels in the feature space. For instance, the particles initially distributed around $+1$ should ultimately converge near $+2$, and vice versa.

We can observe that this behavior is indeed evident during the layer propagation. Specifically, data points with a label of -2, as well as those with a label of $+2$, tend to converge towards gaussian distributions centered around their respective labels.
However, it's crucial to emphasize that, unlike the previous classification problems, the final distribution of data points in this stochastic setting can exhibit variability. This variability arises from the inherent randomness introduced into the dynamics. Consequently, even though the overall classification objective is achieved, the individual data points may display different final distributions due to the stochastic nature of the process.

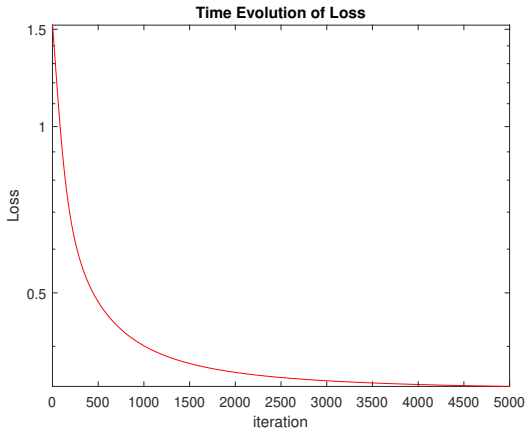


Figure 4.30: Loss minimization through the training process

We can also note the effective minimization of the loss function and the stabilization of the model's weights. This alignment between practical observations and theoretical expectations is indeed a positive outcome.

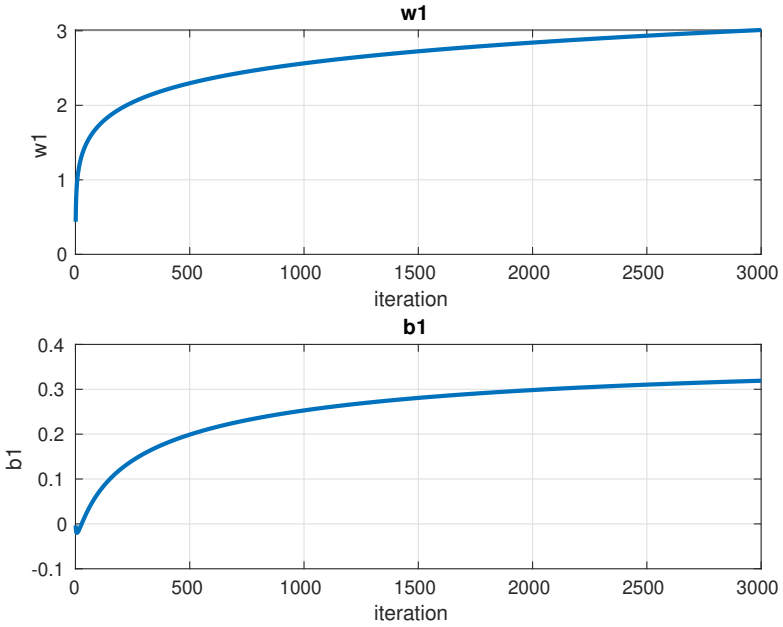


Figure 4.31: Weight and bias stabilization

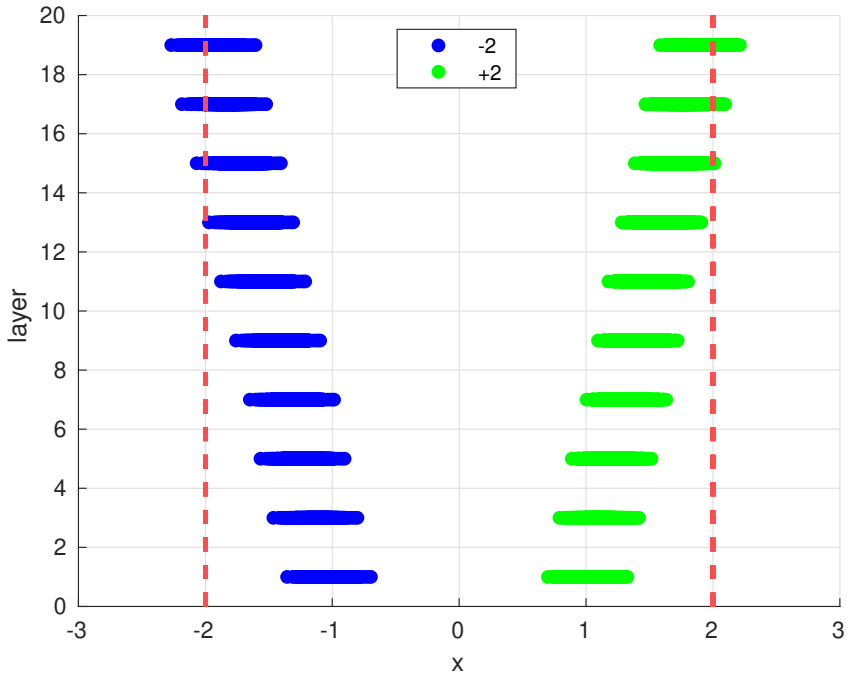


Figure 4.32: Data propagation through the one dimensional stochastic SimResNet

Two dimensional case

Next, our objective is to extend our findings to the two-dimensional scenario.
In particular, now we consider initial data clouds near the points $(-1,-1)$ and $(+1,+1)$ and the goal is to make them converge to the labels given, respectively, by $(-2,-2)$ and $(+2,+2)$.
In this case, we found it necessary to increase the number of layers L to 25 in order to achieve the desired level of loss function minimization.
Furthermore, it is noteworthy that the stochastic SimResNet continues to exhibit excellent performance in terms of data propagation. This results in the final distribution of data converging towards two gaussian distributions, each centered around their respective labels. This behavior underscores the effectiveness of our approach in handling two-dimensional data transformations.

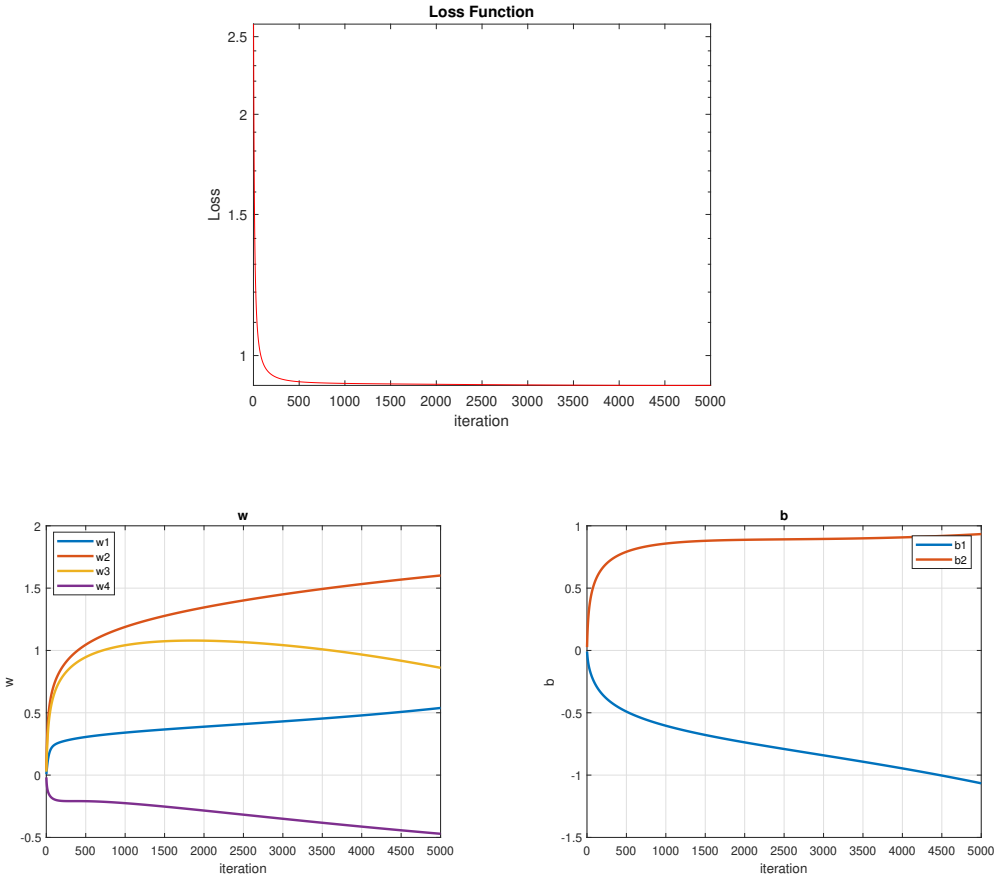
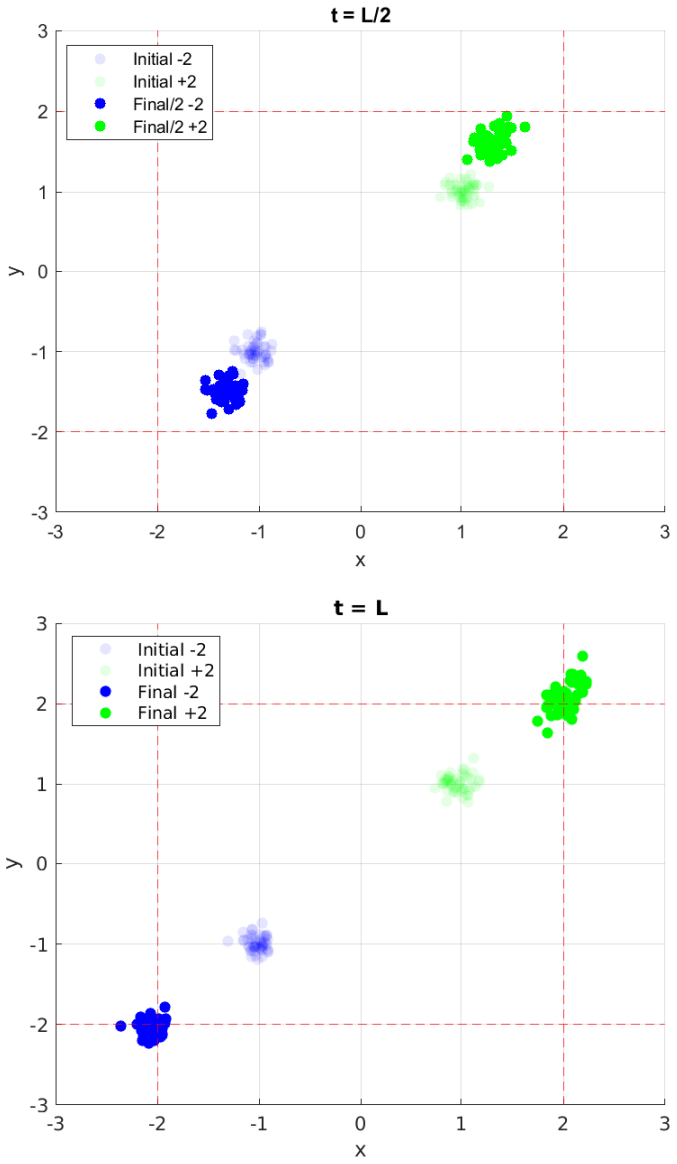


Figure 4.33: Loss minimization and stabilization of 2D stochastic SimResNet's weights



(a) Data propagation through the two dimensional stochastic Sim-ResNet

Conclusion

In conclusion, this master's thesis has delved into the intriguing intersection of partial differential equations and residual neural networks. The primary aim was to explore and demonstrate the remarkable potential of using SimResNets to approximate steady-state solutions of PDEs, while emphasizing the significance of particle-level data propagation within this framework.
An essential aspect of our approach is the emphasis on particle-level data propagation within SimResNets. This innovative technique not only enhances approximation accuracy but also provides a more profound understanding of system dynamics at a granular level.
Practically speaking, our research holds great promise for machine learning applications. We have demonstrated the adaptability of our methodology to regression and classification tasks across data of varying dimensions.
Furthermore, we have explored the distribution of solutions using Dirac delta, Gaussian or Inverse Gamma distributions, enabling precise modeling in real-world scenarios.
Beyond machine learning, the broader applicability of our findings is noteworthy: fields spanning biology, statistics, banking, and many others, where the dynamics of particle propagation are integral, can benefit from our research's insights.
In essence, our research underscores the potential for bridging mathematical modeling and advanced machine learning techniques to tackle complex problems. It offers a foundation for future explorations and applications across disciplines where the comprehension of particle interactions is vital for progress.
As we conclude, we envision our findings continuing to inspire innovative solutions to multifaceted challenges in science, engineering, and various other fields. The ability to track and interpret particle propagation within dynamic systems is a potent tool that promises to drive further innovation and discovery.

Appendix A

Hyperbolic Vlasov type equation

Since we are examining a model that combines neural networks with differential equations, it would be beneficial to provide an introduction to the types of equations we will be dealing with.

In this context, we will focus on (Hyperbolic) Vlasov-type partial differential equations (PDEs), which are relevant for describing the motion of particle densities. These equations provide a framework to understand the collective behavior and dynamics of large ensembles of particles.

Subsequently (not in this section) we will delve into the Boltzmann equation, which serves as a kinetic formulation of the problem. The Boltzmann equation enables us to study the evolution of particle distributions and interactions in a more detailed and microscopic manner.

By discussing Vlasov-type PDEs and the Boltzmann equation, we will gain a comprehensive understanding of the mathematical tools necessary for analyzing the dynamics of particle systems and their relationship to neural networks.

The hyperbolic Vlasov-type equation is a mathematical equation that describes the evolution of a probability function in a particle system. It is a variant of the Vlasov equation, which is commonly used to study the dynamics of a large number of interacting particles.

The usefulness of the hyperbolic Vlasov-type equation lies in its ability to capture the collective behavior of a large number of particles and provide a statistical description of their motion. It allows us to analyze the evolution of the probability distribution function $f(x, v, t)$, which gives the probability of finding a particle at a particular position x and velocity v at time t.

By solving the hyperbolic Vlasov-type equation, we can understand how the probability distribution function evolves over time in response to various forces and interactions among particles. This information is valuable for studying a wide range of phenomena such as the dynamics of charged particles in a specific fields and the statistical properties of collisional systems.

Furthermore, the hyperbolic Vlasov-type equation serves as a foundation for developing more sophisticated models and simulations of complex physical systems.

Let us give a formal description. Given $f(x, v, t)$ the probability distribu-

tion function described before, the hyperbolic Vlasov-type equation is typically written as follows:

$$\partial_t f + v \cdot \nabla_x f + F \cdot \nabla_v f = C[f] \tag{A.1}$$

where:

- $F(x,t)$ is the force acting on the particles, it can depend on position and time.
- $C[f]$ is the collision operator, which accounts for particle interactions and collisions. It describes how particles scatter and redistribute in phase space due to collisions.

The terms in the equation have the following interpretations:

- $\partial_t f$ represents the partial derivative of the distribution function with respect to time. It describes how the distribution function changes over time.
- $v \cdot \nabla_x f$, represents the transport of the distribution function by the particle velocity. It describes how the distribution function changes as particles move through phase space.
- $F \cdot \nabla_v f$, represents the influence of the force acting on the particles. It accounts for the acceleration or deceleration of particles due to external forces.

Solving the hyperbolic Vlasov-type equation involves finding a solution for the distribution function $f(x,v,t)$ that satisfies the equation and appropriate initial and boundary conditions. The specific form of the force term and collision operator will depend on the particular physical system under consideration, allowing us to apply mathematical analysis and numerical simulations to explore a wide range of physical phenomena.

case: in absence of collision

For our analysis can be usefull to understand the case of collisionless dynamics. In fact, we will see later that the internal interaction between particles will not be decisive for our dynamics.

Let's consider, as before, a system of particles with a probability distribution function $f(x,v,t)$ in phase space. In the absence of collisions, the evolution of the distribution function is governed by the equations of motion for individual particles. These equations can be described by the Liouville's equation:

$$\partial_t f + v \cdot \nabla_x f + F \cdot \nabla_v f = 0 \tag{A.2}$$

This equation represents the conservation of particles in phase space. As before, the first term on the left-hand side accounts for the change in the distribution function with time, the second term represents the advection of the distribution function by the particle velocity, and the third term represents the influence of the force acting on the particles.

case: in absence of velocity

Moreover, to simplify the equation we can consider the case in which the probability distribution function depend only on the position x and time t, i.e. $f(x,t)$. We start with the Liouville equation considering that there are no explicit collisions between particles and velocities are not effecting the system, we can obtain the following equation:

$$\partial_t f + \nabla_x f = 0 \tag{A.3}$$

This is the simplified form of the hyperbolic Vlasov equation in the case of collisionless dynamics without explicitly considering velocities. It represents the conservation of particles in phase space, where the distribution function $f(x,t)$ satisfies this equation for a constant velocity vector.

Note that in this simplified form, the velocity information is not explicitly included.
Instead, the equation focuses on the spatial evolution of the distribution function, assuming a constant velocity for all particles.

Appendix B

Analysis of the mean-field limit

Since the proof of the main result of the thesis is quite deep, some proof steps can be find in this Appendix. The bibliography items related to this part are (3) and (5).

Let's first explain the theorem that proves the existence and uniqueness of solution of (2.2).

Theorem 1 (Picard-Lindelőf)**.** *Let $D \subset \mathbb{R} \times \mathbb{R}^n$ be a closed rectangle with $(t,y_)\in intD$, the interior of D. Let $f : D \to \mathbb{R}^n$ be a function that is continuous in t and Lipschitz continuous in y. Then, there exists some $\epsilon > 0$ such that the initial value problem*

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0$$

has a unique solution $y(t)$ on the interval $[t_0 - \epsilon, t_0 + \epsilon]$.

Then we can move on the results related to *Section (2.2)*.
First of all the following definitions are useful in order to construct some minimal results.

Definition 7. *Let $\mathcal{P}(\mathbb{R}^d)$ the set of real-valued probability measures defined on $\mathbb{R}^d$. For $p > 0$ we denote by $\mathcal{P}_p(\mathbb{R}^d) \subset \mathcal{P}(\mathbb{R}^d)$ the set of probability with finite p-th moment, i.e.*

$$\mathcal{P}_p(\mathbb{R}^d) = \left\{ \mu \in \mathcal{P}(\mathbb{R}^d) : \int_{\mathbb{R}^d} |x|^p d\mu(x) < +\infty \right\}$$

Definition 8 (Push-forward)**.** *Given a map $\gamma : \mathbb{R}^d \to \mathbb{R}^d$, the push-forward of $\mu \in \mathcal{P}(\mathbb{R}^d)$*
through γ is defined for every Borel set $A \subset \mathbb{R}^d$ as the unique probability measure $\gamma\#\mu$ such that $\gamma\#\mu(A) := \mu(\gamma^{-1}(A))$

Definition 9. *We define the flow associated to the mean-field equation* (2.7) *as the map $\Phi_t : (x, \tau) \in \mathbb{R}^{d+1} \mapsto \Phi_t(x, \tau) \in \mathbb{R}^{d+1}$ such that*

$$\begin{cases} \partial_t \Phi_t(x,\tau) & = G(\Phi_t(x,\tau)) \\ \Phi_0(x,\tau) & = (x,\tau) \end{cases} \tag{B.1}$$

Next results are usefull to prove some result of the section.

Proposition 7 ((5), Proposition 7.2)**.** *Let v, w two vector fields, both Lipschitz with constant L and bounded. Let $\mu, v \in \mathcal{P}$ two probability measures. Then*

$$W_p(\Phi_t^v \# \mu, \Phi_t^w \# v) \leq e^{Lt} W_p(\mu, v) + \frac{e^{Lt} - 1}{L} ||v - w||_{C^0}$$

Lemma 1 ((5), Lemma 6.1)**.** *1. For a given $v \in C([0,T]; \mathcal{P}_1(\mathbb{R}^d))$ the flow map defined by* (B.1) *satisfies:*

$$|\Phi_t(x_2) - \Phi_t(x_1)| \leq K|x_2 - x_1|$$

for a suitable constant $K > 0$ independent of v, x_1, x_2

2. Let $v_1, v_2 \in C([0,T]; \mathcal{P}_1(\mathbb{R}^d))$ be given and let Φ^{v_1}, Φ^{v_2} be the corresponding flow maps generated by (B.1)*. Then there exists a constant $K > 0$, independent of both v_1 and v_2, such that*

$$|\Phi_t^{v_2}(x) - \Phi_t^{v_1}(x)| \leq K \int_0^t W_1(v_{1,s}, v_{2,s}) ds \qquad \forall x \in \mathbb{R}^d$$

The demonstration of Proposition 1 can be expanded upon below:

Proof of Proposition 1. Under the assumption *(A1)* and *(A2)* we have that G defined by equation (2.6) is also Lipschitz and uniformly bounded for all $(x, \tau) \in \mathbb{R}^{d+1}$, this means that G doesn't grow too quickly and doesn't have sharp changes in its behaviour, making it well-behaved from a mathematical perspective. Hence, due to *Appendix B* (1), the flow Φ_t introduced in the last definition is well-defined and Lipschitz in (x, τ) and $F_t = \Phi_t \# F_0$ for $F_0 \in \mathcal{P}_1(\mathbb{R}^{d+1})$ is the unique weak solution of equation (2.7) in the sense ofdefinition 2. This means that F_t describes how the probability distribution evolves over time according to the given equation.
Furthermore, under the assumptions *(A1)* and *(A2)*, any two weak solutions $F_t^{(1)}$ and $F_t^{(2)}$ in the sense of equation given in definition 2, obtained from initial conditions $F_0^{(1)}$,$F_0^{(2)}$, respectively, fulfill the Dobrushin's stability estimate in 1-Wasserstein distance. The Dobrushin's inequality allows us to prove the convergence of the solutions of the dynamical system (2.5)-(2.6) to F_t. In fact, we first observe that if we consider the initial condition

$$dF_0^N(x, \tau) = \frac{1}{N} \sum_{i=1}^N \delta(x - x_i^0) \delta(\tau) \tag{B.2}$$

with x_i^0 prescribed by (2.5), then the following empirical measure

$$dF_t^N(x, \tau) = \frac{1}{N} \sum_{i=1}^N \delta(x - x_i(t)) \delta(\tau - \tau_i(t)) \tag{B.3}$$

is a weak solution of (2.7) in sense of definition 2, where $(x_i(t), \tau_i(t))$ are the trajectories given by the dynamical system (2.5) for any $i = 1, \ldots, N$. The previous consideration follows from a classical derivation, see (4). Hence, if

the initial empirical measure converges in 1-Wasserstein distance W_1 to some $\bar{F}_0 \in \mathcal{P}_1(\mathbb{R}^{d+1})$ for $N \to \infty$, using the Dobrushin's estimate

$$W_1(\bar{F}_t, F_t^N) \leq C W_1(\bar{F}_0, F_0^N)$$

with C being a constant and $\bar{F}_t = \Phi_t \# \bar{F}_0$, we obtain that (2.7) is the mean-field limit of the particle dynamics (2.5) for $N \to \infty$. □

Appendix C

Moment analysis

Derivation of weak-form of Boltzmann-type equation

The aim is to derive the weak form of the Boltzmann-type equation (10). To do this rigorously, we will follow the standard procedure for deriving weak forms from partial differential equations.
Starting with the Boltzmann-type equation (3.1), we have:

$$\frac{d}{dt}f(t,x) = \frac{1}{\tau}(Q[f](t,x) - f(t,x))$$

Here, $Q[f](t,x)$ represents the *collision operator* acting on the distribution function $f(t,x)$. We want to derive the weak form of this equation. To do this, we will multiply both sides of the equation by a test function $\Phi(x)$ and integrate over the entire domain $\mathbb{R}$:

$$\int_{\mathbb{R}} \frac{d}{dt}f(t,x)\Phi(x)dx = \frac{1}{\tau}\int_{\mathbb{R}} (Q[f](t,x) - f(t,x))\Phi(x)dx$$

Now, let us consider the left-hand side. We have a time derivative of the integral

$$\frac{d}{dt}\int_{\mathbb{R}} f(t,x)\Phi(x)dx$$

By Leibniz's rule, we can exchange the order of differentiation and integration under certain conditions. Assuming that the conditions for this exchange hold, we can write:

$$\frac{d}{dt}\int_{\mathbb{R}} f(t,x)\Phi(x)dx = \int_{\mathbb{R}} \frac{\partial}{\partial t}(f(t,x)\Phi(x))dx$$

Now, we apply this to the left-hand side of the equation:

$$\begin{aligned}\int_{\mathbb{R}} \frac{\partial}{\partial t}(f(t,x)\Phi(x))dx &= \frac{1}{\tau}\int_{\mathbb{R}} (Q[f](t,x) - f(t,x))\Phi(x)dx \\ &= \frac{1}{\tau}\int_{\mathbb{R}} Q[f](t,x)\Phi(x)dx - \frac{1}{\tau}\int_{\mathbb{R}} f(t,x)\Phi(x)dx\end{aligned}$$

At this point, we have expressed the time derivative of the integral of

$f(t,x)\Phi(x)$ on the right-hand side and the left-hand one as the integrals involving the collision operator and the test function.

Now we can subtract the term involving the integral of $f(t,x)\Phi(x)$ from both sides pf the equation to isolate the collision operator term on the left side:

$$\frac{1}{\tau}\int_{\mathbb{R}} Q[f](t,x)\Phi(x)dx = \int_{\mathbb{R}} \frac{\partial}{\partial t}(f(t,x)\Phi(x))dx + \frac{1}{\tau}\int_{\mathbb{R}} f(t,x)\Phi(x)dx$$

Finally, we arrive at the weak form of the Boltzmann-type equation:

$$\frac{d}{dt}\int_{\mathbb{R}} \Phi(x)f(t,x)dx = \frac{1}{\tau}\int_{\mathbb{R}} \Big(\Phi(x^*) - \Phi(x)\Big)f(t,x)dx$$

This weak form is obtained by integrating the original equation against a test function $\Phi(x)$, and it is a standard approach in the context of distribution functions and particle interactions.

Appendix D

Fokker-Planck equation

The Fokker-Planck equation, also known as the Fokker-Planck-Kolmogorov equation, is a fundamental equation in the study of stochastic processes. It describes how the probability density function of a stochastic system evolves over time. This equation finds applications in a wide range of fields, including physics, chemistry, economics, and biology.

Structure of the Fokker-Planck Equation

The Fokker-Planck equation has the following general form:

$$\frac{\partial f(t,x)}{\partial t} = -\nabla \cdot [\mathbf{A}(\mathbf{x},t) f(t,x)] + \frac{1}{2}\nabla \cdot [\mathbf{B}(\mathbf{x},t) \cdot \nabla f(t,x)] \tag{D.1}$$

Where:

- $f(t,x)$: probability density function of the stochastic process
- $\mathbf{x}$: vector of state variables
- $\mathbf{A}(\mathbf{x}, t)$ is the **drift vector field**: it describes the deterministic part of the motion of a stochastic process. In simpler terms, it represents how the system tends to move or evolve on average.[1]
- $\mathbf{B}(\mathrm{x}, t)$ is the **diffusion matrix**: it represents the stochastic or random component (random fluctuations, noise, or diffusion) that affect the system's behavior over time.

The equation consists of two main terms. The first term describes the advection of probability density due to the drift vector field $\mathbf{A}(\mathbf{x},t)$. The second term represents the diffusion of probability density due to the diffusion matrix $\mathbf{B}(\mathbf{x},t)$.

[1] As example, in a financial context, $\mathbf{A}(\mathbf{x},t)$ could represent the expected return rate of a stock portfolio. If this return rate is positive, it implies that, on average, the portfolio is expected to grow, leading to a drift in the PDF towards higher values.

Applications

The Fokker-Planck equation has various applications in different fields:

- *Physics*: it is used to model the behavior of particles undergoing Brownian motion, the dynamics of fluids in turbulence, and the evolution of quantum systems.
- *Chemistry*: in chemical kinetics, it describes the evolution of reactant concentrations in systems with stochastic reactions.
- *Economics*: it can be applied to model stock prices and financial markets, where randomness plays a significant role.
- *Biology*: in population biology, it models the distribution of species populations under random environmental fluctuations.
- *Engineering*: it finds applications in the analysis of noise in electronic circuits and in control systems subject to stochastic disturbances.
- *Finance*: it is used in option pricing models, such as the Black-Scholes equation, which incorporates stochastic volatility.
- *Neuroscience*: it helps analyze the dynamics of neural networks and the propagation of synaptic noise in neurons.

In conclusion, the Fokker-Planck equation is a versatile tool for understanding and predicting the behavior of stochastic systems across various disciplines. Its structure captures both deterministic and stochastic aspects, making it an essential equation in the study of random processes.

Appendix E

Matlab codes

Activation function:

```
% Hyperbolic tangent activation
function output = tanh_activation(x,W,b)
    output = tanh(W.*x+b);
end

% Derivative of the Hyperbolic tangent activation
function output = tanh_derivative(x,W,b)
    output = 1 - tanh(W.*x+b).^2;
end
```

Classification task 1D

```
% Data Generation

% Define the intervals and bin counts
intervals = [2,2.6;2.6,3.2;3.2,3.8;3.8,4.4;4.4,5;
             5,5.6;5.6,6.2;6.2,6.8;6.8,7.4;7.4,8];
bin_counts = [5,5,4,6,3,8,3,6,6,4];

% Generate data based on the intervals and bin counts
data = [];
for i = 1:length(bin_counts)
    interval = intervals(i, :);
    bin_count = bin_counts(i);

    % Generate random values within the interval
    values = interval(1) + (interval(2) - interval(1)) *
                                        rand(bin_count, 1);
    data = [data; values];
end

training_data = data;

data_target = [2.*ones(23,1); 8.*ones(27,1)];


% TRAINING SIMRESNET

% Define the neural network parameters
T = 220;  % Number of layers
```

```
h = 0.05; % Step size

% Initialize the weight and bias
w = randn;  % Random initialization
b = randn;  % Random initialization

% Perform stochastic gradient descent
N = length(training_data);
iteration = 12000;
learning_rate = 0.01;

% Initialize loss array for plotting
loss_array = zeros(iteration, 1);

% Initialize arrays to store the evolution of w and b
w_evolution = zeros(iteration, 1);
b_evolution = zeros(iteration, 1);

for iter = 1:iteration

        loss_iter = 0;  % Accumulate loss for the epoch

        for i = 1:N   %ad ogni epoch scorro ogni train data
            % Select the current training sample
            x_p = zeros(1,T);
            x_j = training_data(i);
            y_j = data_target(i);
            x_p(1) = x_j;

            for t = 1:T-1
                x_p(t+1) = x_p(t) + h * tanh_activation(x_p(t),w,b);
            end

            x = max(2, min(8, x));

            % Calculate the loss for each train data
            loss = 0.5 * norm(x_p(end) - y_j, 2)^2;

            % Accumulate loss for the epoch
            loss_iter = loss_iter + loss;

            % Backward propagation
            lambda = zeros(1,T);
            lambda(end) = x_p(end) - y_j;   %optimality condition
            grad_w = 0;
            grad_b = 0;

            for t = T:-1:2
                grad_w = grad_w + lambda(t) * h *
                        tanh_derivative(x_p(t-1),w,b) * x_p(t-1);
                grad_b = grad_b + lambda(t) * h *
                                    tanh_derivative(x_p(t-1),w,b);

                lambda(t-1) = lambda(t) * tanh_derivative(x_p(t),w,b) * w;
            end


            % Update the weights and biases
            w = w - learning_rate * grad_w * h / T;
            b = b - learning_rate * grad_b * h / T;
```

```
        end

        % Average loss for the epoch
        loss_epoch = h * loss_iter / N;
        loss_array(iter) = loss_iter;

        % Store the current values of w and b
        w_evolution(iter) = w;
        b_evolution(iter) = b;
        disp(iter)
end

% Plot the log-log plot of loss
figure;
semilogy(loss_array,'r');
title('Time Evolution of Loss');
xlabel('iteration');
ylabel('Loss');

% Plot the time evolution of w and b
figure;
subplot(2, 1, 1);
plot(w_evolution,'LineWidth', 2);
title('Time Evolution of Weight');
%txt = ['weight: ' num2str(w)];
%text(4,3,txt)
xlabel('iteration');
ylabel('w');
grid on

subplot(2, 1, 2);
plot(b_evolution,'LineWidth', 2);
title('Time Evolution of Bias');
%txt = ['bias: ' num2str(b)];
%text(4,0.5,txt)
xlabel('iteration');
ylabel('b');
grid on

%final data activation
% Plot the trajectories of neuron activation energies
act_trajectories(:, 1) = training_data;
for k = 1:N
    for t = 1:T-1
        act_trajectories(k,t+1) = act_trajectories(k,t)
             + h * tanh_activation(act_trajectories(k,t),w,b);
        %act_trajectories(k,t+1) = max(2,
                              min(8, act_trajectories(k,t+1)));
    end
end

figure;
hold on;
for i = 1:N
    plot(1:T, act_trajectories(i, :),'LineWidth', 2);
end
hold on;
% Insert horizontal line
plot(5*ones(1,240), '--', 'Color', 'black'); hold off;
title('Trajectories of Neuron Activation Energies');
```

```
axis([0 240 -inf inf])
xlabel('Time');
ylabel('Neuron Activation Energy');
```

Regression task 2D

```
    N_x = 1000;
training_data = zeros(2,N_x*5);
training_data(1,:)=[ones(1,N_x),2.*ones(1,N_x),3.*ones(1,N_x),4.*ones(1,N_x),
                                          5.*ones(1,N_x)];

sigma = 1;

for g = 1:5
    A = [];
    for pos = 1:N_x
        A(pos) = normrnd(training_data(1,pos+N_x*(g-1)), sigma);
    end
    training_data(2,1+N_x*(g-1):N_x*g) = A;
end

x=[1,2,3,4,5];
y=x;
figure
scatter(training_data(1,:),training_data(2,:)); hold on
plot(x,y)
hold off
xlabel('x')
ylabel('y')
grid on
%title('Training data distribution')


%the idea is to generate the couples (m,q)
%picking randomly two points with consecutive x. Calculate
%m with formula (y2-y1)/(x2-x1) and intercept as
%q = y-mx
par = zeros(2,N_x*4);
m_n = zeros(2,N_x*4);
q_n = zeros(2,N_x*4);
for g = 1:4
    index_sp = linspace(1+N_x*g, N_x+g*N_x, N_x);
    for h = 1:N_x
        m_n(1,(g-1)*N_x+h) = (training_data(1,index_sp(h))
                            + training_data(1,(g-1)*N_x+h)).*0.5;
        m_n(2,(g-1)*N_x+h) = (training_data(2,index_sp(h))
     - training_data(2,(g-1)*N_x+h))./(training_data(1,index_sp(h))
                               - training_data(1,(g-1)*N_x+h));
        q_n(1,(g-1)*N_x+h) = (training_data(1,index_sp(h))
                        + training_data(1,(g-1)*N_x+h)).*0.5;
        q_n(2,(g-1)*N_x+h) = training_data(2,(g-1)*N_x+h)
               - m_n(2,(g-1)*N_x+h).*training_data(1,(g-1)*N_x+h);
    end
end
slopes = m_n(2,:);
slopes_perm = slopes(randperm(length(slopes)));
intercepts = q_n(2,:);
intercepts_perm = intercepts(randperm(length(intercepts)));


% TRAINING SIMRESNET

% Define the neural network parameters
```

```
T = 400;  % Number of layers
h = 1/T; % Step size

nstar = 4000;      % number of data considered for the SGD

% Initialize the weight and bias
w = randn(4,1);
b = randn(2,1);

iteration = 2000;
learning_rate = 0.01;

% Initialize arrays to store the evolution of w and b
w_evolution = zeros(4, iteration);
b_evolution = zeros(2, iteration);

rand_ind = randi([1, 5*N_x],1,nstar);
SGD_set = training_data(:, rand_ind);

n_train = 4000;

m_chosen = par(1, randi([1,size(par,2)],1,n_train));
q_chosen = par(2, randi([1,size(par,2)],1,n_train));

loss_epoch = zeros(1,iteration);

for iter = 1:iteration
    disp(iter)

    for f=1:n_train
        m = m_chosen(f);
        q = q_chosen(f);

        for t = 1:T     %forward propagation of the couple (m,q)
            m_ = m;
            m = m + h.*tanh_activation(m, q, w(1), w(3), b(1));
            q = q + h.*tanh_activation(m, q, w(2), w(4), b(2));
        end

        loss_SGD = 0;
        lambda = 0;
        mu = 0;

        for j = 1:nstar
            loss_SGD = loss_SGD + 0.5 * norm(q + m * SGD_set(1, j)
                                        - SGD_set(2, j), 2)^2;
            lambda = lambda + (q + m * SGD_set(1, j) - SGD_set(2, j))
                                            * (SGD_set(1, j));
            mu = mu + (q + m * SGD_set(1, j) - SGD_set(2, j));
        end

        loss_SGD = loss_SGD./nstar;

        loss_epoch(iter) = loss_SGD;

        lambda = lambda / nstar;
        mu = mu / nstar;

     % Backpropagation
        grad_w = zeros(4,1);
```

```
        grad_b = zeros(2,1);

        for t = T:-1:1
            grad_w(1) = grad_w(1) + lambda * h
                        * tanh_derivative(m, q, w(1), w(3), b(1)) * m;
            grad_w(2) = grad_w(2) + mu * h
                        * tanh_derivative(m, q, w(2), w(4), b(2)) * m;
            grad_w(3) = grad_w(3) + lambda * h
                        * tanh_derivative(m, q, w(1), w(3), b(1)) * q;
            grad_w(4) = grad_w(4) + mu * h
                        * tanh_derivative(m, q, w(2), w(4), b(2)) * q;

            grad_b(1) = grad_b(1) + lambda * h
                        * tanh_derivative(m, q, w(1), w(3), b(1));
            grad_b(2) = grad_b(2) + mu * h
                        * tanh_derivative(m, q, w(2), w(4), b(2));

            lambda_ = lambda;
            lambda = lambda + lambda.*w(1)* h
                   * tanh_derivative(m, q, w(1), w(3), b(1))+mu.*w(2)
                        *h *tanh_derivative(m, q, w(2), w(4), b(2));
            mu = mu + mu.*w(4)* h
                   * tanh_derivative(m, q, w(2), w(4), b(2))+lambda.
                        *w(3)*h *tanh_derivative(m, q, w(1), w(3), b(1));

        end

        % Update the weights and biases
        w = w - learning_rate * grad_w;
        b = b - learning_rate * grad_b;

        % Store the current values of w and b
        w_evolution(:, iter) = w;
        b_evolution(:, iter) = b;
    end
end
```

Multivariate regression

```
% DATA CREATION [x,y,z]

n = 3; % Dimension
A_target = randi([-10, 10], 1, n+1); %Target hyperplane

% Define the number of points
num_points = 100;

% Generate random points on the hyperplane
test_points = zeros(num_points, n+1);

for i = 1:num_points
    % Generate n random values in the range [-10, 10]
    random_values = (rand(1, n-1) - 0.5) * 4 * 4;

    last_coordinate = -(A_target(1:n-1) * random_values') /
                                    A_target(n) - 2*randn();

    % Combine all coordinates to form a point on the hyperplane
    test_point = [random_values, last_coordinate, -1];

    % Store the point in the list
    test_points(i, :) = test_point;
end

% Plot the points on the hyperplane
figure;
scatter3(test_points(:,1), test_points(:,2), test_points(:,3),
                                                    'filled');

xlabel('x');    ylabel('y');    zlabel('z');    grid on;

[x, y] = meshgrid(-10:1:10, -10:1:10);
z = -(A_target(1) * x + A_target(2) * y + A_target(4)) / A_target(3);
hold on;
surf(x, y, z, 'FaceAlpha', 0.3);
legend('Points on Hyperplane', 'Target Hyperplane', 'Location', 'Best');
hold off;



% HYPERPLANE TRAIN SET GENERATION

n_hyperplanes_train = 150;
n_hyperplanes_test = 5;
total_hyperplane = n_hyperplanes_train+n_hyperplanes_test;

A_dataset = zeros(total_hyperplane,n+1);

for i = 1:total_hyperplane
% Select three random points from the generated set
selected_indices = randperm(num_points, 3);
selected_points = test_points(selected_indices, :)+5*rand(3,n+1);

% Calculate the normal vector of the plane containing the three points
p1 = selected_points(2, 1:3) - selected_points(1, 1:3);
p2 = selected_points(3, 1:3) - selected_points(1, 1:3);
```

```
normal_vector = cross(p1, p2);

% Calculate the hyperplane equation coefficients
A_coeff = [normal_vector, -dot(normal_vector, selected_points(1, 1:3))];

% Normalize the coefficients
A_coeff = A_coeff / norm(A_coeff(1:3));
A_dataset(i,:) = A_coeff;
end

% Create a figure for the plot
figure;

% Plot each hyperplane
for i = 1:n_hyperplanes_train
    A = A_dataset(i, :); % Get the coefficients for the current hyperplane

    % Define the range for the plot
    z = -(A(1) * x + A(2) * y + A(4)) / A(3);

    % Plot the hyperplane
    surf(x, y, z, 'FaceAlpha', 0.3, 'FaceColor', '[0.1,0.1,0.1]');
    hold on;
end

%title('Hyperplane Train Set');
xlabel('x');
ylabel('y');
zlabel('z');
grid on; hold on;

% Define the range for the plot
z = -(A_target(1) * x + A_target(2) * y + A_target(4)) / A_target(3);

% Plot the hyperplane
surf(x, y, z, 'FaceAlpha',1, 'FaceColor', 'red');hold on;
scatter3(test_points(:,1), test_points(:,2), test_points(:,3), 'filled');
hold on;


%% TRAINING SIMRESNET

L=20;
learning_rate = 0.01;
iterations= 500;
h=1/L;

% Initialize the weight and bias
w = randn(n+1,n+1);  % Random initialization
b = randn(n+1,1);  % Random initialization

% Initialize arrays to store the evolution of w and b
w_evolution = zeros(n+1,n+1, iterations);
b_evolution = zeros(n+1, iterations);

w_evolution(:,:,1) = w./norm(w);
b_evolution(:,1) = b./norm(b);

loss_epoch = zeros(1,iterations);
```

```
for iter = 2:iterations
    disp(iter)
    loss_iter = 0;

    for f = 1:n_hyperplanes_train
        A = A_dataset(f,:);

        for t = 1:L         %forward propagation
            for col = 1:n+1
                A = A + h * (tanh(A * w(:, col) + b(col)) - A);
            end

        end

        total_loss = 0;
        lambda = zeros(n+1,1);

        for data_test = 1:num_points
            total_loss = total_loss + 0.5*norm(A *
                            test_points(data_test,:)').^2;
            lambda = lambda + (test_points(data_test,:)*
                            test_points(data_test,:)').*A';
        end

        total_loss = total_loss/num_points;
        lambda = lambda/num_points;

        % Backward
        grad_w = zeros(n+1,n+1);
        grad_b = zeros(n+1,1);

        for t = L:-1:1

            col = zeros(n+1,1);
            for j = 1:n+1
                col(j) = A*w(:,j)+b(j);
            end

            der_col = 1-tanh(col).^2;

            lambda_der = lambda.*der_col;

            lambda_add = w * lambda_der;

            grad_w = grad_w + A'*(lambda_der');

            grad_b = grad_b + lambda_der;

            lambda = lambda + lambda_add;

        end

        % Update the weights and bias
        w = w - learning_rate * grad_w;
        b = b - learning_rate * grad_b;
        % for row = 1:n+1
        %     w(row, :)=w(row, :)./norm(w(row, :));
        % end
        w = w./norm(w);
        b = b./norm(b);
```

```
        % Store the current values of w and b
        w_evolution(:,:, iter) = w;
        b_evolution(:, iter) = b;

        loss_iter = loss_iter + total_loss;
    end

    loss_iter = loss_iter/n_hyperplanes_train;

    % Update the loss for this iteration
    loss_epoch(iter) = loss_iter;
end

%LOSS PLOT

figure;
plot(loss_epoch,'r');
xlabel('Iteration');
ylabel('Loss');
title('Loss Function');


% SINGLE PROPAGATION

t = L;

% Plot each hyperplane
figure
for i = 1:total_hyperplane
    A = A_dataset_propagation(i, :,t);
    % Define the range for the plot
    z = -(A(1) * x + A(2) * y + A(4)) / A(3);
    % Plot the hyperplane
    surf(x, y, z, 'FaceAlpha', 0.3, 'FaceColor', '[0.7,0.7,0.7]');
    hold on;
end

xlabel('x');   ylabel('y');   zlabel('z');
grid on; hold on;

z = -(A_target(1) * x + A_target(2) * y + A_target(4)) / A_target(3);

% Plot the hyperplane
surf(x, y, z, 'FaceAlpha',1, 'FaceColor', 'red');hold on;
scatter3(test_points(:,1), test_points(:,2), test_points(:,3), 'filled');
hold on;
%title('t=L');
xlabel('x');
ylabel('y');
zlabel('z');

figure
subplot(2,2,1)
histogram(A_dataset_propagation(:, 1, t), 'BinWidth', 0.5, 'EdgeColor',
'black', 'FaceColor', 'blue')
title('a1')
xlabel('Value')
ylabel('Frequency')
xlim([-10, 10])
```

```
xline(A_target(1), '--r'); % Add a vertical line at A_target
grid on;

subplot(2,2,2)
histogram(A_dataset_propagation(:, 2, t), 'BinWidth', 0.5, 'EdgeColor',
'black', 'FaceColor', 'blue')
title('a2')
xlabel('Value')
ylabel('Frequency')
xlim([-10, 10])
xline(A_target(2), '--r'); % Add a vertical line at A_target
grid on;

subplot(2,2,3)
histogram(A_dataset_propagation(:, 3, t), 'BinWidth', 0.5, 'EdgeColor',
'black', 'FaceColor', 'blue')
title('a3')
xlabel('Value')
ylabel('Frequency')
xlim([-10, 10])
xline(A_target(3), '--r'); % Add a vertical line at A_target
grid on;

subplot(2,2,4)
histogram(A_dataset_propagation(:, 4, t), 'BinWidth', 0.5, 'EdgeColor',
'black', 'FaceColor', 'blue')
title('a4')
xlabel('Value')
ylabel('Frequency')
xlim([-10, 10])
xline(A_target(4), '--r'); % Add a vertical line at A_target
grid;
```

Fokker-Planck

```
% Propagation parameters
eps = 1e-2;
w = -1;
b = 0;
L = 100;
K = 1;
num_simulations = 50;

% Single propagation
prop_time = L;
data_trajectories=[];
data_trajectories(:, 1) = data;

for k = 1:num_points
    for t = 1:prop_time
        data_trajectories(k,t+1) = data_trajectories(k,t)
        + eps * identity_act(data_trajectories(k,t),w,b)
        + sqrt(eps)*K*randn();
    end
end

% Plots
kernel_width_f = 0.50; % Adjust the bandwidth for smoothing
pdf_estimate_prop = ksdensity(data_trajectories(:,end),
                     x_range, 'Bandwidth', kernel_width_f );

% Plot the density function
figure;
plot(x_range, pdf_estimate,'.', 'Color', 'black', 'LineWidth',
                     0.001, 'Color','[0.7 0.7 0.7]'); hold on;
plot(x_range, pdf_estimate_prop,'-.', 'LineWidth', 2, 'Color',
                                                  'g');hold on;
plot(x, y, 'Color', 'black', 'LineWidth', 0.5); hold off;
```

Fokker-Planck - Classification

One dimensional case

```
% DATA CONSTRUCTION 1D

num_points = 200;
points = zeros(num_points,2);
points(1:num_points/2,1) = -1 + 0.1*randn(num_points/2,1);
points(num_points/2+1:num_points,1)=1 + 0.1*randn(num_points/2,1);
points(1:num_points/2,2) = -2;
points(num_points/2+1:num_points,2) = 2;

% Scatter plot to visualize the points with labels
figure;
scatter(points(1:num_points/2,1),zeros(num_points/2,1),
                                        'b', 'filled');
hold on;
scatter(points(num_points/2+1:num_points,1),
                zeros(num_points/2,1), 'g', 'filled');
```

```
xlabel('x');
title('Data points');
legend('-2','+2')
grid on
axis([-2.5 2.5 -0.5 5])


% TRAINING STOCHASTIC 1D SIMRESNET

% Define the neural network parameters
L = 20;  % Number of layers

% Initialize the weight and bias
w = 0;  % Random initialization
b = 0;  % Random initialization

% Perform stochastic gradient descent
iteration = 3000;
learning_rate = 0.01;

% Initialize arrays to store the evolution of w and b
w_evolution = zeros(1, iteration);
b_evolution = zeros(1, iteration);

eta = randn(1,L);
K=1;
eps = 1/L;

% Initialize loss array for plotting
loss_array = zeros(iteration, 1);

for iter = 1:iteration
    disp(iter)

    loss_iter = 0;  % Accumulate loss for the epoch

    for i = 1:num_points
        x_p = zeros(1,L);
        x_p(1) = points(i,1);
        label = points(i,2);

        for t = 1:L-1
            x_p(t+1) = x_p(t) + eps*tanh_activation(x_p(t),w,b)
                                        + sqrt(eps)*K*eta(t);
        end

        % Calculate the loss for each train data
        loss = 0.5 * norm(x_p(end) - label, 2)^2;

        % Accumulate loss for the epoch
        loss_iter = loss_iter + loss;

        % Backward propagation
        lambda = zeros(1,L);
        lambda(end) = x_p(end) - label;

        grad_w = 0;
        grad_b = 0;

        for t = L:-1:2
```

```
            lambda(t-1)=lambda(t)+eps*
                           tanh_derivative(x_p(t),w,b) * w;

            grad_w=grad_w+lambda(t)*eps*
                          tanh_derivative(x_p(t-1),w,b)*x_p(t-1);

            grad_b=grad_b+lambda(t)*eps*
                          tanh_derivative(x_p(t-1),w,b);

        end

        % Update the weights and biases
        w = w - learning_rate * grad_w/L;
        b = b - learning_rate * grad_b/L;
    end

    % Average loss for the epoch
    loss_array(iter) = loss_iter/num_points;

    % Store the current values of w and b
    w_evolution(:,iter) = w;
    b_evolution(:,iter) = b;
end
```

Two dimensional case

```
% DATA CONSTRUCTION 2D

num_points = 200;
points = zeros(num_points,3);
points(1:num_points/2,1:2)=-1+0.1*randn(num_points/2,2);
points(num_points/2+1:num_points,1:2)=1
                            + 0.1*randn(num_points/2,2);
points(1:num_points/2,3) = -2;
points(num_points/2+1:num_points,3) = 2;

% Scatter plot to visualize the points with labels
figure;
scatter(points(1:num_points/2,1),
                points(1:num_points/2,2),'b', 'filled');
hold on;
scatter(points(num_points/2+1:num_points,1),
     points(num_points/2+1:num_points,2), 'g', 'filled');
xlabel('x');
ylabel('y');
title('Data points');
legend('-2','+2')
grid on
axis([-2.5 2.5 -2.5 2.5])


% TRAINING STOCHASTIC 2D SIMRESNET

% Define the neural network parameters
L = 50;  % Number of layers

% Initialize the weight and bias
w = zeros(4,1);  % Random initialization
```

```
b = zeros(1,2);  % Random initialization

% Perform stochastic gradient descent
iteration = 5000;
learning_rate = 0.001;

% Initialize arrays to store the evolution of w and b
w_evolution = zeros(4, iteration);
b_evolution = zeros(2, iteration);

%we make the randomness to be deterministic
eta = randn(2,L);
K=1;
eps = 1/L;

loss_array = zeros(iteration, 1);

for iter = 1:iteration
    disp(iter)

    loss_iter = 0; % Accumulate loss for the epoch

    for i = 1:num_points
        x_p = zeros(2,L);
        x_p(:,1) = points(i,1:2);
        label = [points(i,3);points(i,3)];

        for t = 1:L-1
            x__p1 = x_p(1,t);
            x_p(1,t+1) = x_p(1,t) + eps*
            tanh_activation(x_p(1,t), x_p(2,t), w(1), w(3), b(1))
                                        + sqrt(eps)*K*eta(1,t);
            x_p(2,t+1) = x_p(2,t) + eps*
            tanh_activation(x__p1, x_p(2,t), w(2), w(4), b(2))
                                        + sqrt(eps)*K*eta(2,t);
        end

        % Calculate the loss for each train data
        loss = 0.5 * norm(x_p(:,end) - label, 2)^2;

        % Accumulate loss for the epoch
        loss_iter = loss_iter + loss;

        % Backward propagation
        lambda = zeros(2,L);
        lambda(:,end) = x_p(:,end) - label;

        grad_w = zeros(4,1);
        grad_b = zeros(2,1);

        for t = L:-1:2

            lambda(1,t-1)=lambda(1,t)+w(1).*eps.*
            tanh_derivative(x_p(1,t), x_p(2,t), w(1), w(3), b(1))+w(2).
            *eps.*tanh_derivative(x_p(1,t), x_p(2,t), w(2), w(4), b(2));
            lambda(2,t-1)=lambda(2,t)+w(3).*eps.*
            tanh_derivative(x_p(1,t), x_p(2,t), w(2), w(4), b(2))+w(4).
            *eps.*tanh_derivative(x_p(1,t), x_p(2,t), w(1), w(3), b(1));

            grad_w(1) = grad_w(1) + lambda(1,t) .* eps .*
```

```
            tanh_derivative(x_p(1,t),x_p(2,t),w(1),w(3),b(1))*x_p(1,t);
            grad_w(2) = grad_w(2) + lambda(2,t) .* eps .*
            tanh_derivative(x_p(1,t),x_p(2,t),w(2),w(4),b(2))*x_p(1,t);
            grad_w(3) = grad_w(3) + lambda(1,t) .* eps .*
            tanh_derivative(x_p(1,t),x_p(2,t),w(1),w(3),b(1))*x_p(2,t);
            grad_w(4) = grad_w(4) + lambda(2,t) .* eps .*
            tanh_derivative(x_p(1,t),x_p(2,t),w(2),w(4),b(2))*x_p(2,t);

            grad_b(1) = grad_b(1) + lambda(1,t) * eps .*
            tanh_derivative(x_p(1,t), x_p(2,t), w(1), w(3), b(1));
            grad_b(2) = grad_b(2) + lambda(2,t) * eps .*
            tanh_derivative(x_p(1,t), x_p(2,t), w(2), w(4), b(2));

        end

        % Update the weights and biases
        w = w - learning_rate * grad_w/L;
        b = b - learning_rate * grad_b/L;
    end

    % Average loss for the epoch
    loss_array(iter) = loss_iter/num_points;

    % Store the current values of w and b
    w_evolution(:,iter) = w;
    b_evolution(:,iter) = b(:,1);
end
```

List of Figures

List of Algorithms

Bibliography

[1] Michael Herty, Torsten Trimborn. *Mean-field and kinetic descriptions of neural differential equations*. Foundation of Data Science Vol. 4 No. 2, June 2022.

[2] Patrick Kidger. *On Neural Differential Equations*. Thesis for the degree of Doctor of Philosophy, 4 February 2022.

[3] Francois Golse. *On the Dynamics of Large Particle Systems in the Mean Field Limit* . Centre de Mathematiques Laurent Schwartz, 23 January 2013.

[4] Francois Golse. *On the dynamics of large particle systems in the Mean Field Limit.* In *Macroscopic and large scale phenomena: coarse graining, mean field limits and ergodicity*, pages 1-144. Springer, 2016.

[5] Michael Herty, Anna Thunen, Torsten Trimborn, Giuseppe Visconti. *Continuous limits of residual neural networks in case of large input data*. 11 May 2022.

[6] Paulo Tabuada, Bahman Gharesifard. *Universal Approximation Power of Deep Residual Neural Networks via nonlinear control theory*. 16 Dec 2020.

[7] Patrick Kidger, Terry Lyons. *Universal approximation with deep narrow networks*, In *Conference on Learning Theory*, 2020.

[8] Hongzhou Lin, Stefanie Jegelka. *Resent with one-neuron hidden layers is a universal approximator*, NIPS'18, Red Hook, NY, USA, Curran Associates Inc, pages 6172-6181. 2018.

[9] Yulong Lu, Jianfeng Lu. *A universal approximation theorem of deep neural networks for expressing probability distribution*Advances in Neural Information Processing Systems, Curran Associates, Inc., 33. pages 3094-3105 2020.

[10] L. Pareschi, G.Toscani. *Interacting Multiagent Systems. Kinetic equations and Monte Carlo methods*, Oxford University Press, 2013.

[11] B. Bonnet, C. Cipriani, M. Fornasier, H. Huang. *A Measure Theoretical Approach to the Mean-field Maximum Principle for Training NeurODEs*, 2022.

Ringraziamenti

Vorrei ringraziare tutte le persone che in questi cinque anni mi hanno accompagnato in questo bellissimo percorso di laurea e di vita. Prima di tutti, vorrei dedicare un pensiero al mio relatore, Giacomo Albi, per il supporto dato durante la stesura della tesi. Sono veramente soddisfatto del percorso accademico grazie ai corsi svolti con lui e agli aiuti ricevuti.

Vorrei ringraziare la mia famiglia, in particolare mia nonna, mia zia e la donna a cui la tesi è dedicata. Li ringrazio per la vicinanza, l'amore e il sostegno emotivo che mi hanno dato anche senza accorgersene.

Ringrazio tutti i compagni di corso incontrati in questi anni, con un pensiero speciale al gruppo HARSH conosciuto durante l'Erasmus a Grenoble. In particolare, voglio ringraziare le butele Caro, Ross e Reb per le cene, le risate e le crisi di nervi condivise durante quei mesi.

Durante questo periodo ho avuto la fortuna di creare un rapporto con una fantastica persona. Grazie Albi per le chiacchierate fatte durante i progetti e i momenti di studio insieme. Mi hai trasmesso tanta serenità nell'approcciare molte cose e ti ringrazio per avermi fatto sentire bene.

Grazie Quan, compagno di risate ma anche di momenti seri. Voglio ringraziarti per questi cinque anni passati in università e a Grenoble.

Volevo ringraziare i miei due amici Davide e Mimmo. Non so nemmeno per cosa ringraziarli, l'unica cosa che mi viene in mente è che sono veramente fortunato ad averli come compagni di banco, compagni di scelte e compagni di vita. Più vi conosco e più me ne rendo conto.

Per concludere, volevo ringraziare la persona che mi ha spinto ad iscrivermi all'università subito dopo le superiori. Ti ringrazio per questi anni d'amicizia, per le parole spese, per le pacche sulle spalle, per le prime vittorie a Fortnite, per i tragitti casa mia-allenamento, per le serate alla PlayStation "Floricic vs Scaglia", ma soprattutto per essere mio amico. Grazie Jack.