UCLouvain

epl

École polytechnique de Louvain

# On the training of Shallow Neural Networks with the Gradient Method

Author: **Quentin Brisbois**
Supervisors: **Estelle Massart, Geovani Nunes Grapiglia**
Reader: **François Glineur**
Academic year 2023–2024
Master [120] in Mathematical Engineering

UCLOUVAIN

# *Abstract*

Master [120] in Mathematical Engineering

**On the training of Shallow Neural Networks with the Gradient Method**

by Quentin BRISBOIS

Due to the rise of complex and wide neural networks, there is a growing need for efficient and fast learning algorithms to support this trend. Currently, Neural Networks are trained using first-order optimization algorithms like the (Stochastic) Gradient Method, made possible by the discovery of backpropagation. This thesis aims to better understand why and how first-order optimization methods, such as the Gradient Method, perform well when minimizing nonconvex loss functions. The Gradient Method is known to work effectively for convex optimization problems, but the loss functions used in deep learning are typically highly nonconvex, even locally. To address this, we first review the literature to identify existing research in this area and summarize the findings of [5], which details the convergence of Gradient Flow with small initialization and the training dynamics of a one-hidden layer ReLU Network for orthogonal input data. After reproducing their numerical experiments, we conduct our own experiments to explore neuron behavior in groups. Notably, we experimentally find that for small initializations, the training dynamics of Sigmoid and Tanh one-hidden layer neural networks exhibit similar properties to those described in [5] for ReLU networks. Finally, we examine the Polyak-Łojasiewicz inequality for a one-hidden layer network and provide a proof of an important lemma from [5] regarding the balancedness of iterates generated by the Gradient Method, which the authors do not provide.

# *Acknowledgements*

I would like to extend my heartfelt thanks to everyone who contributed to the successful completion and writing of this Master's thesis.

First and foremost, I want to express my gratitude to my thesis supervisors, Professor Estelle Massart for the Neural Networks part and Professor Geovani Nunes Grapiglia for the Optimization part. I am deeply thankful for the time they dedicated to guiding me and answering my questions throughout this project and during its writing.

I thank the École Polytechnique de Louvain and its teaching staff for providing high-quality education in engineering. Their efforts have enabled me to develop all the necessary skills required for a young engineer's future.

I would also like to thank my parents, Isabelle and Alain, who have enabled me to pursue my studies and an engineering curriculum, always supporting me along the way.

Finally, I want to thank my brothers, Guillaume and Nathan, as well as the rest of my family, who have provided me with support and advice throughout this project. Their combined support has been invaluable and greatly contributed to the successful completion of this work over the past year.

# Contents

**Abstract** i

**Acknowledgements** iii

**1 Preliminaries** 3
1.1 Neural Networks : Preliminaries 3
1.1.1 Machine Learning, Shallow Learning and Deep Learning 3
1.1.2 Dataset 4
1.1.3 Models 5
1.1.3.1 First Artificial Neuron and Perceptron 5
1.1.3.2 Artificial Neural Networks 6
1.1.4 Predictions and Errors : Loss function 8
1.1.5 Training of the model 8
1.1.5.1 Gradient Method and Backward Propagation 8
1.2 Unconstrained Nonlinear Optimization : Preliminaries 10
1.2.1 Basic tools 11
1.2.2 Optimality conditions 12
1.2.3 Convergence of Gradient Methods 12

**2 Training of Artificial Neural Networks** 15
2.1 Training of ANN : Partial State-of-The-Art 15
2.1.1 Explosion and Vanishing gradient problem 16
2.1.2 Initialization methods 17
2.1.3 Optimization algorithm used in Deep Learning 18
2.1.4 Landscape analysis of the Loss 21
2.1.5 Convergence analysis results for Deep Networks 25
2.1.6 Results for Shallow Networks 27
2.1.7 Useful tools in Deep Learning 29
2.2 Training dynamics of one-hidden layer ReLU Neural Networks 30
2.2.1 Orthogonal input data 30
2.2.1.1 Model 31
2.2.1.2 Assumptions 32
2.2.1.3 Main results of paper [5] 33
2.2.1.4 Reproduction of numerical experiments of paper [5] 35
2.2.2 Non-orthogonal input data 37

**3 Additional Numerical Experiments** 39
3.1 Unidimensional orthogonal data 40
3.1.1 Dataset 1.1.0 : original data 40
3.1.2 Orthogonal data : role of the labels 49
3.1.2.1 Dataset 1.1.1 : Switched Labels 49
3.1.2.2 Dataset 1.1.2 : Labels of Same Sign 49
3.1.3 Dataset 1.1.4 : alternative orthogonal data 51

3.1.4 S-curve of neurons in $(w_1, w_2)$ plane . . . 54
3.2 Unidimensional quasi-orthogonal data . . . 58
3.3 Unidimensional non-orthogonal data . . . 60
3.3.1 Dataset 1.3.3 : non-orthogonal data with alternate labels . . . 60
3.3.2 Dataset 1.3.2 : non-orthogonal data with random labels . . . 62
3.4 Multidimensional data . . . 65
3.4.1 Orthogonal . . . 65
3.4.2 Non-orthogonal . . . 67
3.5 Summary of Numerical Experiments Results and Conjectures . . . 68

**4 Theoretical exploration of the loss landscape 71**
4.1 Preliminary calculation . . . 71
4.1.1 Gradient of the square loss . . . 73
4.1.2 Hessian of the square loss . . . 73
4.2 PL condition . . . 74
4.2.1 Definition and properties . . . 74
4.2.2 PL condition for Shallow Neural Networks . . . 76
4.2.3 PL condition for Neural Networks in the literature . . . 78
4.3 Proof of Lemma 5 about Balancedness of iterates $\theta^t$ . . . 78

**5 Conclusion 81**

**A Generative AI for Report Improvement 85**

**B Calculation of $\nabla^2 \mathcal{L}(\theta)$ for the square loss 87**

**C Additional Figures to Chapter 3 91**
C.1 Dataset 1.1.0 : original data . . . 91
C.1.1 Neuron representation in $(w_{j,1}, w_{j,2})$ plane . . . 91
C.1.2 Balancedness of iterates $\theta^t$ . . . 92
C.1.3 Transition phase between Lazy and Rich regimes : illustrations for Sigmoid and Tanh . . . 93
C.1.4 Automatic processing of experiment results for all generated hyper-parameter setups . . . 94
C.1.5 Curve fitting on S-curves in $(w_{j,1}, w_{j,2})$ plane . . . 98
C.2 Dataset 1.1.4 : alternative orthogonal data . . . 99
C.3 Unidimensional non-orthogonal data . . . 102
C.3.1 Dataset 1.3.1 . . . 102
C.3.2 Dataset 1.3.2 . . . 105
C.3.3 Dataset 1.3.3 . . . 107

**D Numerical check of gradient and Hessian formulas 111**
D.1 Numerical check of the gradient formulas . . . 111
D.2 Numerical check of the hessian formulas . . . 114

# List of Abbreviations

| | |
|---|---|
| **AI** | **A**rtificial **I**ntelligence |
| **ANN** | **A**rtificial **N**eural **N**etwork |
| **ML** | **M**achine **L**earning |
| **SL** | **S**hallow **L**earning |
| **DL** | **D**eep **L**earning |
| **MLP** | **M**ulti-**L**ayer **P**erceptron |
| **GD** | **G**radient **D**escent (or Gradient Method) |
| **SGD** | **S**tochastic **G**radient **D**escent (or Stochastic Gradient Method) |
| **ReLU** | **Re**ctified **L**inear **U**nit |
| **NTK** | **N**eural **T**angent **K**ernel |
| **PL** | **P**olyak-**Ł**ojasiewicz |
| **i.i.d.** | **i**ndependent and **i**dentically **d**istributed |

# List of Symbols

| | |
|---|---|
| $\|v\|$ | $\ell_2$-norm of the vector $v$; |
| $\|v\|_p$ | $\ell_p$-norm of the vector $v$; |
| $\|M\|$ | $\ell_2$-norm of the matrix $M$; |
| $\|M\|_F$ | Frobenius norm of the matrix $M$; |
| $M^T$ | transposed matrix of $M$; |
| $\langle v_1, v_2 \rangle = v_1^T v_2$ | scalar product of vector $v_1$ by vector $v_2$; |
| $\mathbb{1}_C$ | function equals to 1 when the condition $C$ is true, and 0 otherwise; |
| $I_d$ | Identity matrix of dimension $d$ by $d$; |
| $\nabla_\theta F$ | gradient of the function $F$ with respect to $\theta$; |
| $\nabla_\theta^2 F$ | Hessian of the function $F$ with respect to $\theta$; |
| $\mathcal{J}_g$ | Jacobian of the mapping $g : \mathbb{R}^D \to \mathbb{R}^n$ ($\mathcal{J}_g \in \mathbb{R}^{n \times D}$); |
| $\lambda(M)$ | eigenvalue of the matrix $M$; |
| | |
| $L_F$ | Lipschitz constant of the function $F$; |
| $\beta_F$ | Smoothness constant of the function $F$; |
| $\mu$ | PL constant (only in Chapter 4); |
| | |
| $n$ | number of samples in the dataset; |
| $d$ | dimension of $x_k$, i.e. $x_k \in \mathbb{R}^d$; |
| $\{(x_k, y_k)\}_{k=1,\cdots,n}$ | dataset used in supervised learning; |
| $x_k$ | $k^{\text{th}}$ input of a machine learning model; |
| $y_k$ | label or true value corresponding to $x_k$; |
| $(X, y)$ | matrix form of dataset used in supervised learning with $X \in \mathbb{R}^{n \times d}$ and $y \in \mathbb{R}^n$; |
| $\bar{X}$ | input data matrix without the bias term (only in Chapter 3); |
| | |
| $L$ | number of hidden layers in a fully-connected neural networks; |
| $D$ | number of weights in a neural network model; |
| $m^{(l)}$ | number of neurons in the $l^{\text{th}}$ layer of the neural network; |
| | |
| $w_{i,j}^{(l)}$ | weight connecting neuron $j$ of layer $(l-1)$ to neuron $i$ of layer $(l)$ (for a MLP); |
| $W^{(l)}$ | weight matrix of the layer $l$ of dimension $m^{(l)}$ by $m^{(l-1)}$; |
| $w_j^{(l)}$ | weights of the neuron $j$ of layer $(l)$; |
| $a_{i,j}$ | weight connecting neuron $j$ to output $i$ in the output layer (for a MLP), i.e. $a_{i,j} = w_{i,j}^{(L+1)}$; |
| $\theta$ | vector containing all the weights in the model, i.e. $\theta = \begin{bmatrix} W^{(1)} & W^{(2)} & \cdots & W^{(L)} & a \end{bmatrix}$; |
| | |
| $\sigma(.)$ | generic symbol for an activation function; |
| $\sigma_R(.)$ | ReLU activation function; |
| $\sigma_S(.)$ | Sigmoid activation function; |
| $\sigma_T(.)$ | Tanh activation function; |
| $z_j^{(l)}(x)$ | output of neuron $j$ of layer $(l)$ for input data $x$; |
| $z_{j,k}^{(l)}$ | output of neuron $j$ of layer $(l)$ for input data $x_k$, i.e. $z_{j,k}^{(l)} = z_j^{(l)}(x_k)$; |

| | |
|---|---|
| $s_j^{(l)}(x)$ | aggregation function of neuron $j$ of layer $(l)$ for input data $x$; |
| $s_{j,k}^{(l)}$ | aggregation function of neuron $j$ of layer $(l)$ for input data $x_k$, i.e. $s_{j,k}^{(l)} = s_j^{(l)}(x_k)$; |
| $h_\theta(x)$ | model with weight $\theta$ and generalizing on data input $x$; |
| $y_{pred,k}(\theta)$ | prediction of the model for input data $x_k$, i.e. $y_{pred,k}(\theta) = h_\theta(x_k)$; |
| $\mathcal{L}(\theta)$ | loss function as a function of weights $\theta$; |
| $\alpha_t$ | step-size of an iterative optimization algorithm at epoch $t$; |
| $\theta^t$ | value of $\theta$ at epoch $t$ generated by an iterative optimization algorithms, or $\theta$ as a function of the time $t$ for the Gradient Flow; |
| $\theta^*_{\mathrm{GD}}$ | limit point of the sequence of $\{\theta^t\}$ generated by the GD, i.e. $\theta^*_{\mathrm{GD}} = \lim_{t\to+\infty}\theta^t$; |
| $\lambda$ | scale of initialization; |
| $S_{1,-}$ | set of neurons at initialization (see definition 5); |
| $S_{1,+}$ | set of neurons at initialization (see definition 5); |
| $D_-$ | alignment direction of neurons in $S_{1,-}$ in the $(w_{j,1}, w_{j,2})$ representation; |
| $D_+$ | alignment direction of neurons in $S_{1,+}$ in the $(w_{j,1}, w_{j,2})$ representation; |
| $s_j$ | sign of the output weight at initialization $a_j^0$; |
| $\bar{D}_-$ | alignment direction of neurons in $S_{1,-}$ in the $(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|)$ representation; |
| $\bar{D}_+$ | alignment direction of neurons in $S_{1,+}$ in the $(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|)$ representation; |
| $\delta$ | the deviation of the data with respect to the orthogonal ones, i.e. $\lvert\langle x_k, x_l\rangle\rvert \leq \delta$. |

# Introduction

In recent years, there has been an emergence of Artificial Neural Networks (ANN), and their applications in our daily lives. In classical fully-connected neural networks, the typical size in applications can involve models with a few dozen to even hundreds of thousands of weights. For more complex structures, we can find various domains of application with larger neural network models. Examples include image recognition for autonomous driving cars, such as Tesla's Autopilot, which involves 48 networks that take 70,000 GPU hours to train and output 1,000 distinct tensors (predictions) at each time step (see [39]) ; predicting personal gene expression from DNA sequences using deep convolutional neural networks (CNNs) (in [2]) ; Google's image recognition model Inception-v4, which has 43 million parameters (see [23] the documentation of Inception-v4) ; large language models (LLMs) such as ChatGPT with 175 billion parameters (for version GPT-3.5-turbo); the BERT model with its 340 million parameters (from [7]) ; and the Leela Chess Zero chess program with its 47 million parameters (see the documentation of Leela [29], and this online article [16]). The size of these models is really big, and has tended to increase over the years. The need for good learning algorithms is therefore paramount to support this trend and the quality of the resulting models for the various Deep Learning applications.

Because of the size of ANN models, the training process for such models is challenging since it is costly to compute elements needed for the training algorithm, such as the gradient and Hessian of the loss function used to measure the error between the training data and the model's predictions. Thanks to the discovery of the backward propagation, we can now more efficiently compute the gradient of the loss function, which has enabled the use of first-order optimization methods (such as Stochastic Gradient Descent (SGD) and Gradient Descent (GD)). Moreover, the learning process of ANN models is possible, in part because first-order methods of optimization are surprisingly efficient in minimizing the training loss associated with these models. Indeed, (S)GD is theoretically efficient for convex optimization problems since it converges to the global minimum, if one exists. However, the training loss of an ANN is known to be nonconvex.

In this context, the main objective of our thesis is to better understand why and how first-order optimization methods like (S)GD converge to good minimizers (and sometimes to a global minimizer) of the training loss of an ANN. To achieve this, we begin with some preliminaries in Chapter 1, which is divided into two parts. This work is written for readers with a background in optimization but not necessarily in Machine Learning and Deep Learning. Thus, the first part introduces basic concepts in Machine Learning to set the context and establish the notations used throughout the work. We briefly introduce the dataset used in supervised learning, neural network models, the concept of a loss function, and how to train the model with gradient-based optimization algorithms. The second part of the preliminaries recalls some tools in nonlinear optimization and discusses why classical convergence analysis results for the Gradient Method are inadequate in Deep Learning. In Chapter 2, we review some recent results about the training of artificial neural networks. In particular, we examine the findings outlined in [5] about the training of ReLU neural networks with single hidden layer. Under strong assumptions, we observe zero training loss, implicit bias of Gradient Flow toward the smallest norm interpolator (i.e., the simplest model possible based on the $\ell_2$-norm criterion of the weights), saddle-to-saddle dynamics, and neuron alignment. These assumptions include having orthogonal input data, non-empty sets of neurons at the ANN's initialization, and very small random initialization (leading to an interesting regime called the "rich regime") that is often more beneficial for learning.

Next, we show the reproduction of the numerical experiments in [5] to confirm their validity. At the end of the chapter, we briefly summarize [34], which provides results in the exact same settings with non-orthogonal data. Drawing from this, in Chapter 3, we conduct numerical experiments on one-hidden layer networks with the same neural structure as in [5]. By adjusting certain hyperparameters, we aim to uncover the behavior of neurons (and weights) in groups when relaxing some of the assumptions made in [5]. We perform our numerical experiments using either ReLU, Sigmoid, or Tanh activation functions in the hidden layer. From these experiments, we confirm the training dynamics predicted by [5] for a one-hidden layer ReLU network with orthogonal data. We also explore the similarities and differences in the training dynamics of such networks with non-orthogonal data. Additionally, we observe similar behavior in the training dynamics of one-hidden layer ANNs with Sigmoid and Tanh activation functions, leading us to make some conjectures about their training at the end of the chapter. Specifically, we discover that, with small initialization, the neurons of the hidden layer form an S-curve in their parameter space for Sigmoid and Tanh activation functions. Finally, in Chapter 4, we conduct a brief theoretical exploration of the convergence of GD for the square loss of one-hidden layer ANNs from an optimization perspective, using tools like the PL condition to attempt to understand the convergence of the Gradient Method to zero loss. We also provide a proof of an important lemma from [5] regarding the balancedness of the iterates generated by the Gradient Method during the training process, as the authors did not provide it.

## Usual notations

Before starting the first chapter, we define some usual notations used throughout this work to ensure clarity. For $x, y \in \mathbb{R}^d$, we denote the scalar product as $x^T y = \langle x, y \rangle$. We denote the $\ell_2$-norm by $\|x\|$ and the $\ell_p$-norm by $\|x\|_p$. For a matrix $A$, we denote the Frobenius norm by $\|.\|_F$. We define $\mathbb{1}_C$ as the function equals to 1 when the condition $C$ is true, and 0 otherwise. The identity matrix of dimension $d$ is denoted as $I_d$. We denote the Jacobian of the mapping $g : \mathbb{R}^D \to \mathbb{R}^n$ as $\mathcal{J}_g \in \mathbb{R}^{n \times D}$. The gradient and the Hessian of the function $F$ with respect to the vector of variables $\theta$ are denoted as $\nabla_\theta F$ and $\nabla^2_\theta F$ respectively. We denote an eigenvalue of a square matrix $M$ as $\lambda(M)$.

# Chapter 1

# Preliminaries

In this first chapter, we introduce important concepts and their corresponding notations used throughout this work. We begin with some preliminaries in Machine Learning to establish the context of our exploration, followed by an introduction to artificial neurons. Building on this, we present the structure of fully-connected neural networks (also known as Multi-Layer Perceptrons), which will be used throughout the study. Next, we discuss the concept of loss functions and how to train a fully-connected neural network. Finally, we introduce basic notions in optimization theory, such as convex function and Lipschitz continuity, and discuss Gradient Method and some classic results associated with them.

## 1.1 Neural Networks : Preliminaries

We begin by introducing the Machine Learning framework within which this thesis is situated, along with an overview of Artificial Neural Network models and their training process. The aim is to establish the fundamental concepts of Artificial Neural Networks and to introduce the notation that will be used in subsequent chapters.

### 1.1.1 Machine Learning, Shallow Learning and Deep Learning

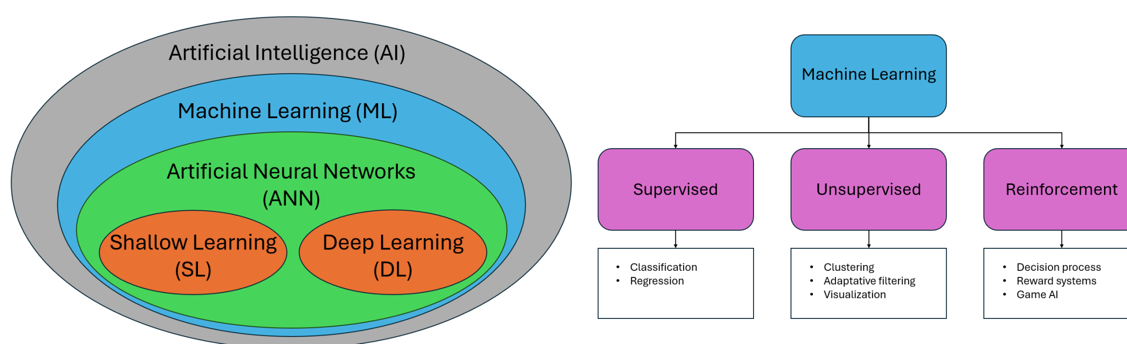



FIGURE 1.1: (Left) Interconnection between Artificial Intelligence, Machine Learning, Artificial Neural Networks, Shallow and Deep Learning; (Right) Common learning methods in Machine Learning.

We begin this section by providing some short informal definitions about Machine Learning, aiming to contextualize this study. You can see how these concepts are interconnected in Figure 1.1.

**Machine Learning** (ML) is a domain of artificial intelligence. The goal is to develop a model that is able to learn from a set of data in order to generalize to unseen data, using a learning algorithm.

An **Artificial Neural Network** (ANN) is composition of functions that are structured by layers. We define this notion more formally in the section 1.1.2.

**Shallow Learning** (SL) is a branch of Machine Learning that uses Shallow Artificial Neural Networks as models, i.e. ANNs with few layers. The advantage of Shallow ANNs is that they are easy to train, but their performance is very limited when used for complex applications.

**Deep Learning** (DL) is a branch of Machine Learning that uses Deep Artificial Neural Networks as models, i.e. ANNs with many layers. In practice, these types of models are more useful than shallow ones for many complex tasks, but they are more difficult to train.

There are 3 main approaches to learn in Machine Learning :

- **Supervised Learning** : we give to the learning algorithm a dataset, that contains the inputs of the model and the desired corresponding output (called label). The learning algorithm is often an optimization algorithm that minimizes a function that computes the error between the prediction made by the model and the label for each input. This function is called a Loss Function.
  There are two main tasks we can deal with in supervised machine learning : regression tasks and classification tasks.
  - **Regression** involves predicting a continuous value based on certain features, such as predicting the price of a house based on factors like location, size, etc.
  - **Classification** involves predicting the class to which an input belongs, such as predicting the gender of a person based on their age, height, etc.
- **Unsupervised Learning** : we only give to the learning algorithm the data without their corresponding labels. The goal is to find a structure in the dataset.
- **Reinforcement Learning** : this approach mimics the learning behavior of an agent in an uncertain environment. The agent is in a given situation, and must make a decision. Then, he receives a random reward from his environment. Based on this reward, the agent can learn by evaluating the decision he took in the given situation.

In this thesis, we will focus on Shallow Neural Networks from the third chapter, and especially on Neural Networks with one hidden layer. Moreover, we will stay in a Supervised Learning context for regression tasks.

In the following section, we will describe the dataset (subsection 1.1.2), the fully-connected ANN models (subsection 1.1.3), the method for calculating the error between the model's predictions and the true labels (subsection 1.1.4), and the optimization algorithms used to train the model by minimizing these errors (subsection 1.1.5).

### 1.1.2 Dataset

We use a dataset of $n$ elements $\{(x_k, y_k)\}_{k=1,\cdots,n}$ where each $x_k \in \mathbb{R}^d$ is a vector of features, and $y_k \in \mathbb{R}$ is its true label[1], i.e. the value we aim to predict at the output of the model for the input $x_k$. We define also the input data matrix $X \in \mathbb{R}^{n\times d}$ with each row corresponding to a vector of features $x_k$, and the label vector $y \in \mathbb{R}^n$.

[1] We only consider one-dimensional output, as we are working in a one-dimensional regression setting.

In general, in Machine Learning, we divide the dataset into two subsets. The first one is the training set, which is used to train the model. The second one is the test set. We never use this data in the training process, so it remains unseen to the model. Since we also have labels for the test set, it can be used to evaluate the quality of the final model. In this work, we focus on the training process of an ANN, so we only need to consider the training dataset.

### 1.1.3 Models

#### 1.1.3.1 First Artificial Neuron and Perceptron

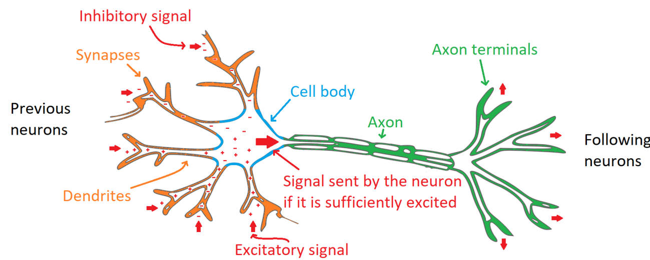



FIGURE 1.2: Simplified functioning of a biological neuron.

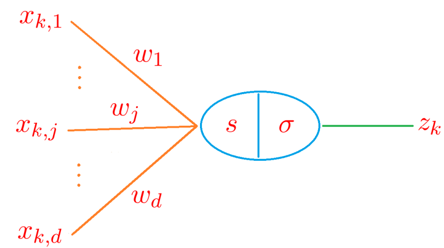



FIGURE 1.3: Model of an artificial neuron.

An artificial neuron mimics in a simple way the functioning of a biological neuron (see Figure 1.2 and [11] for more information). Indeed, a biological neuron is an excitable cell that is connected with other ones, and that transmits information to the other ones. A neuron receives a signal from the predeceasing neurons by the synapse. It can receive two types of signals : inhibitory or excitatory. The signals go through the dendrites until the cell body. There, if the excitation of the neuron is higher than a certain threshold, the neuron is activated and sends an electric signal to the following neurons by its axon and axon terminals.

The artificial neuron models this behavior in a simple way. The inputs and output signals are modeled by numbers $x_k \in \mathbb{R}^d$ and $z_k \in \mathbb{R}$ respectively. The excitatory or inhibitory character and its intensity are modeled by weights $w_j$, that links the input component $x_{k,j}$ and the neuron. The activation of a neuron is modeled by an aggregation function

$$s : \mathbb{R}^d \to \mathbb{R} : x \mapsto s(x) = \sum_{j=1}^{d} x_j$$

and an activation function $\sigma : \mathbb{R} \to \mathbb{R} : x \mapsto \sigma(x)$.

We can use various activation functions. While we can certainly use a linear activation function, such as $\sigma(x) = x$, it is in general more interesting to employ nonlinear activation functions. The three main ones commonly used in the literature are Rectified Linear Unit (ReLU), Sigmoid and Hyperbolic Tangent (Tanh). Their respective expressions are

$$\mathrm{ReLU}(x) = \max\{0, x\} = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

$$\mathrm{Sigmoid}(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{1.2}$$

$$\text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.3}$$

In the remainder of this work, we will denote the ReLU, Sigmoid and Tanh activation functions in mathematical expressions by $\sigma_R$, $\sigma_S$ and $\sigma_T$ respectively.

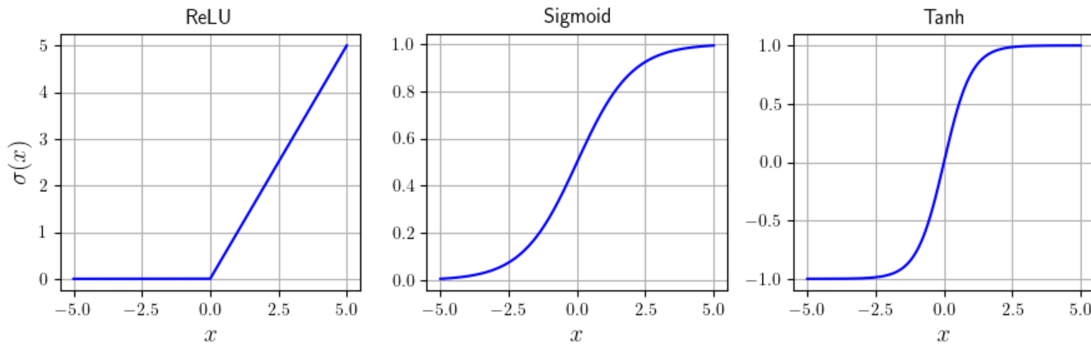



FIGURE 1.4: Graphs of the ReLU, Sigmoid and Tanh activation functions.

At the end, the model formed by one artificial neuron is

$$h_w(x) = \sigma\left(\sum_{j=1}^{d} w_j x_j\right) \tag{1.4}$$

However, this simple structure is not powerful enough for many problems. If $\sigma(x) = x$, the model is a linear regressor, and if $\sigma$ is a nonlinear activation function, the model can only fit a function of the form $\sigma$ to the data. To create a more complex and effective model, the idea is to stack neurons, one after the other, forming a neural network.

**Remark** : It is common to add a bias to our artificial neuron. But we can keep the previous notations by posing that the last component of each feature vector is equal to 1, i.e. $x_{k,d} = 1\ \forall k$. For example, consider a dataset with unidimensional input $\tilde{X} = [-0.5, 2]^T$. Adding the bias, we finally get

$$X = \begin{bmatrix} -0.5 & 1 \\ 2 & 1 \end{bmatrix}.$$

As noted above, we denote the matrix of input data without the bias term as $\tilde{X}$ and the one with the bias term as $X$ to clearly distinguish between them.

#### 1.1.3.2 Artificial Neural Networks

Here, we describe the principle of a fully-connected neural network, also known as a multi-layer perceptron (abbreviated by MLP). We will not go into more detail about more complex structures for ANNs, such as Convolutional Neural Networks or other more elaborate ones, as they are beyond the scope of this work.

The main idea of fully-connected neural network is to put together many artificial neurons by connecting the inputs of neurons with the outputs of the previous ones, and to structure them by layers. You can see this kind of structure on the figure 1.5. Because of the complexity of the structure, we need clear notations to identify clearly each element of the network. On the right of Figure 1.5, we display the main notations of expressions related to a neuron $i$ of a layer $l$.

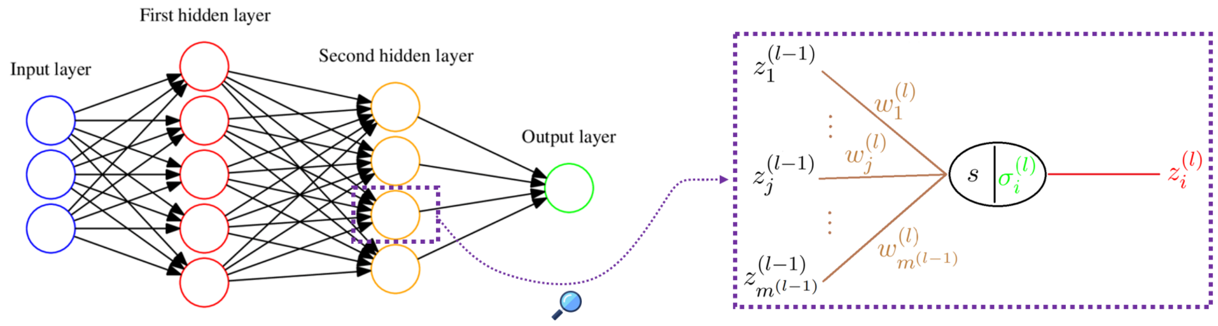



FIGURE 1.5: Example of a fully-connected neural network structure with 2 hidden layers (Left), and the notations associated with one neuron (Right). (Figure adapted from [17])

More precisely, we denote

- the number of hidden layers in the network : $L$,
- the number of neurons in layer $(l)$ : $m^{(l)}$,
- the weight connecting the neuron $j$ of layer $(l-1)$ to the neuron $i$ of layer $(l)$ : $w_{i,j}^{(l)}$,
- in order to differentiate weights related to hidden layers and weights related to output layer, we will denote : $w_{i,j}^{(L+1)} = a_{i,j}$,
- the matrix containing all weights of the networks : $\theta = \begin{bmatrix} W^{(1)} & W^{(2)} & \cdots & W^{(L)} & a \end{bmatrix}$ with $W^{(l)} \in \mathbb{R}^{m^{(l)} \times m^{(l-1)}}$ and $a \in \mathbb{R}^{m^{(L)}}$ the matrices of weights related to each layer,
- the output of the neuron $i$ of layer $(l)$ for input data $x$ : $z_i^{(l)}(x)$, and by definition,

$$z_i^{(l)}(x_k) = z_{i,k}^{(l)}$$

$$z_{j,k}^{(0)} = x_{k,j}$$

$$z_{j,k}^{(L+1)} = y_{pred,k,j}$$

  where $y_{pred,k}$ is the prediction of the model for input $x_k$;
- the aggregation function of the neuron $i$ of layer $(l)$ for input data $x$ :

$$s_i^{(l)}(x) = \sum_{j=1}^{m^{(l-1)}} w_{i,j}^{(l)} z_j^{(l-1)}(x) \tag{1.5}$$

- so we have the following relation : $z_i^{(l)}(x) = \sigma_i^{(l)}\left(s_i^{(l)}(x)\right)$.

In order to simplify a bit the notation, we can note

$$\sigma_i^{(l)}\left(s_{i,k}^{(l)}\right) = \sigma\left(s_{i,k}^{(l)}\right)$$

since the indices $i$ and $l$ of $\sigma_i^{(l)}$ are always the same as those of $s_{i,k}^{(l)}$. Moreover, since we are more interested by regression problem with one output value (i.e. $y_{pred,k} \in \mathbb{R}$), we can drop the $j$ index in above expressions related to the output layer $L+1$.

With these notations, we are able to express the model iteratively as

$$h_\theta(x) = \sigma^{(L+1)}\left(s_{1,k}^{(L+1)}\right) \tag{1.6}$$

and with the expressions defined above. It should be noted that $h$ can be viewed as a function of the input data when considering the prediction ability of the model, or as a function of weights $\theta$ when considering the training of the model. In this context, we will denote it as $h_\theta(x)$ or $h_x(\theta)$ depending on the focus.

### 1.1.4 Predictions and Errors : Loss function

In order to make predictions with respect to an input $x_k$ with the model, we pass the $x_k$ at the entry of the neural network, we compute the outputs of the neurons of the current layer. After that, we pass these outputs as inputs of the following layers, and we repeat this until the output layer. We call this process a **forward propagation**, and the output of the last layer gives the prediction $y_{pred,k}$. Mathematically, it is equivalent to

$$y_{pred,k}(\theta) = h_\theta(x_k). \tag{1.7}$$

We aim to have a model that satisfies (as much as possible) the following constraints :

$$y_{pred,k}(\theta) \approx y_k \quad \forall k \in \{1, \cdots, n\}. \tag{1.8}$$

In order to evaluate the quality of the predictions, we can define a function that computes the errors between the predictions and the corresponding labels. We call this function a **Loss Function**. There exist several loss functions, and in general, they take the following form :

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{k=1}^{n} \ell_k(y_{pred,k}(\theta), y_k). \tag{1.9}$$

The most commonly used in the regression context is the square loss function with $\ell_k(y_{pred,k}(\theta), y_k) = \left(y_{pred,k}(\theta) - y_k\right)^2$.

### 1.1.5 Training of the model

The principle of training a neural network is to adjust the weights $\theta$ of the model in order to have a model that fits as good as possible the data. Mathematically, as mentioned in the previous section, we need to adjust the weights in order to satisfy the constraints (1.8). To reach this goal, we can use the loss function we defined in 1.1.4, and we can minimize it with respect to these weights $\theta$.

$$\min_{\theta \in \mathbb{R}^D} \mathcal{L}(\theta) \tag{1.10}$$

with $D$ the number of weights in the model. This is an unconstrained nonconvex optimization problem. We recall some basic tools related to this field in Section 1.2.

#### 1.1.5.1 Gradient Method and Backward Propagation

The simplest algorithm used to minimize the loss $\mathcal{L}$ is the Gradient Method. It consists to start from an initial weights vector $\theta^0$, and iteratively compute a new $\theta$ by making a step in the opposite direction of the gradient of the loss. Formally,

$$\theta^{t+1} = \theta^t - \alpha_t \nabla \mathcal{L}\left(\theta^t\right) \tag{1.11}$$

with $\alpha_t$ the step-size of the Gradient Method at $t$, where $t$ denotes the $t^{\text{th}}$ epoch. We call an **epoch** a pass through the whole training data in the optimization algorithm[2]. Thus, for the Gradient Method, an iteration is equivalent to an epoch.

The Gradient Method has nice properties when the function to minimize is convex, or at least locally convex around their local minima. However, the square loss function $\mathcal{L}$ is in general deeply nonconvex. We will discuss it later.

Applying the gradient method to the training of a deep neural network is not even simple. The most difficult thing to compute is the gradient of loss function $\nabla\mathcal{L}$. Since the model is a composition of functions structured by layers, it is possible to compute the gradient by using the chain rule of partial derivative of $\mathcal{L}$ with respect to its weights. So, we need to begin with the output layer, and we continue iteratively until the input layer of the neural network. We call this method the **backward propagation**[3]. The Algorithm 1 shows a pseudo-code of complete process.

**Algorithm 1** Training of ANN

1: **procedure** TRAINING_ANN($X, y, \theta^0, \alpha$)
2: Initialization : $\theta \leftarrow \theta^0$
3: **for** $epoch \in \{1, \cdots, N_{epochs}\}$ **do**
4: Forward propagation : $y_{pred,k}(\theta) = h_\theta(x_k) \quad \forall k\{1, \cdots, n\}$
5: Loss value : $\mathcal{L}(\theta) = \frac{1}{2n}\sum_{k=1}^{n}\left(y_{pred,k}(\theta) - y_k\right)^2$
6: Back propagation : compute $\nabla\mathcal{L}(\theta)$
7: Gradient Step : $\theta \leftarrow \theta - \alpha\nabla\mathcal{L}(\theta)$ ▷ or other Gradient-based opti. algo.
8: **return**

For a general Fully-Connected ANN with a scalar output $y_{pred} \in \mathbb{R}$, we can easily write expressions for the partial derivative of the square loss. It allows to calculate the gradient of $\mathcal{L}(\theta)$. In order to do that, we use the chain rule of derivative. So, we get

$$\frac{\partial\mathcal{L}}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^{n}\frac{\partial\mathcal{L}}{\partial s_{i,k}^{(l)}}\frac{\partial s_{i,k}^{(l)}}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^{n}\frac{\partial\mathcal{L}}{\partial s_{i,k}^{(l)}} z_{j,k}^{(l-1)}.$$

[2]Or a pass with the same computational cost when the optimization algorithm randomly chooses the indices of the data at each iteration.

[3]As said in [38], "From an optimization perspective, it is just an efficient implementation of gradient computation".

Now, we have 2 cases :

1. Output layer $l = L+1$ : for this case, we have enough information to derive an explicit formula.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial s_{1,k}^{(L+1)}} &= \frac{\partial \mathcal{L}}{\partial z_{1,k}^{(L+1)}} \frac{\partial z_{1,k}^{(L+1)}}{\partial s_{1,k}^{(L+1)}} \\
&= \frac{\partial \mathcal{L}}{\partial y_{pred,k}} \frac{\partial y_{pred,k}}{\partial s_{1,k}^{(L+1)}} \\
&= \frac{1}{n}(y_{pred,k} - y_k) \frac{\partial}{\partial s_{1,k}^{(L+1)}} \left( \sigma \left( s_{1,k}^{(L+1)} \right) \right) \\
&= \frac{1}{n}(y_{pred,k} - y_k)\, \sigma' \left( s_{1,k}^{(L+1)} \right)
\end{aligned}$$

$$\Longrightarrow \frac{\partial \mathcal{L}}{\partial a_j} = \frac{\partial \mathcal{L}}{\partial w_{1,j}^{(L+1)}} = \frac{1}{n} \sum_{k=1}^{n} (y_{pred,k} - y_k)\, \sigma' \left( s_{1,k}^{(L+1)} \right) z_{j,k}^{(L)}$$

2. Hidden layer $l \leq L$ : here, we derive a recursive formula with respect to the following layer $(l+1)$.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial s_{i,k}^{(l)}} &= \sum_{p=1}^{m^{(l+1)}} \frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l+1)}} \frac{\partial s_{p,k}^{(l+1)}}{\partial s_{i,k}^{(l)}} \\
&= \sum_{p=1}^{m^{(l+1)}} \frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l+1)}} \frac{\partial}{\partial s_{i,k}^{(l)}} \left( \sum_{j=1}^{m^{(l)}} w_{p,j}^{(l+1)} z_{j,k}^{(l)} \right) \\
&= \sum_{p=1}^{m^{(l+1)}} \frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l+1)}} w_{p,i}^{(l+1)} \sigma' \left( s_{i,k}^{(l)} \right)
\end{aligned}$$

$$\Longrightarrow \frac{\partial \mathcal{L}}{\partial w_{i,j}^{(l)}} = \sum_{k=1}^{n} \left[ \sum_{p=1}^{m^{(l+1)}} \frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l+1)}} w_{p,i}^{(l+1)} \sigma' \left( s_{i,k}^{(l)} \right) \right] z_{j,k}^{(l-1)}.$$

It can be also done to derive the expression of the Hessian $\nabla^2 \mathcal{L}(\theta)$. You can find calculations in appendix B.

For more complex neural network structures, backpropagation can be performed using *automatic differentiation*. This method is a principled approach to computing gradients of functions built as a series of successive operations. For example, as reported in [42], well-known libraries such as PyTorch and TensorFlow are implemented with this method.

## 1.2 Unconstrained Nonlinear Optimization : Preliminaries

Here, we recall some basic tools of unconstrained nonlinear optimization and well-known convergence results of Gradient Methods. We just mention results which can be applied to nonconvex objective functions, since our goal at the end is to understand the behavior of Gradient Methods for the Loss function used to train ANN models. All these basic notions will be addressed in Section 2.1.

In this section, let consider a (possibly nonconvex) function $F : \mathbb{R}^D \mapsto \mathbb{R} : \theta \rightarrow F(\theta)$.

### 1.2.1 Basic tools

We start by defining what a stationary point is.

**Definition 1.** ***(Stationary point)*** *A stationary point $\theta^*$ of a differentiable function $F$ is a point such that the gradient at this point is zero, i.e. $\nabla F(\theta^*) = 0$.*

A stationary point can be either a minimum, a maximum or a saddle point. This is an important concept, as many iterative optimization algorithms use the gradient direction to determine the step direction to take.

Another definition which is often needed for convergence results in nonlinear optimization is the Lispchitz continuity. Roughly speaking, this notion quantifies the maximal rate at which the value of a function can vary while the coordinates change.

**Definition 2.** ***($L_F$-Lipschitz function)*** *A function $F$ is said $L_F$-Lipschitz continuous if and only if $\forall \theta_1, \theta_2 \in \mathbb{R}^D$*

$$\|F(\theta_1) - F(\theta_2)\| \leq L_F \|\theta_1 - \theta_2\|. \tag{1.12}$$

When we apply this definition to the gradient of a function, we obtain the notion of smoothness.

**Definition 3.** ***($\beta$-smooth function)*** *A continuously differentiable function $F$ is said $\beta$-smooth if and only if its gradient is $\beta_F$-Lipschitz continuous, i.e. $\forall \theta_1, \theta_2 \in \mathbb{R}^D$*

$$\|\nabla F(\theta_1) - \nabla F(\theta_2)\| \leq \beta_F \|\theta_1 - \theta_2\|. \tag{1.13}$$

Also, to define a nonconvex function, it is necessary to define what a convex one is.

**Definition 4.** ***(Convex function)*** *A function $F$ is convex if and only if $\forall \theta_1, \theta_2 \in \mathbb{R}^D$ and $t \in [0, 1]$*

$$F(t\, \theta_1 + (1-t)\, \theta_2) \leq t\, F(\theta_1) + (1-t)\, F(\theta_2). \tag{1.14}$$

If these inequalities are strict, we say that the function is strictly convex.

**Example.** To illustrate these concepts, when applied to the three activation functions considered in this work, we find that :

- ReLU function is convex with a Lipschitz constant $L_{\text{ReLU}} = 1$, but it is not smooth.
- Sigmoid and Tanh are both nonconvex, Lipschitz with $L_{\text{Sigmoid}} = \frac{1}{4}$ and $L_{\text{Tanh}} = 1$, and smooth with $\beta_{\text{Sigmoid}} = \frac{1}{6\sqrt{3}}$ and $\beta_{\text{Tanh}} = \frac{4}{3\sqrt{3}}$.

To obtain the values of the Lipschitz constant $L$ and the smoothness constant $\beta$ for Sigmoid and Tanh functions, we use the result that states if a function $F$ is differentiable, $F$ is $L_F$-Lipschitz continuous if and only if $\|\nabla F(x)\| \leq L_F$ for all $x \in \mathbb{R}^D$ (see [20]). This leads to $L_\sigma = \max_x |\sigma'(x)|$ and $\beta_\sigma = \max_x |\sigma''(x)|$. For the ReLU function, we simply use the definition 2.

### 1.2.2 Optimality conditions

We consider the following minimization problem

$$\min_{\theta \in \mathbb{R}^D} F(\theta) \tag{1.15}$$

which is an unconstrained optimization problem. This type of problem is particularly relevant to us because minimizing the loss function used to train a neural network is also an unconstrained optimization problem. From [4], we have these well-known propositions characterizing the optimality of a given point $\theta^*$.

**Proposition 1.** ***(Necessary Optimality Conditions)*** *Let $\theta^*$ be an unconstrained local minimum of $F$, and assume that $F$ is continuously differentiable in an open set $S$ containing $\theta^*$. Then*

$$\nabla F(\theta^*) = 0. \quad \textit{(First Order Necessary Condition)} \tag{1.16}$$

*If in addition $F$ is twice continuously differentiable within $S$, then*

$$\nabla^2 F(\theta^*) \succcurlyeq 0. \quad \textit{(Second Order Necessary Condition)} \tag{1.17}$$

**Proposition 2.** ***(Second Order Sufficient Optimality Conditions)*** *Let $F$ be twice continuously differentiable in an open set $S$. Suppose that a point $\theta^* \in S$ satisfies the first and second order necessary conditions stated in Proposition 1. Then $\theta^*$ is an unconstrained local minimum of $F$.*

These propositions are useful for characterizing the local minimality of a stationary point for a continuously differentiable function, such as a neural network with Sigmoid or Tanh activation functions.

### 1.2.3 Convergence of Gradient Methods

Consider the following Gradient method :

$$\theta^{t+1} = \theta^t - \alpha_t \nabla F(\theta^t). \tag{1.18}$$

As said in [4], the best we can expect for a gradient-based algorithm for a general objective function, and thus for a nonconvex function such as the training loss of ANNs, is that the sequence (or a sub-sequence) of iterates $\theta^t$ converges to a stationary point $\bar{\theta}$. We list below two classical convergence results which provide conditions to ensure that every limit point of a sequence of iterates generated by the Gradient Method is a stationary point.

As provided below as illustration, there are two types of classical convergence results for Gradient Descent. The first type is obtained via Line Search and applies to any differentiable function. However, these methods are costly since they require minimization or multiple evaluations of the objective function, and they do not perform well with large-scale problems such as those encountered in deep learning. The second type of results requires Lipschitzness of the gradient of the objective function. Unfortunately, the gradient of the loss functions used to train Neural Networks does not generally have a global Lipschitz constant (that is the case for loss of the form given by expression 1.9). This issue can be circumvented by assuming that iterates are always bounded or by constraining iterates to lie within a certain ball. However, these approaches have theoretical drawbacks, and the Lipschitz constant can be very small or very large, making Gradient Descent with a step-size related to the Lipschitz constant (e.g. $\alpha = \frac{1}{\beta_{\mathcal{L}}}$) impractical in many situations. Moreover, in [32], the authors show that convexity-based convergence analysis is inadequate for studying

over-parameterized models, which is often the case for neural networks (This is discussed in more detail in Section 2.1.4). All of this underscores the importance of finding new frameworks to characterize the convergence of gradient methods for functions similar to the loss functions used in training neural networks. Our first step towards this goal is to explore the existing literature on this topic. In the next chapter, we provide a partial state-of-the-art review of training in deep learning.

**Examples.** As illustration of common results about the convergence of GD, we provide two theorems. For line search methods, we have the following theorem from [4] that ensures the convergence of the Gradient Method.

**Theorem 1.** ***(Stationarity of Limit Points for Gradient Methods)*** *Let $\{\theta^t\}$ generated by Gradient Method 1.18, and assume that $\alpha_t$ is chosen by the minimization rule, or the limited minimization rule, or the Armijo rule[4]. Furthermore, if the function $F$ is bounded from below, then every limit point of $\{\theta^t\}$ is a stationary point of $F$.*

The second kind of results needs the smoothness constant of $F$. The following theorem from [35] illustrates it.

**Theorem 2.** ***(Global convergence of GD)*** *Suppose that $F : \mathbb{R}^D \to \mathbb{R}$ is $\beta_F$-smooth, and let $\{\theta^t\}$ be generated by the Gradient Method with step-size $\alpha = \frac{1}{\beta_F}$ from any initial point $\theta^0$.*
*If $F$ is bounded from below by some $F_{low} \in \mathbb{R}$, then*

$$\lim_{t \to +\infty} \|\nabla F(\theta^t)\| = 0. \tag{1.19}$$

*If additionally, the sublevel set $\mathcal{L}_F(\theta^0) = \{\theta \in \mathbb{R}^D | F(\theta) \leq F(\theta^0)\}$ is bounded, then $\{\theta^t\}$ has a subsequence that converges to a stationary point of $F$.*

[4] See the reference [4] for the definitions of these step-size selection rules.

# Chapter 2

# Training of Artificial Neural Networks

In this chapter, we begin with a partial state-of-the-art review of the training in deep learning. We briefly summarize results from several articles to provide an overview of the progress made in the field and to introduce some useful concepts used in the discussion of numerical experiments. As the field is currently booming and the number of published articles is huge, it is impossible to give a complete overview. Secondly, we summarize the main results of [5] in more details. They proved the convergence of the gradient flow of one-hidden layer ReLU Neural Networks to zero loss with specific dynamics for small initialization under strong assumptions, such as orthogonal input data. We briefly add some results of [34], which describes the training dynamics for the same network structure, but for more general training dataset.

## 2.1 Training of ANN : Partial State-of-The-Art

Now, we mention some results in the literature about the training of neural networks and some useful tools used to study the dynamics of the training process. In [38], the authors made an overview of Optimization for Deep Learning. It covers superficially main interesting results obtained until 2019. In the following, we just summarize some results we are interested in, and we use it as base for the state-of-the-art review. All results coming from another article will be clearly cited.

First, it is important to identify design choices that are crucial for achieving good generalization performance on unseen data with neural network models. There are four kinds of design choices presented in [38] :

- Data processing : that regroups all methods to make the data usable for the model (e.g. data augmentation, features selection, normalization of the dataset, etc).
- Optimization methods : optimization algorithms, learning rate schedule, initialization, etc.
- Regularization : methods that add a term to the objective function to penalize high complexity in the model (e.g., $L_1$ and $L_2$ regularization). The objective is to avoid overfitting, which occurs when the model fits the training data perfectly (resulting in low loss on the training data), but generalizes poorly to unseen data (resulting in high loss on test data). Overfitting is one of the biggest challenges in Machine Learning.
- Neural network architecture : this includes all the choices regarding the network structure (e.g. depth, width, connections patterns, type of layers, activation functions, etc).

In this work, we focus on optimization methods, particularly on basic neural network architectures such as shallow fully-connected neural network when it is possible.

### 2.1.1 Explosion and Vanishing gradient problem

They begin by discussing one of the biggest problems when training a Neural Network with a gradient-based optimization algorithm: the exploding/vanishing gradient problem. This issue often occurs when there are many layers in the network. During backward propagation, we compute the partial derivatives of the loss with respect to all weights in each layer. These partial derivatives are applied as corrections to the weights when we take a step of GD. Each layer can amplify or attenuate the corrections made by the GD for the previous[1] layers, and with many layers, the corrections can either explode or vanish.

The main problem of the explosion/vanishing of the gradient is that GD does not converge in polynomial time due to the large condition number of the Hessian matrices of the loss on the training path. Indeed, the condition number of the Hessian often determines the speed of convergence of the GD. They explain that, in the case of explosion/vanishing of the gradient, the diagonal entries of the Hessian matrices can be very large or very small, and these entries correspond to local per-entry Lipschitz constants of the gradient. Consequently, the diagonal entries of the Hessian matrices can exhibit a wide dynamic range[2], potentially leading to exponentially large condition numbers.

The explosion/vanishing gradient problem also complicates selecting an appropriate step-size for GD. Computing the local Lipschitz constant of the gradient of the loss function is generally computationally expensive. Therefore, in practice, a constant step-size is often chosen. However, if the local Lipschitz constant varies significantly along the training path, the optimization algorithm can be significantly slowed down in regions where the chosen step-size is much smaller than the theoretical step-size corresponding to this constant.

They provide a simple example illustrating why the explosion/vanishing of the gradient is problematic (cfr. Figure 2.1). This function exhibits three distinct regions affecting the convergence rate of Gradient Descent. Between $-0.8$ and $0.8$, the gradient is very low and almost vanishes, resulting in a very slow convergence rate. Prior to $-0.8$ and beyond $1.2$, the gradient is very high, and explodes when we deviate from these values. This poses a problem if the chosen constant step-size is too large, as it may cause the gradient step to overshoot the minimum. Between $0.8$ and $1.2$, we are in a "good basin", leading to a good convergence rate towards the minimum.

In order to avoid explosion/vanishing gradient problem, they identify three tricks :

- Initialization,
- Normalization,
- Changing the structure of the Neural-Net.

Since Normalization and Changing the neural structure alter the network, we skip them and focus on initialization methods. The goal of a good initialization method is to find an initial point for the optimization method that is in a good basin, facilitating a fast convergence to a good minimum. Of course, it is not an easy task since good basins are a priori unknown.

[1] Here, "previous" is used in the context of the backward propagation. So, if we consider the layer $l$, the previous layer is the layer $l+1$.

[2] i.e. the ratio between the largest and the smallest values of the diagonal entries.

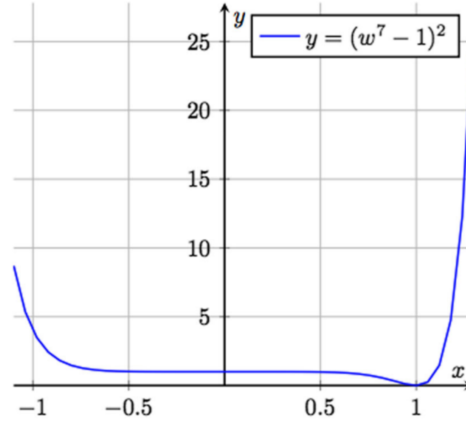



FIGURE 2.1: Illustration of "good basin" ($w \in [0.8, 1.2]$), almost vanishing gradient ($w \in [-0.8, 0.8]$), and region where gradient explodes ($w \in ]-\infty, -0.8] \cup [1.2, +\infty[$). (Figure from [38])

### 2.1.2 Initialization methods

In order to simplify the notations, we will omit the super-script or the index corresponding to $t = 0$ in this section when it is clear. For example, $\theta^0$ and $\left(w_{i,j}^{(l)}\right)_{t=0}$ becomes $\theta$ and $w_{i,j}^{(l)}$ respectively in such situations. We recall that $D$ denotes the total number of weights in the network, and $m^{(l)}$ corresponds to the number of neurons in the layer $l$.

The first class of initialization methods is data-independent, meaning that we do not need to use the dataset to initialize the weights $\theta^0$ of the ANN.

- **Naive Initialization** : Choose the zero point, a sparse initial point, or a point from a random distribution. The drawback of these methods is that it is difficult to make them work efficiently for a large class of problems.
- Random Initialization with Specific Variance :
  - **Bottou Initialization** : For ANN with sigmoid activation function, choose $\theta^0$ such that :

$$\mathbb{E}\left[w_{i,j}^{(l)}\right] = 0, \quad \text{Var}\left(w_{i,j}^{(l)}\right) = \frac{1}{m^{(l-1)}} \tag{2.1}$$

    The advantage is that it is easy to tune for a specific network, but it is not easily transferable to another one.
  - **Xavier Initialization** : For ANN with sigmoid activation function, choose $\theta^0$ such that :

$$\mathbb{E}\left[w_{i,j}^{(l)}\right] = 0, \quad \text{Var}\left(w_{i,j}^{(l)}\right) = \frac{2}{m^{(l-1)} + m^{(l)}} \tag{2.2}$$

  - **Kaiming Initialization** : It is a modified version of Xavier initialization for ANN with ReLU activation function :

$$\mathbb{E}\left[w_{i,j}^{(l)}\right] = 0, \quad \text{Var}\left(w_{i,j}^{(l)}\right) = \frac{2}{m^{(l-1)}} \text{ or } \frac{2}{m^{(l)}} \tag{2.3}$$

- **Orthogonal Initialization** : It consists to initialize each layer weight matrix with a random orthogonal matrix, i.e. $\left(W^{(l)}\right)^T W^{(l)} = I_{m^{(l-1)}}$. A possible process to get these orthogonal matrices is the following :
  1. initialize all the weights $w_{i,j}^{(l)}$ with a random normal initialization (of zero mean and unit variance),
  2. for each layer weight matrix $W^{(l)}$, perform a QR (or SVD) matrix decomposition
  $$W^{(l)} = Q^{(l)}R^{(l)}, \tag{2.4}$$
  3. Replace $W^{(l)}$ by the orthogonal matrix $Q^{(l)}$.

A second type of initialization methods is data-dependent.

- **Layer-Sequential Unit-Variance Initialization (LSUV)** : It consists in two steps :
  1. Find an initial point $\theta^0$ using orthogonal initialization (see above),
  2. For each mini-batch, normalize the variance of the output of each layer to be 1 by directly scaling the weight matrices.
- **Meta-initialization** : Dauphin and Schoenholz [12] introduced the Gradient Quotient as a measure of the local linearity of a function's region, avoiding the costly computation of the complete Hessian. This measure is the Gradient Quotient :
  $$\text{GQ}(\mathcal{L},\theta) = \frac{1}{D}\left\|\frac{\nabla\mathcal{L}(\theta) - \nabla^2\mathcal{L}(\theta)\nabla\mathcal{L}(\theta)}{\nabla\mathcal{L}(\theta)+\epsilon} - 1\right\|_1 \approx \frac{1}{D}\left\|\frac{\nabla\mathcal{L}(\theta - \nabla\mathcal{L}(\theta))}{\nabla\mathcal{L}(\theta)+\epsilon} - 1\right\|_1$$
  with $\epsilon = \epsilon_0(2 \times \mathbb{1}_{\nabla\mathcal{L}(\theta)\geq 0} - 1)$, and with $\mathbb{1}_{\nabla\mathcal{L}(\theta)\geq 0} = 1$ if $\nabla\mathcal{L}(\theta) \geq 0$ and 0 otherwise. The goal is to find a region with minimal second-order effects. In such regions, the direction and magnitude of the gradient descent should not change drastically between iterations due to these second-order effects, making the gradient descent more efficient. To find $\theta^0$ :
  1. find a guess $\tilde{\theta}^0$ with a random normal initialization or an orthogonal initialization,
  2. compute $\theta^0$ by solving
  $$\theta^0 = \text{MetaInit}(\mathcal{L},\theta) = \arg\min_\theta \text{GQ}(\mathcal{L},\theta)$$
  with the Gradient Method from $\tilde{\theta}^0$.

### 2.1.3 Optimization algorithm used in Deep Learning

They also provide a summary of the principal optimization algorithms used in Deep Learning.

1. **Stochastic Gradient Descent** : we consider a unconstrained minimization problem as follow
   $$\min_\theta F(\theta) \quad \text{with } F(\theta) = \frac{1}{n}\sum_{k=1}^{n} F_k(\theta) \tag{2.5}$$
   Because the complete gradient $\nabla F(\theta) = \frac{1}{n}\sum_{k=1}^{n}\nabla F_k(\theta)$ can be costly to compute, we can only compute it for a subset of indices $\mathcal{K}_j = \{n_b(j-1)+1,\cdots,n_bj\} \subset \{1,\cdots,n\}$.

In Deep Learning, it is equivalent to partition the dataset into several batches $B_{\mathcal{K}_j} = \{(x_k, y_k)\}_{k\in\mathcal{K}_j}$ of same size. The SGD computes iteratively a new point $\theta^{t+1}$ such that

$$\theta^{t+1} = \theta^t - \alpha_t \nabla F_{\mathcal{K}_j}\left(\theta^t\right) \tag{2.6}$$

with $j$ chosen randomly and $\nabla F_{\mathcal{K}_j}(\theta) = \frac{1}{|\mathcal{K}_j|}\sum_{k\in\mathcal{K}_j}\nabla F_k(\theta)$. In this setting, a batch $j$ is always the same, but we can shuffle the indices $k$ of the data at each iteration in order to get a random batch.

We also need a step-size schedule, i.e. a value for $\alpha_t$ for each epoch $t$. The ones listed in the article [38] are :

- Constant step-size : $\alpha_t = \alpha$ for each $t$.
- Simple digressive step-size : we divide the step-size by a constant every few iterations, or if iterates are stuck.
- Step-size warmup : it is a type of schedule where we use a small step-size for fixed number of epochs (the warmup), and after that, we use a higher constant value for the rest of the process.
- Cyclical step-size : there are different types of cyclical schedules. Their common feature is that the step-size varies between a lower and an upper bounds in a cyclical manner. We can cite three variations : SGDR, the Ioshchilov et al. variant, and the restart variant.

You can find a summary of classical convergence analysis for SGD in [20]. The issues with the classical analysis are that it assumes the existence of a Lipschitz continuous gradient of the objective function and requires a decreasing step-size over epochs. There are also some results with a constant step-size, but convergence to a point where the gradient is zero is not reached. Indeed, due to the step-size and the randomness of SGD, iterates jump around the stationary point within a certain confusion zone.

More recently, for Neural Networks, it has been proved that SGD with a constant step-size converges if the Network can represent the true underlying function from which the data have been drawn, i.e. the global minimum of the loss function is zero. In this case, there is an "automatic variance reduction effect" that leads to the convergence of SGD with a constant step-size.

From a practical point of view, it is also notable that SGD converges faster than classical GD, with an acceleration ratio depending on the problem.

2. **SGD with momentum** :

$$\begin{aligned} m_t &= \beta\, m_{t-1} + (1-\beta)\nabla F_i\left(\theta^t\right) \\ \theta^{t+1} &= \theta^t - \alpha_t m_t \end{aligned}$$

In practice, SGD with momentum appears to be faster than classical SGD in the field of Machine Learning. For convex problems, there are results showing that GD with momentum is indeed faster than classical GD. However, this is not the case for SGD, as some convex problems have been found where the convergence rate of SGD is not improved by using momentum. For nonconvex problems, these results no longer hold, and the acceleration of convergence is not guaranteed. In both cases, [38] lists some

more complex methods to achieve a better convergence rate using SGD with momentum. However, these methods do not seem to be widely used in practice due to their specific requirements.

3. **Adaptive Gradient Methods** : Let us define $\bullet$ as the component-wise product operator, i.e. for $a, b \in \mathbb{R}^d$,

$$a \bullet b = \begin{bmatrix} a_1 & 0 & \cdots & 0 \\ 0 & a_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_d \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_d \end{bmatrix} = \begin{bmatrix} a_1 b_1 \\ a_2 b_2 \\ \vdots \\ a_d b_d \end{bmatrix},$$

and $a^{-\frac{1}{2}}$ as the component-wise operator such that $a_i^{-\frac{1}{2}}$ for each element $i$ of the vector $a$.

The purpose of adaptive methods is to assign a different step-size for each component of $\theta$. This adaptive step-size is chosen in an automatic way to account for the different scales at which the landscape of the objective function changes for each coordinate of $\theta$. The motivation behind these methods is to solve problems with sparse and highly unbalanced data more efficiently.
The most popular adaptive methods are listed just below.

(a) AdaGrad :

$$g_t = \nabla F_i \left(\theta^t\right) \tag{2.7}$$

$$v_t = \sum_{j=1}^{t} g_j \bullet g_j \tag{2.8}$$

$$\theta^{t+1} = \theta^t - \alpha_t v_t^{-\frac{1}{2}} \bullet g_t \tag{2.9}$$

(b) RMSProp :

$$g_t = \nabla F_i \left(\theta^t\right) \tag{2.10}$$

$$v_t = \beta\, v_{t-1} + (1-\beta) g_t \bullet g_t \tag{2.11}$$

$$\theta^{t+1} = \theta^t - \alpha_t v_t^{-\frac{1}{2}} \bullet g_t \tag{2.12}$$

(c) Adam :

$$g_t = \nabla F_i \left(\theta^t\right) \tag{2.13}$$

$$m_t = \beta_1\, m_{t-1} + (1-\beta_1) g_t \tag{2.14}$$

$$v_t = \beta_2\, v_{t-1} + (1-\beta_2) g_t \bullet g_t \tag{2.15}$$

$$\theta^{t+1} = \theta^t - \alpha_t v_t^{-\frac{1}{2}} \bullet m_t \tag{2.16}$$

(d) AMSGrad :

$$g_t = \nabla F_i\left(\theta^t\right) \tag{2.17}$$
$$m_t = \beta_1\, m_{t-1} + (1-\beta_1)g_t \tag{2.18}$$
$$\bar{v}_t = \beta_2\, \bar{v}_{t-1} + (1-\beta_2)g_t \bullet g_t \tag{2.19}$$
$$v_t = \max\{v_{t-1}, \bar{v}_t\} \tag{2.20}$$
$$\theta^{t+1} = \theta^t - \alpha_t v_t^{-\frac{1}{2}} \bullet m_t \tag{2.21}$$

In short, the difference between adaptive methods is as follows. AdaGrad assigns equal importance to all past gradients. RMSProp decreases the importance of older gradients exponentially. Adam combines RMSProp with momentum methods. Empirically, Adam seems to converge faster than SGD (with or without momentum), but it tends to achieve worse generalization performance. Theoretically, Adam (and RMSProp) can diverge, even for some convex problems. AMSGrad is a correction of Adam to ensure the convergence for convex problems.
For nonconvex problems, there exist some convergence results for adaptive gradient methods under certain assumptions[3], but these methods are still widely misunderstood.

4. **Higher Order Methods** : They also mention research on how to apply second-order methods or their Hessian-free versions, such as the Newton method and quasi-Newton methods, in deep learning.

### 2.1.4 Landscape analysis of the Loss

Until now, we have cited results about local issues and characteristics of the training process of Neural Networks. While we have guarantees about the convergence to stationary points, we do not know if these points are global minima. In [38], they also discuss a more global perspective by summarizing works on the convergence to a global minimum and the quality of this minimum through the characterization of the loss function landscape. They refer to this field as "Global Optimization of Neural Networks" (GON).

Let us recall that the total number of parameters is denoted $D$. Therefore, the surface of the loss (or the landscape) $(\theta, \mathcal{L}(\theta))$ is in $\mathbb{R}^{D+1}$. Since $D$ is generally very large, a complete visualization or characterization of this surface is impossible.

They first mention some empirical results. In [13], it is shown that as $D$ increases, the proliferation of bad local minima (i.e. with a high value compared to the global minimum) is slower than the proliferation of saddle points for one-hidden layer fully-connected neural networks. In high-dimensional space, these saddle points are numerous and are surrounded by wide plateaus. Moreover, local minima with high error seem to be exponentially rare. The biggest challenge for an algorithm like (S)GD is then to traverse these plateaus, as the convergence can significantly slow down in these areas. This slowdown can be so dramatic that it may give the impression that the algorithm has fallen into a bad local minimum.
However, in [21], they experimentally show that optimization algorithms training some common ANNs (Fully-Connected Neural Networks in supervised learning, CNNs, Deep Linear Neural Networks) rarely encounter obstacles such as saddle points or bad local minima on the training path.

[3] See [3] for an example of such results.

In [31], they study the structure of the loss landscape and the effect of choosing certain hyperparameters (such as learning rate, batch size, optimization algorithm, etc.) on the shape of the loss landscape using several visualization methods. Based on this, they characterize the landscape around a minimizer and its resulting ability to generalize to unseen data. They mention that the landscape becomes smoother as the width of the network increases.

The authors of [15] find empirically that two global minima can be connected by an almost equal-value path. For one-hidden layer fully-connected neural networks, they cite results showing that there are some high-loss value barriers between minima for three or fewer neurons in the hidden layer, and these barriers disappear for more neurons. They experimentally show the same phenomenon for networks with convolutional layers.

There is a link between the landscape of the loss and the ability of a trained network to generalize to unseen data. It has been conjectured that flat and wide minima generalize better than sharp minima. We can get the intuition of this conjecture from the example in Figure 2.2. It seems that wide minima are more robust with respect to errors in the testing loss function. The difference between the training and testing loss functions affects sharp minima more than wide minima, resulting in better generalization ability. Some articles (e.g. [22] and [26]) provide numerical experiments that support this conjecture.

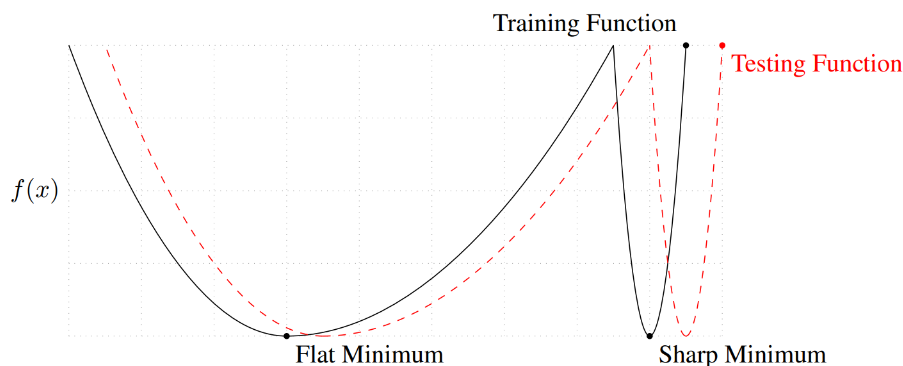



FIGURE 2.2: Illustration of sharp and wide minima, and their ability to generalize. (Figure from [38])

Moreover, in [31], they also discuss the link between the sharpness of a minimizer and its ability to generalize. They cite some results indicating that, in certain settings, sharp minimizers can generalize well, suggesting that the generalization ability of a minimizer is not directly related to the curvature of the loss surface. Subsequently, they show empirically that the sharpness of minimizers correlates well with generalization error when normalized visualization is used to remove scaling effects, but not when non-normalized visualization is used.

In [38], they list two main types of deep neural networks for which positive results exist so far :

1. **Deep Linear Network** : This type of networks is very interesting from a theoretical point of view, but it is not very powerful in practice since it has a little representation power. In [13], it is shown that studying deep linear neural networks can be useful since their training dynamics resemble those of ANNs with nonlinear activation functions in some aspects. Although the prediction outputs of deep linear ANNs are linear in the input data, the model is nonlinear with respect to the weights $\theta$. Finding the optimal weights $\theta^*$ is a nonconvex problem, and the training dynamics have similarities

to those for networks with nonlinear activation functions. Moreover, the loss landscape of such networks presents many saddle points, but does not contain any bad local minima since all minima are global minima of the loss. These global minima are linked to each other in a continuous manifold. This statement has been proven in articles cited in [38] under mild conditions (See [25], [33], [27], and [36]). They also find necessary and sufficient conditions for a stationary point to be a global minimum.

2. **Deep Over-parameterized Network** : An over-parameterized network is a network where the number of tunable weights/parameters is strictly greater than the number of constraints. In Supervised Learning, this means the number of weights is greater than the number of training data, i.e. $D > n$.

   There is a common belief that if a network has more parameters than necessary (thus being over-parameterized), the more parameters there are, the smoother the landscape will be. There is no rigorous proof of this idea, but there exist some numerical experiments that support this view.

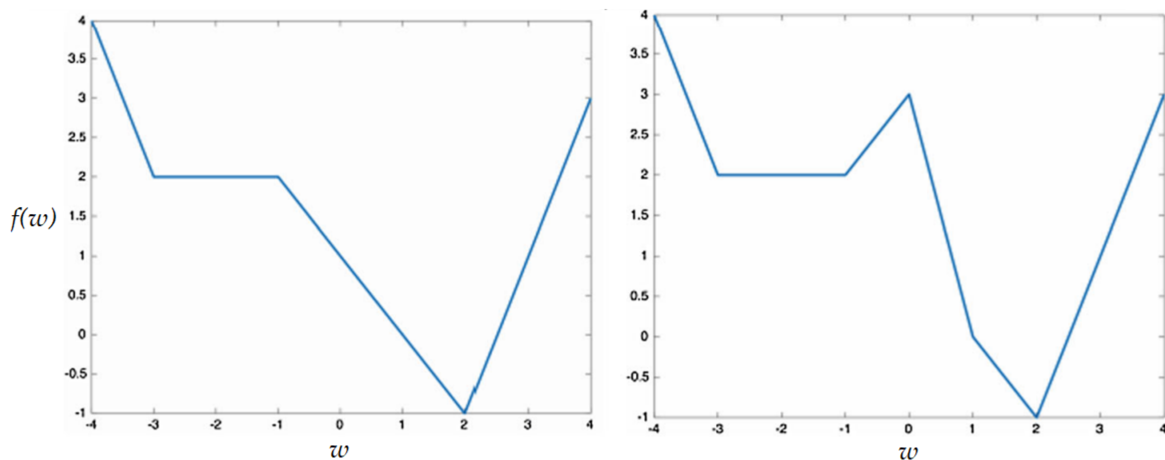
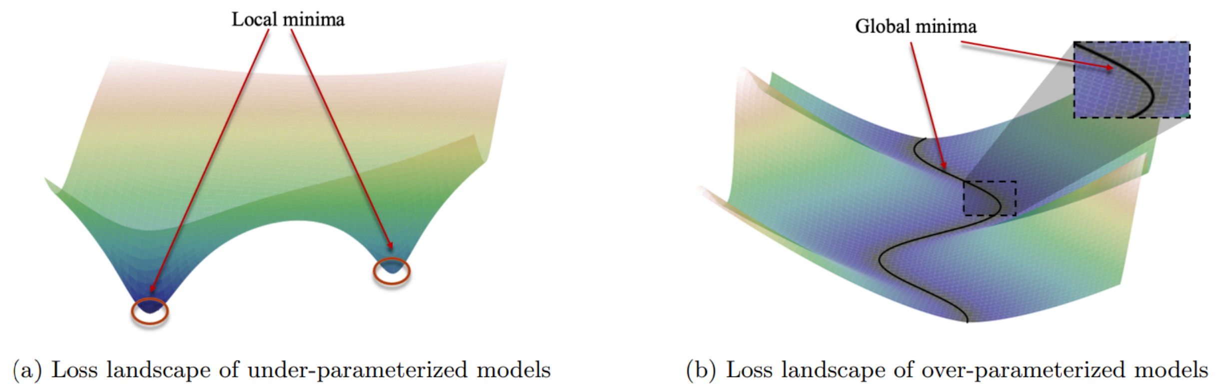



FIGURE 2.3: Illustration of the notion of sub-optimal basin. (Left) Function without sub-optimal basin. (Right) Function with sub-optimal basin ($w \in [-3, -1]$) with a barrier in the direction of the global minimum. (Figure from [30])

In [30], the authors define the notion of a "sub-optimal basin." A sub-optimal basin is a region of the loss landscape containing "set-wise local strict minima" (see Figure 2.3). Roughly speaking, a set-wise local strict minimum is a compact subset of local minima surrounded by barriers of greater value. If the loss function has no sub-optimal basins, then a continuous path exists from any initial point to a global minimum. They also show a transition phase between the loss landscapes of narrow and wide networks. Specifically, they prove that sub-optimal strict local minima can exist for narrow networks, while wide networks have no sub-optimal basins.
For fully-connected neural networks, if the last hidden layer has more neurons than the number of samples (under other mild assumptions), it has been proven that the loss landscape has no sub-optimal basins for generic input data. These results align with [32], which argues that sufficiently over-parameterized neural networks have no strict local minima under mild assumptions, and each local minimum is path-connected to a global minimum, with a non-increasing loss value along the path.

In [38], they claim that over-parameterization can only avoid sub-optimal basins without other assumptions. This is supported by [14], where the authors construct sub-optimal minima for arbitrarily wide neural networks with a large class of activation functions[4]. Thus, sub-optimal minima (as non-strict local minima) can exist in the previously mentioned setting. Additionally, [14] reports that for a one-hidden layer Sigmoid neural network, if the number of neurons in the hidden layer exceeds the number of data points, every local minimum of the square loss is a global minimum.




(a) Loss landscape of under-parameterized models (b) Loss landscape of over-parameterized models

FIGURE 2.4: Illustration of the loss landscape of under-parameterized and over-parameterized networks around minima. (Figure from [32])

In [32], the authors examine the loss landscape of over-parameterized neural networks. They note that, unlike under-parameterized networks, the loss landscape of over-parameterized networks is generally highly nonconvex, even locally. This key difference means that around the minima, the under-parameterized landscape is typically locally convex, allowing classical convergence analysis. Conversely, the over-parameterized landscape remains nonconvex around global minima, even in arbitrarily small neighborhoods. Therefore, they conclude that convexity-based analysis is inadequate for over-parameterized systems. Indeed, the set of global minima in such networks forms a positive-dimensional manifold (typically $D-n$) with no isolated global minimum, thus maintaining nonconvexity locally[5] (See Figure 2.4). For the square loss, they give the following proposition.

**Proposition 3.** *Let $\mathcal{L}$ be the square loss and $h_\theta$ an over-parameterized model. For $\mathcal{L}(\theta^*) = 0$ and, furthermore, assume that $\nabla_\theta h(\theta^*) \neq 0$ and Rank $\left(\nabla^2 h_{x_k}(\theta^*)\right) > 2n$ for at least one $k \in \{1, \cdots, n\}$. Then $\mathcal{L}(\theta)$ is not convex in any neighborhood of $\theta^*$.*

A more useful framework for this kind of highly nonconvex landscape is the Polyak-Łojasiewicz (PL) condition. We will discuss this notion in more detail in Chapter 4. In short, in [32], the authors show that the loss landscape satisfies the PL* condition (a variant of the Polyak-Łojasiewicz condition) over most of the parameter space. This condition guarantees the existence of solutions and efficient optimization by (S)GD. The last important thing to note is that these results hold for neural networks trained in the lazy regime. In this regime, due to the high level of over-parameterization, each weight $w_{i,j}^{(l)}$ moves only slightly during training, more precisely, for $m$ being the

[4]And this is not an isolated result since [38] provides about six other references in which authors construct sub-optimal local minima for ReLU and smooth activation functions.
[5]Unless the landscape is linear, but it is extremely rare.

smallest number of neurons in a hidden layer (i.e. $m = \min_{l \in \{1,\cdots,L\}} m^{(l)}$),

$$\left| w_{i,j}^{(l)} - (w_{i,j}^{(l)})_{t=0} \right| = \mathcal{O}\left(\frac{1}{\sqrt{m}}\right).$$

### 2.1.5 Convergence analysis results for Deep Networks

Here, we present some convergence analysis results for deep networks. We start with linear networks and then introduce the concept of the Neural Tangent Kernel. Following that, we discuss important concepts frequently used throughout our work: implicit bias and training regimes. Finally, we briefly discuss the automatic reduction of a network's internal representation during training.

- **Linear Network** : In [1], they analyze the convergence rate of the Gradient Method for deep linear neural networks minimizing the square loss for whitened data[6]. They make three assumptions to ensure convergence to a global minimum at linear rate :

  1. Widths of hidden layers are greater than or equal to the minimum between the dimensions of input and output, i.e. $m^{(l)} \geq \min\{d, m^{(L+1)}\}\ \forall l \in \{1, \cdots, L\}$;
  2. Weights $\theta^0$ at initialization are $\delta$-balanced :

     $$\left\| \left(W^{(l+1)}\right)^T W^{(l+1)} - \left(W^{(l)}\right)^T W^{(l)} \right\|_F \leq \delta$$

     for all $l \in \{1, \cdots, L\}$;
  3. The initial loss is smaller than the loss of any rank-deficient solution (solution with some $W^{(l)}$ which is low rank matrices).

  They also state that the two last assumptions on initialization are necessary for the convergence to the global minimum, and these conditions are satisfied for scalar regression ($m^{(L+1)} = 1$). Based on the concept of $\delta$-balanced initialization and rank-deficient solution, they highlight a trade-off between slow but guaranteed convergence and fast convergence with a high risk of divergence. When the linear network is initialized with random normal initialization with zero mean and fixed variance, a small variance allows meeting the two initialization requirements with high probability but results in a slow convergence rate. When the variance is higher, these two requirements are less likely to be met, but if they are, the convergence is faster. Furthermore, the optimization path of GD maintains the weight balancedness property over epochs.

  In [24], the authors prove convergence of the gradient method and gradient flow to zero loss for a deep linear neural network on linearly separable data. Moreover, they show an asymptotic weight matrix alignment. For each weight matrix $W^{(l)}$, its rank-1 approximation becomes asymptotically the only contribution to the final model, and the rank-1 weight matrix approximations of two adjacent layers align. This phenomenon leads to a minimum norm solution $\theta^*$.

- **Neural Tangent Kernel (NTK) and Linearization** : Considering the neural network problem with the square loss function

  $$\min_{\theta \in \mathbb{R}^D} \frac{1}{2n} \sum_{k=1}^{n} \left(h_\theta(x_k) - y_k\right)^2, \tag{2.22}$$

[6] i.e. data is transformed such that $\frac{1}{n} X X^T = I_d$.

we can define the gradient flow as follows

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = -\nabla\mathcal{L}(\theta) = -\frac{1}{n}\sum_{k=1}^{n} \nabla_\theta h_\theta(x_k)\left(h_\theta(x_k) - y_k\right). \tag{2.23}$$

The Neural Tangent Kernel (NTK) is then

$$K = G^T G \tag{2.24}$$

with $G = \begin{bmatrix}\nabla_\theta h_\theta(x_1) & \cdots & \nabla_\theta h_\theta(x_n)\end{bmatrix} \in \mathbb{R}^{D\times n}$. Defining the residuals vector $r = \begin{bmatrix}h_\theta(x_1) - y_1 & \cdots & h_\theta(x_n) - y_n\end{bmatrix}^T$, we have

$$\frac{\mathrm{d}\theta}{\mathrm{d}t} = -Gr \tag{2.25}$$

$$\Longrightarrow G^T\frac{\mathrm{d}\theta}{\mathrm{d}t} = -G^T Gr \tag{2.26}$$

$$\Longrightarrow \frac{\mathrm{d}r}{\mathrm{d}t} = -K(t)r. \tag{2.27}$$

So, the gradient flow can be studied from the last expression with the NTK. Under certain conditions, it can be proved that $K(t)$ is a constant matrix over $t$, and equation 2.27 reduces to a simple system of Ordinary Differential Equations. For example, when $h_\theta(x) = \theta^T x$, then $K = X^T X$.

In some situations, it can be interesting to replace the initial network $h_\theta(x)$ by its linearized version around $\theta^0$ :

$$h_x^{\mathrm{lin}}(\theta) = h_x(\theta^0) + \theta^T\nabla_\theta h_x(\theta^0) \tag{2.28}$$

which gives interesting final predictor models in some situations.

- **Implicit bias (Lazy training and rich training)** : First, we can define the notion of implicit bias of an algorithm. We will need it at the end of this chapter and in Chapter 3 when we discuss the influence of the initialization scale for neural networks. Roughly speaking, the **implicit bias** of an optimization algorithm is its tendency to find a certain kind of minimizers (i.e., with specific properties) depending on parameters such as initialization and learning rate.

  When training an ANN model, we can identify two different regimes related to the initialization of the model, leading to two different implicit biases : the *Lazy Regime*, and the *Rich Regime*[7].

  The Lazy Regime occurs when the model behaves like its linearization around its initial point. Because of the large number of weights in the model, each weight only needs to be updated slightly to find a minimizer. Thus, the linearized network is a good approximation of the true network. The Lazy Regime often occurs when the scale of the initialization is high and exceeds a certain threshold. The training dynamics are similar to those of a convex function, resulting in a fast convergence to a minimizer (linear convergence), but this minimizer generally has poor generalization ability.
  In [6], they provide a simple criterion for Lazy Regime based on the notion of "inverse

[7] Also called Lazy and Rich Training

relative scale" of the model $h_\theta$ at $\theta^0$ ($\kappa_h(\theta^0)$) when using square loss function :

$$\kappa_h(\theta^0) = \|h_{\theta^0} - y\| \frac{\|D^2 h_{\theta^0}\|}{\|D h_{\theta^0}\|^2} \ll 1. \tag{2.29}$$

$\kappa_h(\theta^0)$ measures how much the training dynamics differ from the linearized training dynamics. For one-hidden layer neural networks with large width $m$, we have the following inequality when the network is randomly initialized with i.i.d. weights such that $\mathbb{E}\left[a_j \sigma\left(\sum_{i=1}^d w_{j,i} x_i\right)\right] = 0$, and is used with a smooth activation function :

$$\mathbb{E}\left[\kappa_h(\theta^0)\right] \lesssim m^{-\frac{1}{2}} + (m\alpha(m))^{-1} \tag{2.30}$$

where $\alpha(m) > 0$ is a scaling factor introduced in the model such that

$$h_\theta(x) = \alpha(m) \sum_{j=1}^m a_j \sigma\left(\sum_{i=1}^d w_{j,i} x_i\right).$$

It implies that the scaling factor plays a significant role in the occurrence of Lazy Regime. When $m\alpha(m) \xrightarrow{m \to +\infty} +\infty$, the model will surely reach the Lazy Regime for sufficiently large values of $m$.

On the other hand, the Rich Regime occurs when the scale of initialization is small. This regime is generally more interesting since we can observe feature learning in this regime. The training dynamics are very nonconvex and slower than those in the Lazy Regime, but the minimizer is generally more powerful at generalizing to unseen data. Of course, there is a transition phase between the two regimes. We will explore these notions in the Chapter 3 in the presentation of our numerical experiments on one-hidden layer ANNs.

- **Automatic Reduction of the Internal Representation of Networks** : In [34], the authors refer to results from other studies discussing the presence of two distinct phases in the training dynamics of ANNs using (S)GD. These phases indicate that, initially, the network fits the data, and subsequently, the internal representation is compressed (i.e., the information in the hidden layers about the input is reduced). However, in their own work, they observe the opposite behavior. First, the model compresses its representation by aligning neurons, and then it fits the data. We will go into the details of this alignment phenomenon in section 2.2.

### 2.1.6 Results for Shallow Networks

It is well-known that one-hidden layer neural networks can approximate any continuous function with a finite number of neurons, thanks to the *Universal Approximation Theorem*. Unfortunately, this theorem does not provide any guidance on how to determine the structure and weights of the network to achieve this best approximation. Here, we list some results from the literature on one-hidden layer neural networks from both a landscape and an algorithmic point of view.

- **Global Landscape of one-hidden layer Neural-Nets** : Results for one-hidden layer neural networks seem to concentrate on the characterization and existence of bad local minima and bad basins. For example, for ReLU activation functions and ultra-large width, there are no bad basins in the landscape (See [19]). In [41], they find a necessary and sufficient condition for the non-existence of bad basins using the notion of

"intrinsic dimension." For a one-hidden layer ReLU neural network, all local minima are non-differentiable (except for flat bad local minima) (See [28]).

- **Algorithmic Analysis of one-hidden layer Neural-Nets** : This kind of analysis is generally conducted for very specific cases of training algorithms or specific data. For example, in [40], they show that minimizing the square loss and using spherical input data, GD with random initialization converges to a minimizer corresponding to the best degree $k$ polynomial approximation of the target function. They also provide an upper bound for the required width of the hidden layer and the number of iterations needed to achieve this result: $n^{\mathcal{O}(k)}\log(\frac{1}{\epsilon})$. Furthermore, they argue that to obtain these results, it is sufficient to train the output weights while keeping the hidden weights constant.

In [37], the authors characterize the convergence of first-order optimization methods for over-parameterized models when the loss function has certain properties over a sufficiently small neighborhood of the initial point. We only summarize the results for GD, but the authors also provide similar results for SGD. Denoting the Jacobian of the model $h$ by $\mathcal{J}_h \in \mathbb{R}^{n\times D}$, they make the following two assumptions:

**Assumption 1.** ***(Jacobian Local Lispchitz continuity)*** *Consider a set $\mathcal{D} \subset \mathbb{R}^D$ such that $\theta^0 \in \mathcal{D}$. We assume that $\forall\theta^1, \theta^2 \in \mathcal{D}$*

$$\|\mathcal{J}_h(\theta^2) - \mathcal{J}_h(\theta^1)\| \leq L_{\mathcal{J}_h}\|\theta^2 - \theta^1\|_2. \tag{2.31}$$

**Assumption 2.** ***(Jacobian spectrum)*** *Consider a set $\mathcal{D} \subset \mathbb{R}^D$ such that $\theta^0 \in \mathcal{D}$. We assume that $\forall\theta \in \mathcal{D}$ the following inequality holds*

$$a \leq \sigma_{\min}(\mathcal{J}_h(\theta)) \leq \|\mathcal{J}_h(\theta)\| \leq b \tag{2.32}$$

*with $0 < a \leq b$.*

Under these assumptions, they prove the following theorem.

**Theorem 3.** *Consider a nonlinear least-squares optimization problem. Suppose the Jacobian mapping associated with $h_\theta$ obeys Assumption 2 over a ball $\mathcal{D}$ of radius $R = \frac{4\|h_x(\theta^0)-y\|_2}{a}$ around a point $\theta^0 \in \mathbb{R}^D$. Furthermore, suppose Assumption 1 holds over $\mathcal{D}$ and set $\alpha \leq \frac{1}{2b^2}\min\left\{1, \frac{a^2}{L_{\mathcal{J}_h}\|h_x(\theta^0)-y\|_2}\right\}$.*
*Then, running gradient descent $\theta^{t+1} = \theta^t - \alpha\nabla\mathcal{L}(\theta^t)$ starting from $\theta^0$, all iterates obey :*

$$\|h_x(\theta^t) - y\|_2^2 \leq \left(1 - \frac{\alpha a^2}{2}\right)^t \|h_x(\theta^0) - y\|_2^2, \tag{2.33}$$

$$\frac{a}{4}\|\theta^t - \theta^0\|_2 + \|h_x(\theta^t) - y\|_2 \leq \|h_x(\theta^0 - y\|_2, \tag{2.34}$$

$$\sum_{t=0}^{+\infty}\|\theta^{t+1} - \theta^t\|_2 \leq \frac{4\|h_x(\theta^0) - y\|_2}{a}. \tag{2.35}$$

**Corollary 4.** *Consider the setting and assumptions of Theorem 3. Let $\theta^*$ denote the global optima of the loss $\mathcal{L}(\theta)$ with the smallest euclidean distance to the initial point $\theta^0$. Then, the gradient iterates $\theta^t$ obey*

$$\|\theta^t - \theta^0\|_2 \leq \frac{4b}{a}\|\theta^* - \theta^0\|_2, \tag{2.36}$$

$$\sum_{t=0}^{+\infty} \|\theta^{t+1} - \theta^t\|_2 \leq \frac{4b}{a} \|\theta^* - \theta^0\|_2. \tag{2.37}$$

From Theorem 3 and Corollary 4, they prove that GD has the following properties :

1. The iterates converge at a linear rate to a global minimum even when the loss is nonconvex (cfr. equation 2.33),
2. Among all global minima, the iterates converge to one with a near-minimal distance to the initial point, as shown by equation 2.36. Moreover, the GD iterates never leave a neighborhood of radius $\frac{4}{a}\|h_x(\theta^0) - y\|_2$ around the initial point $\theta^0$;
3. Equation 2.37 implies that the iterates take a near-direct path from the initial point to this global minimum, as the total path length is within a factor of the distance between the closest global minimum and the initial point.

Still in [37], they characterize the role of the number of data in the over-parameterized loss landscape. They make two observations :

1. Adding more data to the dataset leads to a higher condition number of $\mathcal{J}_h(\theta)$, resulting in larger $b$ and smaller $a$.
2. $\|h_x(\theta^0) - y\|_0 \sim \sqrt{n}$ when data are drawn from a random distribution and are i.i.d..

With Theorem 3, we conclude that more data leads to a more difficult optimization problem. Because the Jacobian condition number increases, the convergence rate is slower, and we need a more larger neighborhood $\mathcal{D}$ in Theorem 3 because $R \sim \sqrt{n}$. They also provide a lower bound on the size of $\mathcal{D}$, showing that it is not possible to have a significantly smaller neighborhood. The minimal distance between the initial point $\theta^0$ and the closest global minimum is lower bounded by $\|\theta - \theta^0\|_2 \geq \frac{\|h_x(\theta^0)-y\|_2}{b}$, so the radius of $\mathcal{D}$ must satisfy $R \geq \frac{\|h_x(\theta^0)-y\|_2}{b}$.

They apply their results to the training of one-hidden layer neural networks with scalar output. This shows that the previous properties of GD hold when the number of data is less than the dimension of the input ($n \leq d$) for arbitrary initialization, width of hidden layer, and for strictly increasing activation function. These restrictions can be slightly relaxed by assuming random initialization. In this case, non-decreasing activation functions (such as ReLU) can be used, and we can have more data relative to the number of weights. We find these results very interesting since they apply to the exact same settings used in our numerical experiments with orthogonal input data. Indeed, orthogonal input data impose $n \leq d$.

### 2.1.7 Useful tools in Deep Learning

- **Visualization Tools for the Loss Landscape** : In [31], the authors present the basics of visualizing the loss landscape. Here, we briefly explain the three methods presented in the paper. The goal of these methods is to visualize the landscape in low dimensions (1D or 2D) while retaining the characteristics of the high-dimensional landscape we want to show.
  - **1-Dimensional Linear Interpolation** : This is a simple way to plot the value of the loss between two weight points $\theta^1$ and $\theta^2$. We draw the function $f(\alpha) = \mathcal{L}(\theta(\alpha))$ with $\theta(\alpha)) = \alpha\theta^1 + (1-\alpha)\theta^2$ and $\alpha \in \mathbb{R}$.

This visualization is useful for studying the sharpness of a minimizer or showing the variation of the loss value between the initial and final points generated by an optimization algorithm. However, it does not easily visualize non-convexities of the loss in 2D.

- **Contour Plots and Random Directions** : This method works as follows :
  1. Choose a center point $\bar{\theta}$ and two directions $\delta$ and $\eta$.
  2. Plot $f(\alpha, \beta) = \mathcal{L}(\bar{\theta} + \alpha\delta + \beta\eta)$ with $\alpha, \beta \in \mathbb{R}$.

  The directions $\delta$ and $\eta$ can be chosen randomly. To visualize in 1D, it is sufficient to set $\beta = 0$. This method is useful for exploring the training paths of different optimization algorithms.
- **Filter-Wise Normalization** : This method is similar to the previous one, except for the choice of the directions $\delta$ and $\eta$. A generic direction $d$ is chosen randomly with a random Gaussian vector, and each component is normalized as follows : $d_{i,j} \leftarrow \frac{d_{i,j}}{\|d_{i,j}\|_F}\|\theta_{i,j}\|_F$. Finally, we set $\delta = d$, and repeat the process to find $\eta$.
  This method removes scale effects when plotting the loss landscape, allowing us, for example, to find a correlation between the sharpness of a minimizer and its ability to generalize to unseen data, as discussed previously.

- **Lipschitzness of a ANN** : In [42], the authors discuss the sensitivity of deep neural networks to small, well-chosen perturbations of input data. To understand this sensitivity and improve the network's robustness, it can be important to know the regularity of the network by computing the Lipschitz constant of the network $h_\theta(x)$ with respect to $\theta$. This Lispchitz constant is also used in some generalization bounds. Unfortunately, they show that even for a one-hidden layer neural-net, the exact computation of the Lispchitz constant is a NP-hard problem. Subsequently, they introduce an algorithm called "SeqLip" that efficiently estimates the Lipschitz constant of a network. For fully-connected neural networks, SeqLip improves upon the well-known upper bound stated in the following proposition.

  **Proposition 4.** *For any Fully-Connected Neural Network with* 1*-Lipschitz activation function (e.g. ReLU, Sigmoid, Tanh), the exact Lipschitz constant of the network* $L_h$ *satisfies*

$$L_h \leq \prod_{l=1}^{L+1} \|W^{(l)}\|_2. \tag{2.38}$$

## 2.2 Training dynamics of one-hidden layer ReLU Neural Networks

In this section, we first aim to summarize the main results of [5], which describe the training dynamics of one-hidden layer ReLU neural networks for orthogonal input data. We then reproduce their experimental results using a one-dimensional orthogonal dataset. Finally, we briefly summarize the findings of [34], which provides results for the same settings with non-orthogonal input data. In the next chapter, we present our own numerical experiments, extending the experiments of [5] by varying some model hyperparameters. This approach helps us identify the limits of certain assumptions in [5] and better understand the training dynamics of shallow neural networks.

### 2.2.1 Orthogonal input data

Let us start by summarizing the main results of the paper [5]. In this article, the authors analyze the gradient flow dynamics for one-hidden layer ReLU networks with square loss

and orthogonal inputs, focusing on regression problems.
They provide an analysis of the following points :

- the convergence of gradient flow to zero loss;
- the implicit bias of the optimization algorithm used for training;
- the influence of the scale of initialization on the training dynamics.

#### 2.2.1.1 Model

The model analyzed in this paper is a one-hidden layer ReLU neural network, and it is depicted by the following figure 2.5. We use a simplified version of notations presented in section 1.1.3.2, since the model we use in this chapter has only one hidden layer and a scalar output. We recall

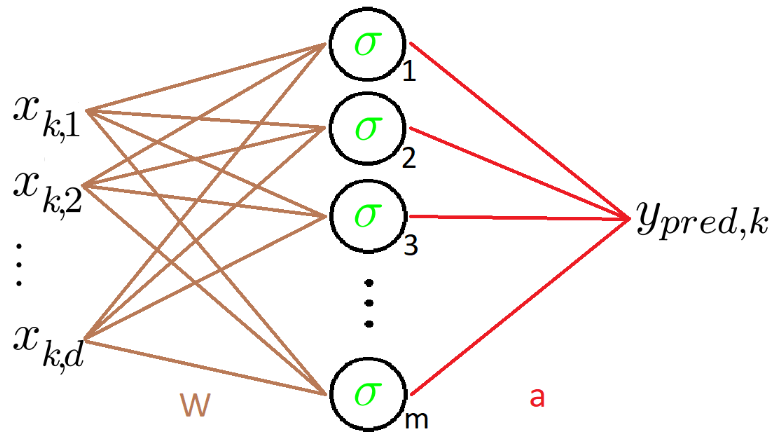



FIGURE 2.5: One-hidden layer neural network.

with

- the number of hidden neurons (called the width) : $m \in \mathbb{N}$ ;
- the number of data : $n \in \mathbb{N}$ ;
- the size of one input $x_k$ : $d \in \mathbb{N}$ ;
- the inputs : $X \in \mathbb{R}^{d \times n}$ ;
- the outputs : $y \in \mathbb{R}^n$ ;
- the weights : $\theta = (W, a) \in \mathbb{R}^{m \times d} \times \mathbb{R}^m$. Moreover, we denote

$$W = \begin{bmatrix} w_1^T \\ w_2^T \\ \vdots \\ w_m^T \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,d} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,d} \\ \vdots & \vdots & & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,d} \end{bmatrix}$$

with $w_j^T \in \mathbb{R}^d$ the line $j$ of $W$.

We minimize the square loss function

$$\mathcal{L}(\theta) = \frac{1}{2n}\sum_{k=1}^{n}\left(y_{pred,k} - y_k\right)^2 = \frac{1}{2n}\sum_{k=1}^{n}\left(h_\theta(x_k) - y_k\right)^2 \tag{2.39}$$

with $y_{pred,k} = h_\theta(x_k) = \sum_{j=1}^{m} a_j\, \sigma(w_j^T x_k)$ the prediction made by the model for the input $x_k$.

We define the gradient flow as follow

$$\frac{\mathrm{d}\theta^t}{\mathrm{d}t} = -\nabla\mathcal{L}(\theta^t) = -\frac{1}{n}\sum_{k=1}^{n}\left(h_{\theta^t}(x_k) - y_k\right)\nabla_\theta h_{\theta^t}(x_k). \tag{2.40}$$

with the superscript $t$ holding for the time. In roughly words, the gradient flow is the gradient method with infinitesimal step size. We initialize it with $\theta^0 = \left(W^0, a^0\right)$.
Moreover, when we differentiate $h_\theta(x)$ with respect to $\theta$, we differentiate the activation function $\sigma$. However, at zero, $\sigma_{\text{ReLU}}$ is not differentiable, so we have to consider the subgradient. For some reasons outlined in the paper, the only viable choice for the subgradient to ensure the existence of a global solution is

$$\sigma'_{\text{ReLU}}(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

We initialize $\theta^0 = (W^0, a^0)$ with a balanced initialization as follow

$$\begin{cases} w_j^0 = \lambda g_j \quad \text{where } g_j \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, I_d) & (2.41)\\ a_j^0 = s_j \|w_j^0\| \quad \text{where } s_j \overset{\text{i.i.d.}}{\sim} \mathcal{U}\left(\{-1, 1\}\right) & (2.42) \end{cases}$$

where $\lambda > 0$ is the scale of the initialization. If we choose a large $\lambda$, we are in a lazy regime, and if we choose it small, we are in a rich regime (which it is more interesting for learning).

The analysis of the training dynamics of the network can be reduced by looking at the hidden layer solely thanks to the following Lemma. This lemma states that the iterates $\theta^t$ remain balanced throughout the entire training process.

**Lemma 5.** *For all $t \geq 0$ and $j \in \{1, \cdots, m\}$, $(a_j^t)^2 - \|w_j^t\|^2 = (a_j^0)^2 - \|w_j^0\|^2$.*
*Assume furthermore that for all $j \in \{1, \cdots, m\}$, the initialization is balanced and non-zero : $|a_j^0| = \|w_j^0\| > 0$. Then $|a_j^t| = \|w_j^t\| > 0$, and letting $s = \text{sign}\left(a^0\right) \in \{-1, 1\}^m$, we have that $a_j^t = s_j\|w_j^t\|$ for all $t \geq 0$.*

In Section 4.3, we propose a proof of this lemma since the authors of [5] do not provide one.

The paper suggests that the analysis can be extended to unbalanced initialization, i.e. $\|w_j^0\| \neq |a_j^0|$ for some $j$.

#### 2.2.1.2 Assumptions

Now, we explicit the three assumptions made in this paper.

**Assumption 3.** *The input set $X$ is an orthogonal family, i.e.*

$$\forall k, l \in \{1, \cdots, n\}, \quad \langle x_k, x_l\rangle = \begin{cases} c > 0 & \textit{if } k = l \\ 0 & \textit{otherwise} \end{cases} \tag{2.43}$$

This assumption significantly restricts the applicability of the results. It can be extended to cover almost orthogonal data, i.e., data with an angle near 90 degrees. More precisely, quasi-orthogonal input data are input data $X$ such that $|\langle x_k, x_l\rangle| \leq \delta$, with $\delta$ being the deviation of the data from orthogonality. According to the authors, the results still hold for $\delta$ of order $\lambda$. However, for general cases of data, the analysis presented in this paper does not hold.

**Assumption 4.** *Condition to avoid degenerate cases :*

$$y_k \neq 0 \quad \forall k \in \{1, \cdots, n\}\,, \text{and} \sum_{k:y_k>0} y_k^2 \neq \sum_{k:y_k<0} y_k^2 \tag{2.44}$$

This assumption merely serves to avoid certain degenerate cases.

**Assumption 5.** *Condition to ensure the convergence of the gradient flow :*

$$S_{+,1} = \left\{ j \in \{1, \cdots, m\} \mid s_j = 1 \text{ and } \forall k : y_k > 0,\ \langle w_j^0, x_k\rangle \geq 0 \right\} \neq \emptyset \tag{2.45}$$

$$S_{-,1} = \left\{ j \in \{1, \cdots, m\} \mid s_j = -1 \text{ and } \forall k : y_k < 0,\ \langle w_j^0, x_k\rangle \geq 0 \right\} \neq \emptyset \tag{2.46}$$

This assumption is crucial to ensure the convergence of the gradient flow. It is, in fact, a necessary and sufficient condition to guarantee the convergence of the gradient flow to a minimal norm interpolator in the over-parametrization regime, i.e. when the model has many more parameters than training data.

#### 2.2.1.3 Main results of paper [5]

**Convergence and implicit bias**

**Theorem 6.** *Under assumptions 1 to 3, there exits $\lambda^* > 0$ depending only on the data and the width $m$ such that, if $\lambda \leq \lambda^*$, the gradient flow initialized according to 2.41 converges almost surely to some $\theta_\lambda^\infty$ of zero training loss, i.e. $L\left(\theta_\lambda^\infty\right) = 0$.*
*Furthermore, there exists $\theta^*$ such that*

$$\lim_{\lambda \to 0} \lim_{t \to +\infty} \theta^t = \theta^* \in \underset{\mathcal{L}(\theta)=0}{\arg\min} \|\theta\|^2.$$

This theorem ensures several aspects of the convergence of the gradient flow. Firstly, Theorem 6 ensures that the gradient flow converges to zero loss. Secondly, it characterizes the implicit bias[8] of the gradient flow when the initialization scale is infinitesimal. In our case, Theorem 6 ensures that gradient flow converges to a global minimizer with the smallest $l_2$-norm. Furthermore, this fact only depends on the internal structure of gradient flow and the scale of initialization.

Moreover,

$$\lambda^* \sim \frac{\mathbf{\Theta}(1)}{\sqrt{m}} \mathrm{e}^{-\mathbf{\Theta}(n)}\,. \tag{2.47}$$

So, the more data we have, the smaller the initialization of weights can be to maintain the same kind of theoretical results. However, the authors do not provide the explicit formula

[8] We recall that the implicit bias of an optimization algorithm is the tendency for the algorithm to converge to a minimizer rather than other ones.

for $\lambda^*$. They also state that, empirically, GD still converges towards a minimal norm interpolator with higher $\lambda$[9], but the training dynamics change[10].

### Training dynamics

During the training, we observe an alignment of neurons, and a saddle-to-saddle dynamics. There are four phases during the training process of the model 2.2.1 trained under the settings of Theorem 6:

**Phase 1.** Alignment phase : During this phase, all neurons maintain a small norm, and neurons with similar activations align themselves in the same direction. In the end, there are two main directions for neuron alignment

$$D_+ = \frac{1}{n} \sum_{k:y_k>0} y_k x_k \tag{2.48}$$

$$D_- = \frac{1}{n} \sum_{k:y_k<0} y_k x_k. \tag{2.49}$$

**Phase 2.** Positive labels fitting : During this phase, the neurons in $S_{+,1}$ (i.e. the neurons align with $D_+$) grow in norm to fit data with positive label.

**Phase 3.** Negative labels fitting : During this phase, the neurons in $S_{-,1}$ (i.e. the neurons align with $D_-$) grow in norm to fit data with negative label.

**Phase 4.** Final convergence : Finally, we enter the phase where the gradient flow converges to an interpolator that is close to an interpolator of minimal norm. When $\lambda \to 0$, the distance between the obtained interpolator and the minimal norm interpolator also tends to zero.

At the same time, we can observe a saddle-to-saddle dynamics, i.e., the training trajectory taken by the gradient flow passes near a saddle point of the loss function (resulting in very slow convergence at this moment), then it enters a quick convergence phase, and repeats this phenomenon until it finds a minimizer.
In our case, it starts near a saddle point, quickly converges until the end of phase 2, and encounters another saddle point before converging to the global minimizer.

They also provide the speed at which each of these phases occurs. For a small $\epsilon$ depending only on the dataset and the width of the hidden layer, we have

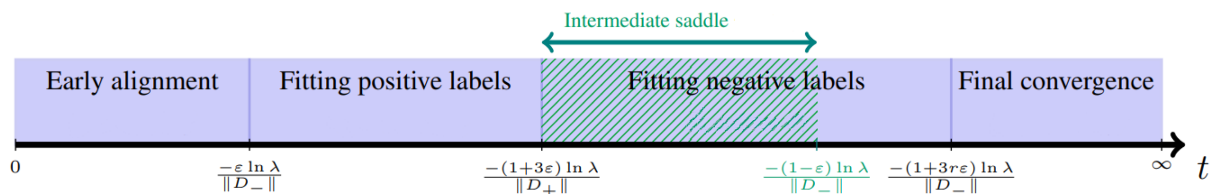



FIGURE 2.6: Timeline of the training dynamics of one-hidden ReLU neural networks for orthogonal data. (Timeline from [5])

[9] If the dynamics is still in the mean field regime. We do not detail this notion here.
[10] There is no separation between Phase 2 and Phase 3 (see the following section), resulting in the disappearance of intermediate saddle from the trajectory.

#### 2.2.1.4 Reproduction of numerical experiments of paper [5]

Using the model described in Section 2.2.1, we employ the following dataset

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -0.5 & 1 \\ 2 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

with the last component of each $x_k$ representing the bias term.

We have the following hyperparameters

- the width of the ANN : $m = 60$,
- the number of data : $n = 2$,
- the size of one input : $d = 2$,
- the scale of initialization : $\lambda = \frac{10^{-6}}{\sqrt{m}} \approx 1.2909 \times 10^{-07}$,
- the step-size of gradient method : $\alpha = 10^{-3}$,
  we take it small in order to approximate the gradient flow.

We tried to reproduce the graphs and results described in [5]. The results are plotted on Figures 2.7 and 2.8 just below. Figure 2.7 represents the training loss value. We can observe the saddle-to-saddle dynamics as expected in the article. Iterates seem to start near an initial saddle point, converge faster, to pass near an intermediate saddle point before converging fast to a global minimum (with zero loss). On Figure 2.8, we can observe the alignment of each neuron (blue stars and left y-axis) and the resulting model (green line and right y-axis) for the dataset we aim to generalize (red points) for some epochs corresponding to the four Phases presented in section 2.2.2. In short, we can observe the initial state of the network at Epoch 0; at Epoch 4000, we observe the end of the alignment phase; at Epochs 8000 and 15000, the model has reached the positive and negative label data respectively; and finally, after 15000 it is the final convergence to zero loss. Neuron alignment is described in the orientation-signed norm space $(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|)$, where $-\frac{w_{j,2}}{w_{j,1}}$ is equivalent to the angle of each neuron in the parameter space $(w_{j,1}, w_{j,2})$. Specifically, the angle $\beta$ between the $w_{j,1}$-axis and the vector $(w_{j,1}, w_{j,2})$, is given by $\beta = \arctan\left(\frac{w_{j,2}}{w_{j,1}}\right)$.

**Remark.** On Figure 2.8, we only draw the weight related to the hidden layer. In article [5], they state that for a ReLU one-hidden layer ANN and a balanced initialization, the following iterates remain balanced until the end of the training thanks to Lemma 5. Mathematically, $\forall t \geq 1$

$$a_j^t = s_j\|w_j^t\| \tag{2.50}$$

if the initial point $\theta^0$ is drawn from a balanced initialization. We check experimentally if this behavior remains for other settings in the following chapter, and we propose a proof of the lemma in Section 4.3.

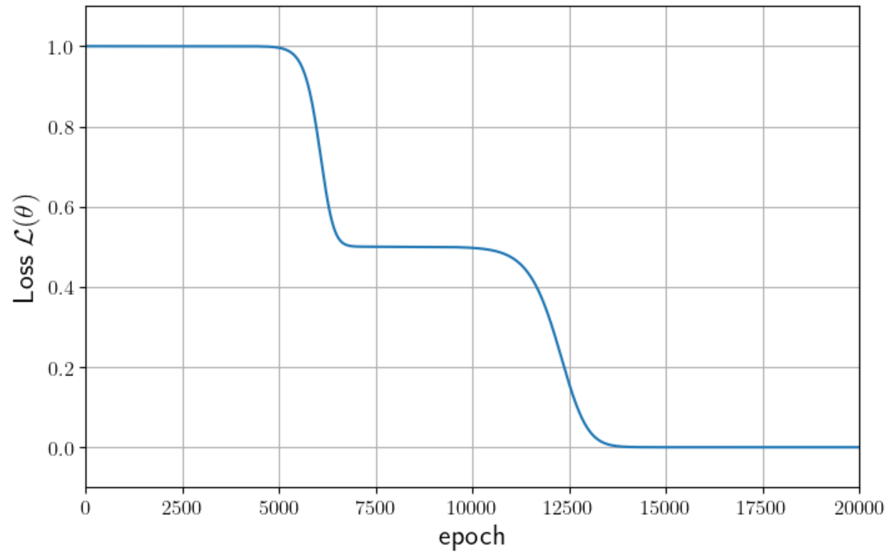



FIGURE 2.7: Loss value in function of iteration of gradient method (epoch).

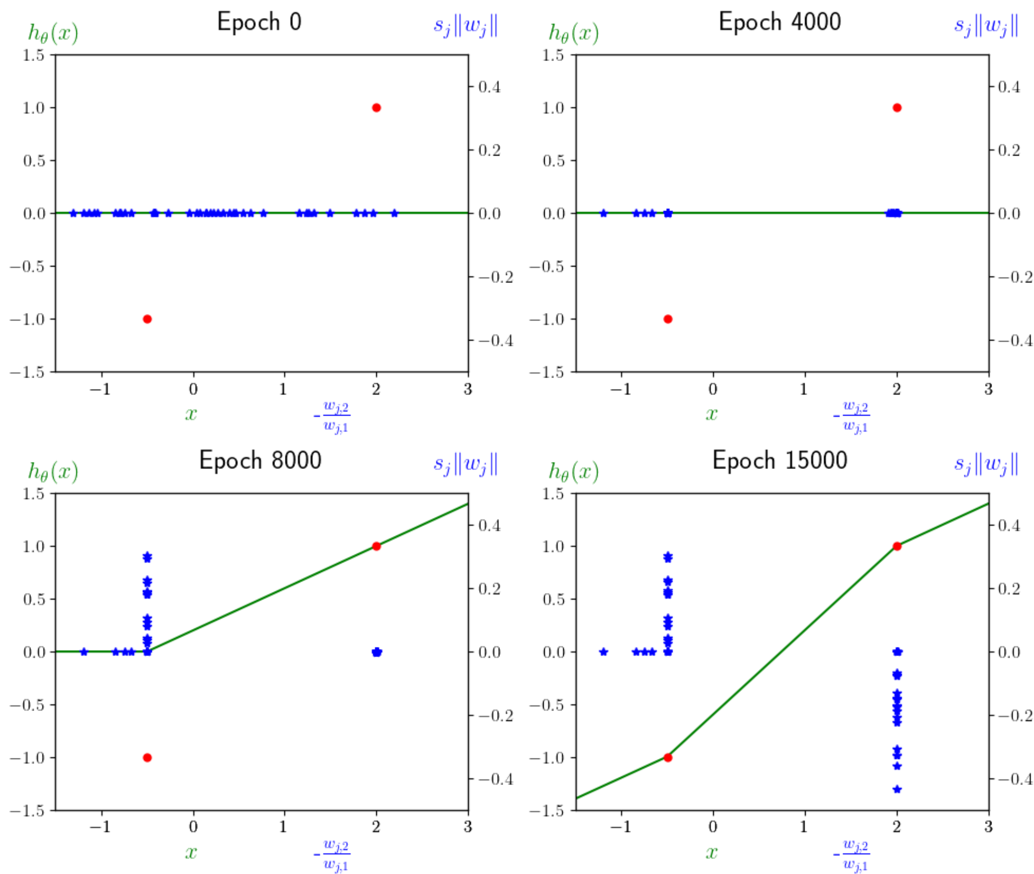



FIGURE 2.8: Neurons alignment during training. Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(X, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = \frac{10^{-6}}{\sqrt{60}}$.

### 2.2.2 Non-orthogonal input data

In [5], they observe experimentally that, for very small initialization scale $\lambda$, the training dynamics for non-orthogonal input data is similar to one describe in section 2.2.1.3.

In a second article [34], they analyze the question of why the Gradient Method finds pretty good solutions in over-parameterized regime minimizing the square loss (or the cross-entropy loss) for one-hidden layer ReLU Neural Networks (i.e. the same structure of models than the previous article [5]). They initialize the network with normal initialization and scale factor $\lambda$, i.e. $\theta_j = \lambda g_j$ with $g_j \sim \mathcal{N}$. Moreover, they assume small initialization and small learning rate.

They highlight a similar quantization effect during the training for general training data : the weight vectors tend to concentrate at a small number of directions determined by input data. In other words, neurons align in a finite number of directions. The training dynamics occurs in two different steps :

1. without changing the loss significantly, the neuron weight vectors align to a discrete set of possible directions, which only depend on the training data (not on the size of the network).
2. Loss is reduced, and the neurons keep their orientation.

They also show that, because of the previous behavior, there exists a finite set of simple resulting models that can be obtained for a specific input data, which is independent of the size of the network. Indeed, the training behavior implies that the resulting network is equivalent to a simpler one obtained by greedily adding one neuron at a time. They call this network "a replacement network", and they argue that there exists only a finite number of such networks.

Experimentally, they remark that in this setting with small initialization, the final model is often a "simple" function, that is relatively similar to a simple linear interpolation of the training data. For larger initialization scale, they retrieve more complex resulting models. For 1D input, they show that the final model tends to be a simple piecewise linear function while $\lambda$ decreasing.

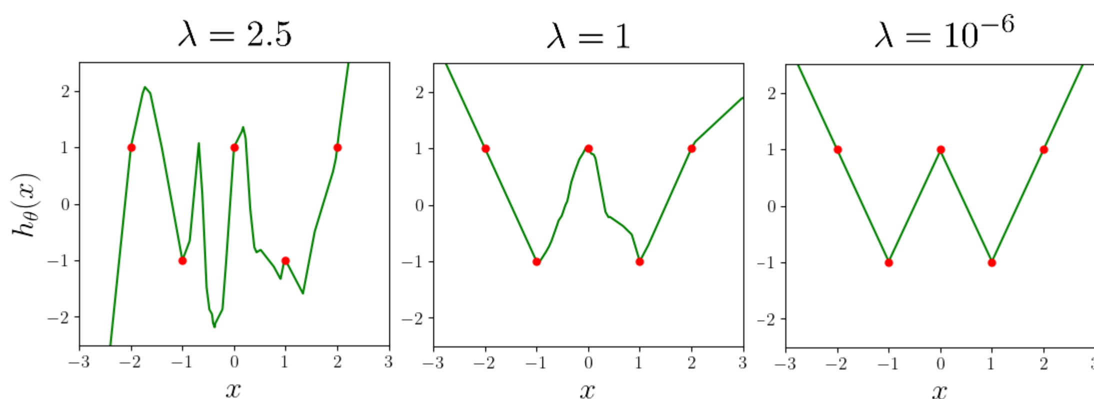



FIGURE 2.9: Final interpolator for one-hidden ReLU ANN with 60 neurons with $\lambda = 2.5$ (Left), $\lambda = 1$ (Middle) and $\lambda \leq 10^{-6}$ (Right).

# Chapter 3

# Additional Numerical Experiments

After reporting the theoretical results of [5] and replicating their basic experiment to verify their validity and to have a comparison point, we aim to discern the limitations of the postulated assumptions, and to uncover behaviors of neurons in groups over the training process. To achieve this, we conduct identical numerical experiments as in [5], employing varied sets of hyperparameters and different datasets to train a one-hidden layer neural network. We attempt to answer to the following questions :

- Is there a convergence to zero loss ?
- Is this convergence fast or slow ?
- How does the training dynamics (i.e. neurons alignment, saddle-to-saddle dynamics) change with respect to each hyperparameter ?
- Is the final interpolator minimal in the $\ell_2$-norm ?
- What is the transition like between the lazy regime and the rich regime ?
- Is the iterates $\theta^t$ still balanced throughout the training ?

To address these questions, we use several unidimensional and multidimensional datasets and various hyperparameter sets. Note that, for input data, we do not consider bias in the dimension. Thus, when the input data is unidimensional, adding the bias term gives $d = 2$ in the model by adding a supplementary entry $x_{k,d} = 1$ for each data point $k$. For this chapter, we reserve in the text the notation $x_k$ for input data that includes the bias term, and we denote it as $\bar{x}_k$ when the bias term is excluded[1]. Our base configurations include:

- Width of the ANN : $m \in \{2, 10, 60, 100, 600, 1000\}$,
- Type of activation function : ReLU, Sigmoid or Tanh,
- Type of initialization : balanced or normal,
- Scale of initialization : $\lambda \in \{10^{-10}, 10^{-8}, 10^{-6}, 10^{-4}, 10^{-2}, 1, 10^{2}\}$,
- Step-size of the gradient method : $\alpha \in \{10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}\}$, (We choose a small step size to approximate the gradient flow.)
- Maximum number of epochs : $3 \times 10^5$ for dataset with $n = 2$. For bigger dataset, we perform $10^6$ epochs, and if the loss does not decrease at all, we stop the training. If the loss decreases, we continue the training for $10^6$ epochs, repeating this process until convergence.

[1] In the figures, the use of the bias term is always clear, so we do not distinguish between the two.

Intermediate values for some hyperparameters can be used if necessary.

Each dataset is indexed in the following format: a.b.c. "a" represents the dimension of the input data, "b" denotes the type of input data ("1" for orthogonal, "2" for quasi-orthogonal, and "3" for non-orthogonal), and "c" is the dataset number when multiple datasets are of the same type. The different kinds of datasets for unidimensional data are:

- datasets 1.1.x : orthogonal data ($n = 2$)
- datasets 1.2.x : quasi-orthogonal data ($n = 2$) with $\delta \in \{10^{-8}, 10^{-6}, 10^{-4}, 10^{-2}, 1, 10^{2}\}$
- datasets 1.3.x : non-orthogonal data ($n = 5$ or $n = 10$) with or without patterns,

We recall that $n$ is the number of data points in the dataset, and $\delta$ is the deviation of the data from orthogonality. The details of each dataset are clearly explained in the corresponding section below.

Finally, we conduct numerical experiments on multidimensional input data. We randomly generate multidimensional datasets and use visualization methods to observe the disposition of neurons in several representations. To clarify, we will refer to *neuron alignment* when neurons in a group share the same direction, and we will use *neuron disposition* when groups of neurons form possibly other coherent patterns or shapes. The methods used are detailed in the corresponding sections.

## 3.1 Unidimensional orthogonal data

In this section, we present the results of our numerical experiments on unidimensional orthogonal input data. We begin with an in-depth discussion of the findings from the toy dataset used to replicate the experiment in [5], referred to as "Dataset 1.1.0." We analyze the general aspects of the training loss curve, neuron alignment/disposition throughout the training process, and the implicit bias of gradient descent (GD) when varying different hyperparameters to address the questions posed in the introduction of this chapter. Next, we briefly discuss the impact of modifying the labels in Dataset 1.1.0. Finally, the last subsection presents the results for an alternative orthogonal dataset with different label values, highlighting certain behaviors in the disposition of neurons for Sigmoid and Tanh activation functions in the hidden layer.

### 3.1.1 Dataset 1.1.0 : original data

We begin our numerical experiments using the same toy dataset as in [5], which is again detailed below :

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -0.5 & 1 \\ 2 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

with the last component of each $x_k$ representing for the bias term. Since this will be the case for all our datasets, we will not repeat this in each section.

**(i) Loss value during training.**

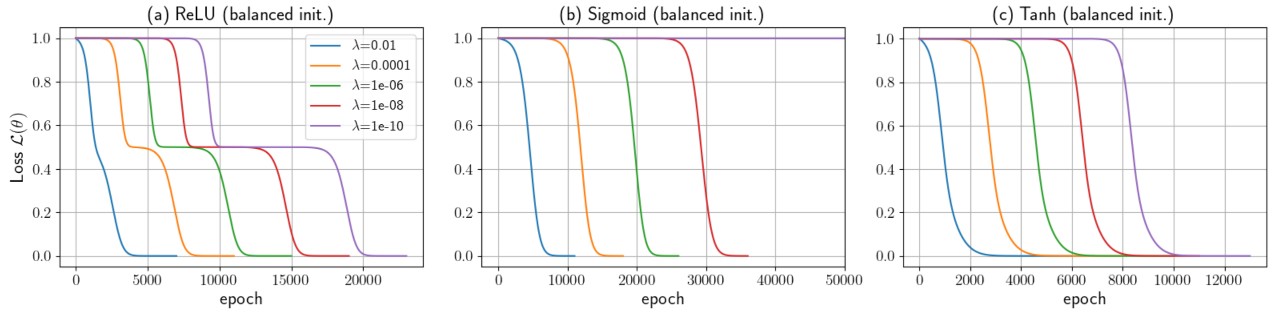



FIGURE 3.1: Training loss curves for one-hidden layer model with 60 neurons, $\alpha = 10^{-3}$. Comparison with respect to the activation function and the initialization scale (in rich regime).

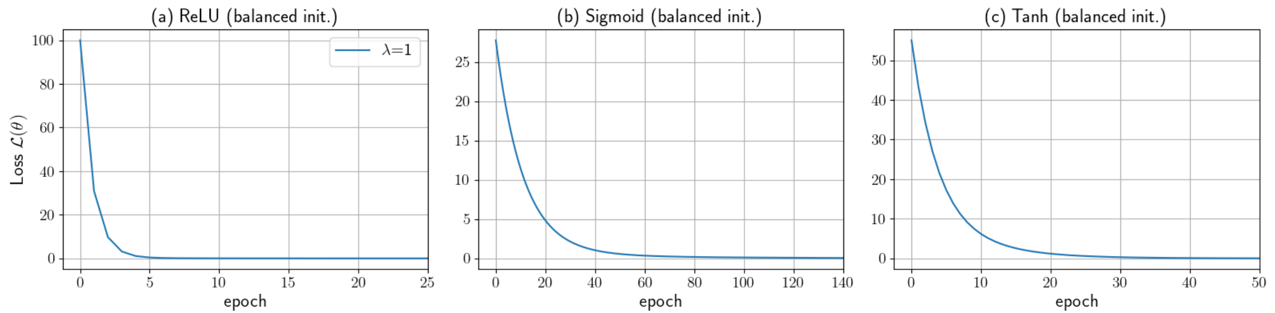



FIGURE 3.2: Training loss curves for one-hidden layer model with 60 neurons, $\alpha = 10^{-3}$. Comparison with respect to the activation function and for an initialization scale $\lambda = 1$ (in lazy regime).

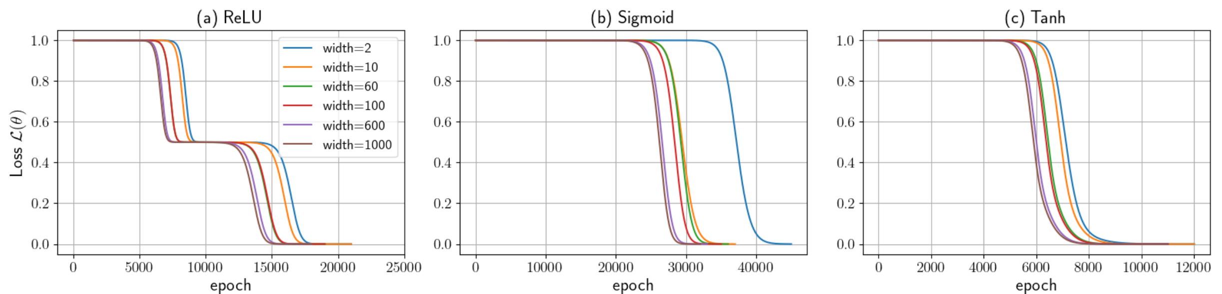



FIGURE 3.3: Training loss curves for one-hidden layer model with $\alpha = 10^{-3}$, and balanced init. with $\lambda = 10^{-8}$. Comparison with respect to the width $m$ of the hidden layer (in rich regime).

In Figure 2.7 (reproduction of results in [5]), we can see the shape of the loss curve during the training of a one-hidden layer ReLU neural network. In Figures 3.1, 3.2, and 3.3, we show the resulting loss curves for several hyperparameter settings using the default orthogonal dataset.

In Subfigure 3.1 (a), we display the loss curves for the ReLU activation function and a balanced initial point $\theta^0$. Each curve corresponds to a different initialization scale $\lambda$. For $\lambda$ between $10^{-10}$ and $10^{-2}$, we observe a similar overall shape of the loss, exhibiting the same saddle-to-saddle dynamics as described in 2.2. Moreover, smaller $\lambda$ values lead to slower

training, exhibiting the same dependency as stated in [5] (see Timeline 2.6). Indeed, the initial plateau, intermediate plateau, and convergence to a near-zero loss take longer as $\lambda$ decreases, following a dependency proportional to $-\ln(\lambda)$. We call "plateau" the parts of the loss curve where the derivative is very low, indicating a slow training. Thus, we can expect that with a higher $\lambda$, the plateaus become so small that they disappear, and the training dynamics change. This is evident in 3.2 (a) with an initialization scale $\lambda = 1$. Here, the model transitions from the rich regime to the lazy regime, where the training dynamics with GD resemble those of minimizing a convex function.
We perform the same numerical experiments for ReLU activation function and a normally initial point $\theta^0$ (i.e. the weights of the hidden and output layers are chosen independently with a normal distribution). As mentioned in the article, in the rich regime, the results are very similar to those with balanced initialization. This similarity also extends to the lazy regime and holds true for all the experiments we conducted.

In Subfigures 3.1 (b) and 3.1 (b), we display the loss curves for Sigmoid activation function, and on Subfigures 3.1 (c) and 3.1 (c), we show the loss curves for Tanh activation function. In both cases, in rich regime, the training loss curve does not exhibit an intermediate plateau. However, it begins with an initial plateau that increases as $\lambda$ decreases, which is a behavior similar to that of ReLU activation function.
As $\lambda$ increases beyond a certain threshold (see Figure 3.2), the initial plateau disappears, analogous to the ReLU case, and we enter the lazy regime.

In Figure 3.3, we observe the influence of the number of neurons in the hidden layer (i.e., the width $m$) for the three considered activation functions. Visually, the influence of $m$ on the training dynamics appears to be marginal compared to $\lambda$. We see that, as $m$ decreases, more epochs are needed to converge. It seems logical since we have less weights in the model. However, an epoch takes longer when $m$ is higher, resulting in a significantly longer training time for larger $m$ values.

**(ii) Neuron alignment and interpolator.**

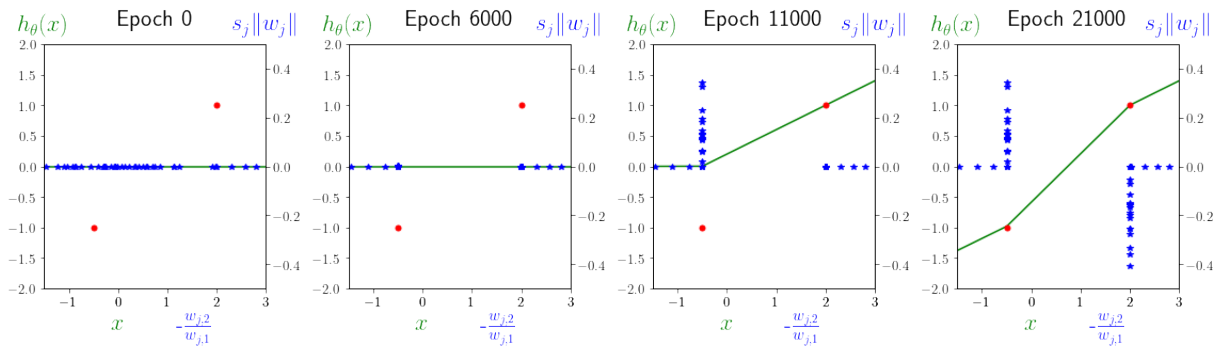



FIGURE 3.4: Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-10}$.

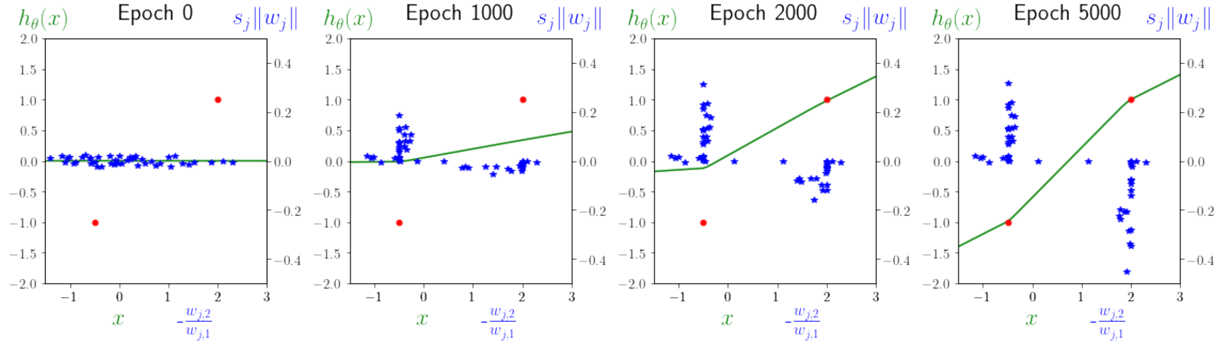



FIGURE 3.5: Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-2}$.

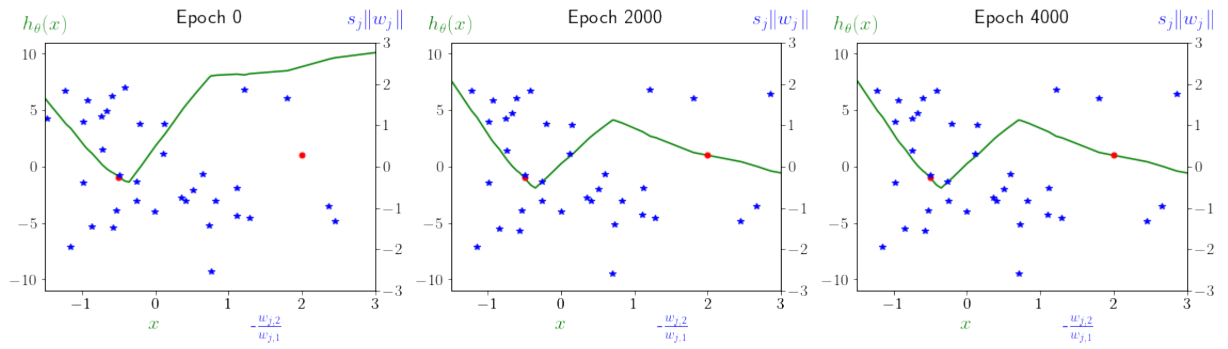



FIGURE 3.6: Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 1$.

On Figures 3.4, 3.5 and 3.6, we plot the neuron norm signed with the initial sign of the corresponding output weight (i.e. $s_j\|w_j\|$) as a function of their orientation $-\frac{w_{j,2}}{w_{j,1}}$ (blue stars) at some interesting epochs for ReLU network. Superimposed on these, we draw the model (green line) with respect to the unidimensional input $\bar{x}$ (without the bias entry). Additionally, we plot the data points $(\bar{x}_k, y_k)$ with red dotes. These figures illustrate this for an initialization scale $\lambda \in \{10^{-10}, 10^{-2}, 1\}$ for a ReLU one-hidden layer ANN with a width of 60. They correspond to the loss curves presented in Figures 3.1 (a) and 3.2 (a).

For $\lambda \leq 10^{-2}$, we can clearly observe the four phases presented in section 2.2 :

1. **The Alignment Phase** : Shown in the second window of each figure (when possible, since data is saved every 1000 epochs). Comparing this phase for different values of $\lambda$, we see that the smaller the $\lambda$, the better the alignment. This is particularly clear when comparing the alignment for $\lambda = 10^{-10}$ and for $\lambda = 10^{-2}$.

2. **The end of the Positive Labels Fitting Phase** : Shown in the third window of each figure. Neurons corresponding to positive labels grow in norm until the model reaches the positive label data.

3. **The end of the Negative Labels Fitting Phase** : Shown in the fourth window of each figure. This phase follows the same pattern as the positive labels fitting phase.

4. **The Final Convergence Phase happens** : Occurs at the end of the negative labels fitting phase. This phase is more explicit in the figures showing the training loss curve.

Additionally, the model is piece-wise linear and breaks when encountering the two data points. This precisely aligns with results of [5] and with the statement in [34], which notes that the final interpolator generally resembles the piece-wise linear interpolation of the dataset.

For $\lambda = 1$, the difference is evident. The neurons are already large in norm at initialization and have random orientations. To reach the data, the neurons only shift slightly, without aligning or adopting specific structures as they do with smaller $\lambda$ values. When training concludes, the resulting model is more complex, with more breakpoints.

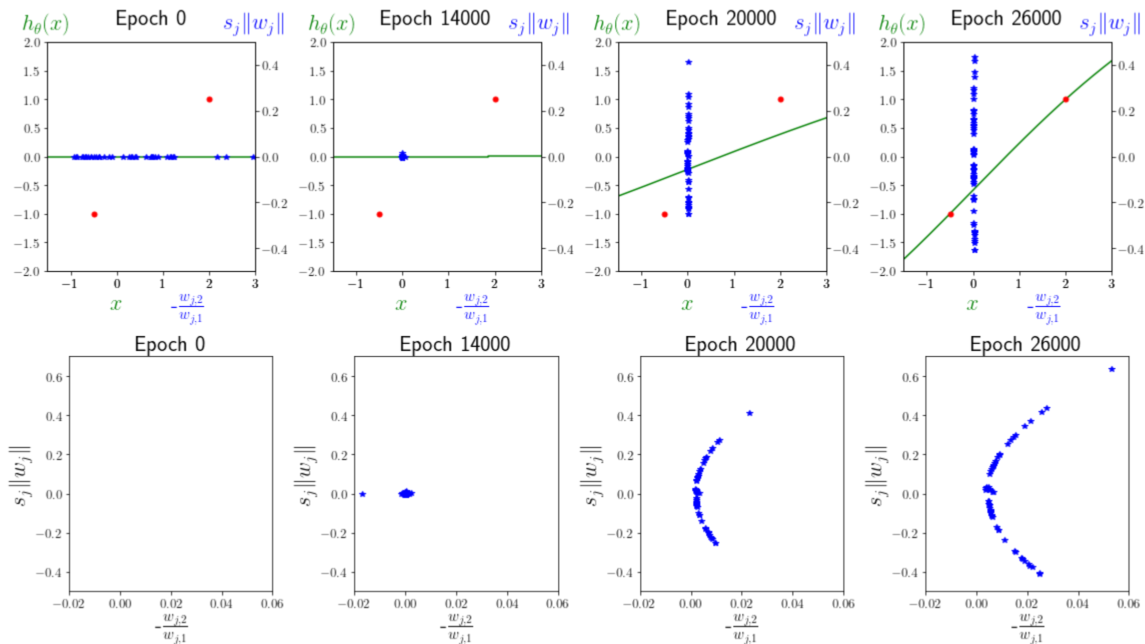



FIGURE 3.7: Above : Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a Sigmoid one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-6}$. Below : A zoom on the neuron disposition to see better the shape of neuron alignment.

For Sigmoid activation function and small $\lambda$ (in rich regime), neurons align similarly to ReLU, but all in the same direction. Subsequently, neurons grow in norm and reach positive and negative label data simultaneously. Unlike ReLU, as neurons grow, they slightly change direction, forming a shape resembling the tip of an ellipse[2]. The resulting interpolator appears almost as a straight line. We can observe this in Figure 3.7, which displays the training process for the Sigmoid activation function with $\lambda = 10^{-6}$. For larger $\lambda$ (in lazy regime), the training resembles that of ReLU, but the resulting model is smooth. We do not observe any particular shape formed by neurons, and neurons only shift slightly over the whole training process.

[2]We will see later that the shape formed by neurons looks more like a bell lying horizontally.

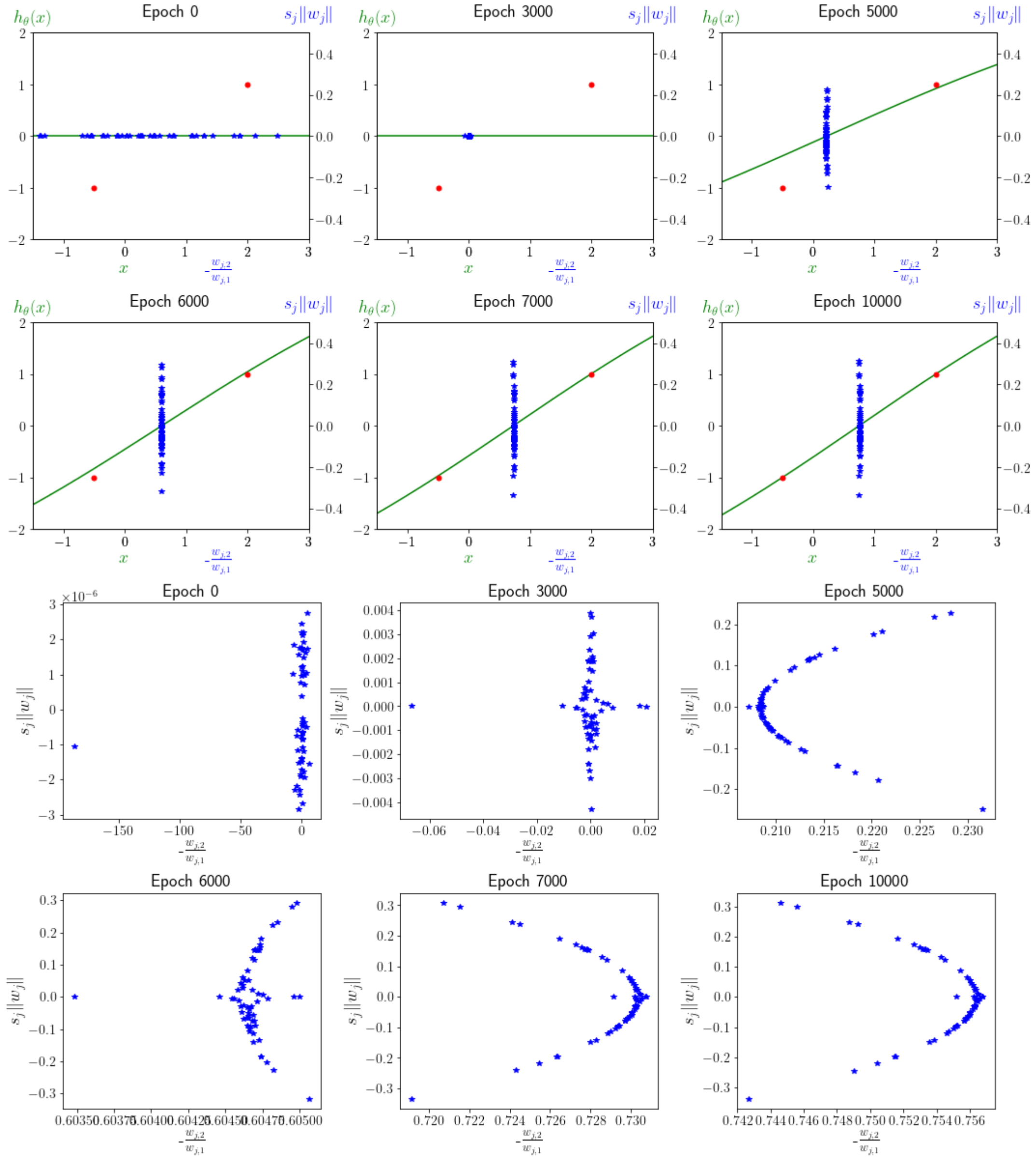



FIGURE 3.8: Above : Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a Tanh one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-6}$. Below : A zoom on the neuron disposition to see better the shape of neuron alignment.

For Tanh activation function and small $\lambda$ (in rich regime), we can observe a neuron disposition similar to that of Sigmoid. After that, neurons grow in norm, forming a pseudo-half-ellipse to the right to reach the positive label data, before changing direction to reach the negative label data. The resulting model is very similar to a straight line too. In Figures 3.8, we provide an illustration of the training for Tanh activation function with $\lambda = 10^{-6}$. For larger $\lambda$ (in lazy regime), the training dynamics are similar to those for ReLU and Sigmoid.

**(iii) Neuron representation in $(w_{j,1}, w_{j,2})$ plane.**
As we will need it later, we introduce another way to represent neuron disposition. The first representation we used was in the $(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|)$ plane (e.g., see Figure 3.4). This is useful for showing neuron alignment in ReLU ANNs and visualizing the relationship between neurons and the model. However, in some cases, we prefer to use the $(w_{j,1}, w_{j,2})$ plane, even though it does not show the model superimposed on the neuron representation. We provide an example below which illustrates the training of ReLU ANNs under the same settings as Figure 3.4. In this Figure, neurons are colored in magenta if they lie in $S_{1,+}$ and in aqua blue if they lie in $S_{1,-}$. Other neurons remain colored in blue.

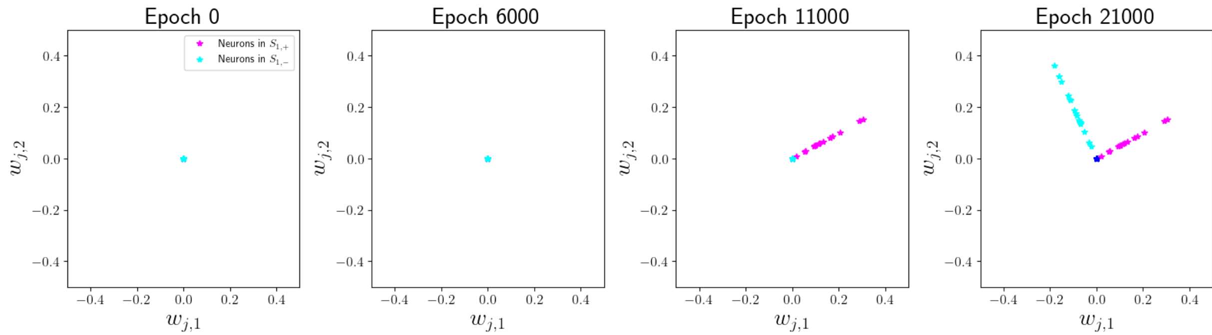



FIGURE 3.9: Neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-10}$.

**(iv) Balancedness of the iterates $\theta^t$ throughout the training.**
The observations of neuron disposition across the various figures provide a complete explanation of the whole training dynamics for the ReLU activation function. This is ensured by Lemma 5, which asserts that the iterates remain balanced throughout the training process. We now examine by experiment whether this observation holds true and whether it also applies to the Sigmoid and Tanh activation functions. On Figure 3.10, we plot the $\ell_1$ balancedness error averaged over the number of neurons $m$, as well as the $\ell_\infty$ error during training (blue curves) in the first and second rows, respectively, for the three activation functions. For comparison, we also plot the average absolute value of $a^t$ and $\|w^t\|$. We observe that for ReLU, the iterates remain perfectly balanced throughout the entire training process. For Sigmoid and Tanh, this is also mostly true, with only slight deviations at some stages of training, likely due to numerical error or the fact that the balancedness property applies to Gradient Flow, whereas we used GD with a finite step size for training. For example, in Figure 3.10, a deviation is observed at the end of the training. However, this deviation can occur earlier, and the balancedness might be restored afterward, depending on the dataset. In Appendix C.1.2, you can find plots of the evolution of $a_j^t$ for each neuron over epochs.

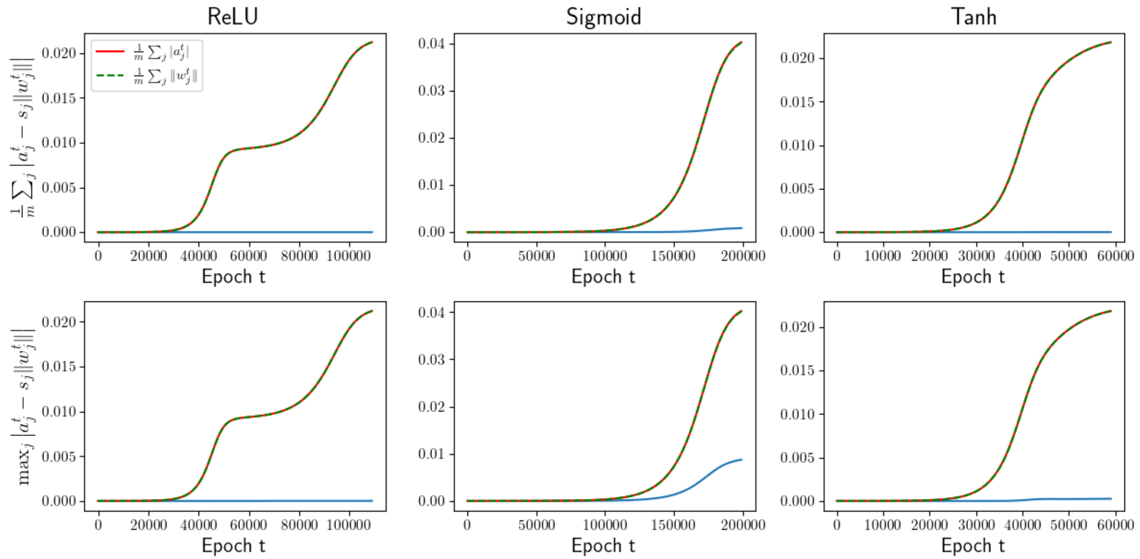



FIGURE 3.10: Balancedness of $\theta^t$ throughout the training process for ReLU, Sigmoid and Tanh. Upper windows : averaged $\ell_1$-norm of the balancedness error $\frac{1}{m}\sum_{j=1}^{m}\left|a_j^t - s_j\|w_j^t\|\right|$ (blue line). Lower windows : $\ell_\infty$-norm of the balancedness error $\max_{j\in\{1,\cdots,m\}}\left|a_j^t - s_j\|w_j^t\|\right|$ (blue line). In each window, $\frac{1}{m}\left|a_j^t\right|$ (red line) and $\frac{1}{m}\left\|w_j^t\right\|$ (green line) are plotted to provide the order of magnitude of the weights and to facilitate comparison with the computed errors.

**(v) Complexity of the resulting interpolator.**

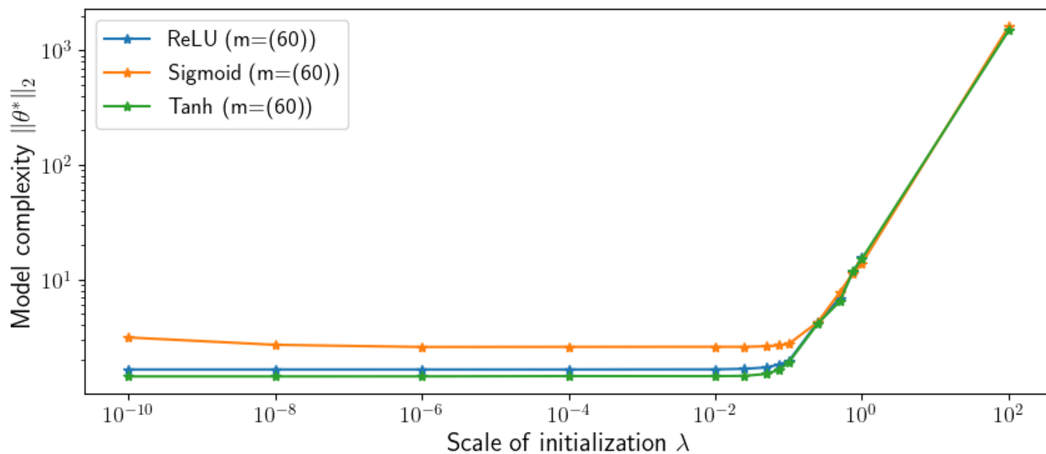



FIGURE 3.11: Model complexity with respect to $\ell_2$-norm (i.e. $\|\theta^*_{\text{GD}}\|_2$) as function of $\lambda$ for a network with 60 neurons and a balanced initialization.

Figure 3.11 above illustrates the relationship between the initialization scale $\lambda$ and the norm of the final weight vector $\theta^*_{\text{GD}}$ generated by the Gradient Method. We refer to this norm as the *model complexity*. The three activation functions exhibit a similar relationship: for $\lambda \leq 10^{-2}$ (in the rich regime), the model complexity is low and constant across all $\lambda$. For higher values of $\lambda$ (in the lazy regime), model complexity increases linearly with $\lambda$. While we cannot definitively state that the lowest complexity value on the graph is close to the minimum possible (i.e. $\theta^* \in \arg\min_{\mathcal{L}(\theta)=0}\|\theta\|^2$), this result aligns with the findings in [5].

This figure also reveals the transition phase between the lazy and rich regimes for all three activation functions, which appears to occur for $\lambda$ values between $10^{-2}$ and 1 in the settings of this section. Notably, this transition seems to occur at the same $\lambda$ value for all three activation functions.

**(vi) Transition between Lazy and Rich regimes.**

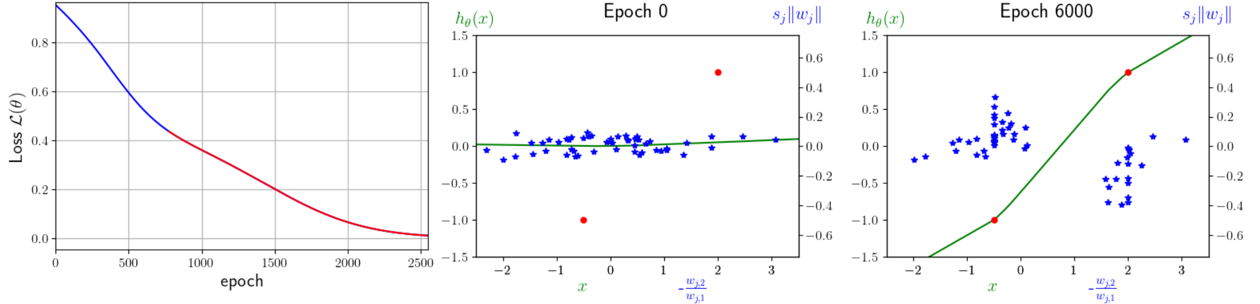



FIGURE 3.12: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 3.5 \times 10^{-2}$.

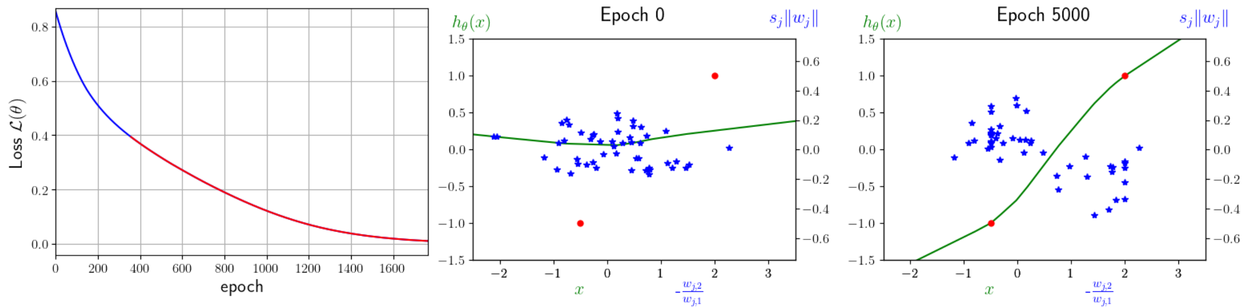



FIGURE 3.13: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 7.5 \times 10^{-2}$.

As discussed in the previous section, the transition between the lazy and rich regimes does not happen instantaneously. Instead, there appears to be a transition phase between the two. In the two figures above, we plot the model and neuron alignment for the ReLU activation function with $\lambda = 3.5 \times 10^{-2}$ and $\lambda = 7.5 \times 10^{-2}$. As expected, there is a mix of both regimes, with pseudo-alignment of neurons displaying moderate variance in their alignment direction, and a relatively simple resulting interpolator. Moreover, the training loss curve progressively becomes convex as $\lambda$ increases. Interestingly, the loss becomes convex faster than the increase in alignment variance, suggesting that this transition phase might offer a good trade-off between the better generalization performance of rich regime and the fast convergence of lazy regime. Similar behaviors can be observed for Sigmoid and Tanh.

**(vii) Automatic processing of experiment results for all generated hyperparameter setups.** In Section C.1.4, we detail the methods used to verify the results of our numerical experiments. In summary, after training models with various hyperparameter settings, we conducted automatic processing to detect four aspects of the training: convergence to zero loss, the number of plateaus in the loss curve, and the presence of initial and intermediate plateaus. Detecting these elements guided our research and confirmed that our observations were not isolated or random cases. This automatic processing was applied to all orthogonal and quasi-orthogonal datasets. However, due to the extended training time and frequent vanishing gradient issues with non-orthogonal data, we could not use this processing for such datasets.

**Remark.** In some figures, the training loss curve is shown with red and blue parts. This coloring results from our detection method. Based on a threshold defined in Section C.1.4, we classify each epoch as either "fast" or "slow." In our plots, slow epochs are colored red, while fast epochs are colored blue.

### 3.1.2 Orthogonal data : role of the labels

#### 3.1.2.1 Dataset 1.1.1 : Switched Labels

In these experiments, we use a modified dataset where the labels have been switched with respect to the Dataset 1.1.0 to observe whether the training dynamics remain unchanged for the three activation functions, and if the model still reaches the positive labels first for ReLU activation function. The dataset is :

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -0.5 & 1 \\ 2 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

From our experiments, we did not observe any differences in the training dynamics for the three activation functions. The training of ReLU networks follows the same phases as reported in Section 2.2.1.3. Similarly, for Sigmoid and Tanh, there are no noticeable differences.

#### 3.1.2.2 Dataset 1.1.2 : Labels of Same Sign

Once again, we modified the labels of Dataset 1.1.0. The goal here is to observe what happens when all data points have labels with the same sign. Therefore, we altered the original dataset by assigning the same label to both data points :

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -0.5 & 1 \\ 2 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Here, we observe more interesting phenomena. For ReLU, the loss curve no longer shows any intermediate plateau (see Figure 3.14). Despite the orthogonality of the input data, we observe an unexpected second direction in neuron alignment (see Figure 3.15). Initially, we only expected the direction associated with the positive labels. Additionally, the directions of the two branches change as they grow in norm, differing from the results in [5]. This occurs because the model seeks to be horizontal to perfectly interpolate the data, requiring two breakpoints[3], each corresponding to a direction of alignment. This is due to the absence of bias in the output layer. When one of the two labels is multiplied to observe the effect of differing label values, we still observe two directions of alignment (see Figure 3.16).

[3] We define a *breakpoint* as a change in direction within the piece-wise linear model.

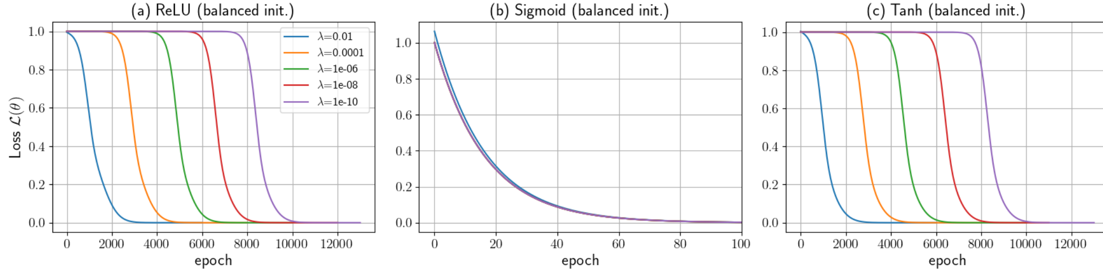



FIGURE 3.14: Training loss function for one-hidden layer model with 60 neurons, $\alpha = 10^{-3}$. Comparison with respect to the activation function, the initialization type and scale (in rich regime).

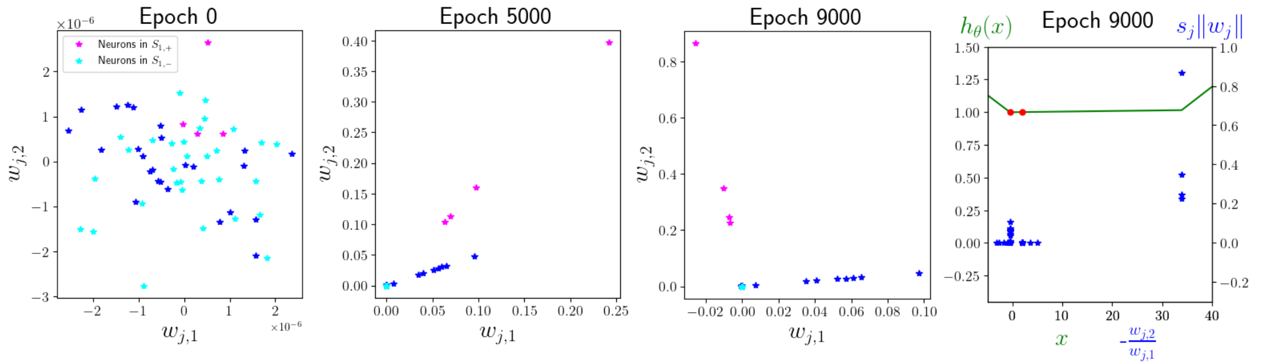



FIGURE 3.15: (Three first windows) Neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-6}$. (Right window) Corresponding model $h_\theta(x)$ and neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$ at the end of the training.

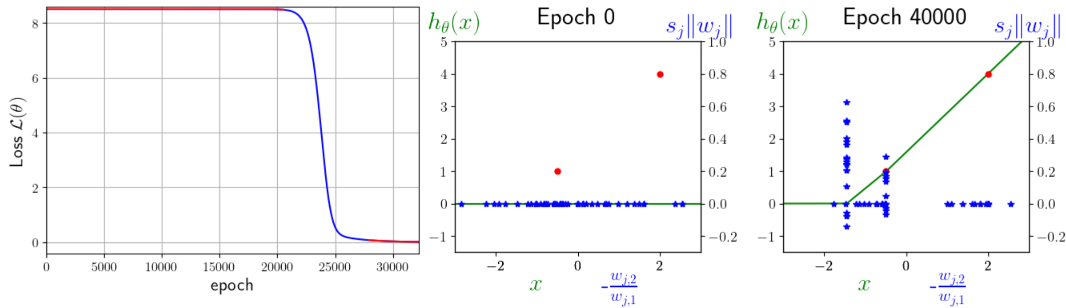



FIGURE 3.16: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-10}$.

For Sigmoid, only the lazy regime is observed when both labels are positive, given the hyperparameter settings used in this work. For Tanh, there is no fundamental difference in the training process whether the labels have opposite signs or the same signs.

### 3.1.3 Dataset 1.1.4 : alternative orthogonal data

Finally, we use two alternative unidimensional orthogonal datasets. Since we obtain similar results for both, we decide to present the results for the following dataset:

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -0.1 & 1 \\ 10 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -c \\ c \end{bmatrix}$$

where the label factor $c > 0$ is a scalar. Additional results for this dataset and the other alternative dataset can be found in Appendix C.

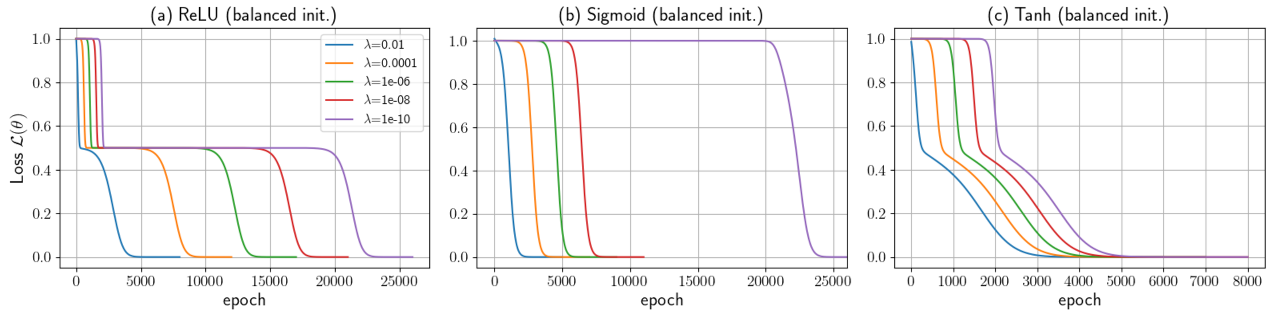



FIGURE 3.17: Training loss function for one-hidden layer model with 60 neurons, $\alpha = 10^{-3}$. Comparison with respect to the activation function, the initialization type and scale (in the rich regime). The label factor is $c = 1$.

For the ReLU activation, we obtain similar results as for Dataset 1.1.0 (see Section 3.1.1). It is worth mentioning that the speed at which the four phases of training occur depends on the values of $x_k$ and $y_k$ and $\lambda$ as reported on the Timeline 2.6. The dependency on $-\ln(\lambda)$ is clear in Figure 3.17(a). We can also observe the dependency of the dataset on the speed of the different phases. The first fast convergence part of the curve, corresponding to positive label fitting, is inversely proportional to $y_2 x_2 = 10$. We see that this phase progresses much faster than the negative label fitting phase (the second fast convergence part of the curve, just after the intermediate plateau), which is inversely proportional to $y_1 x_1 = 0.1$. These experimental results align with the theoretical results of [5].

For the Tanh, we also observe a difference when using $x_k$ values that differ significantly. On Figure 3.17(c), the first half of the fast convergence part of the loss curve progresses much faster than the second half. This behavior can be attributed to the fact that, like ReLU, Tanh ANNs first fit the positive label, then the negative one. The speed of these two phases is likely proportional to the absolute value of their corresponding $y_k x_k$. This is not observed for Sigmoid ANNs, which fit the positive and negative labels simultaneously.

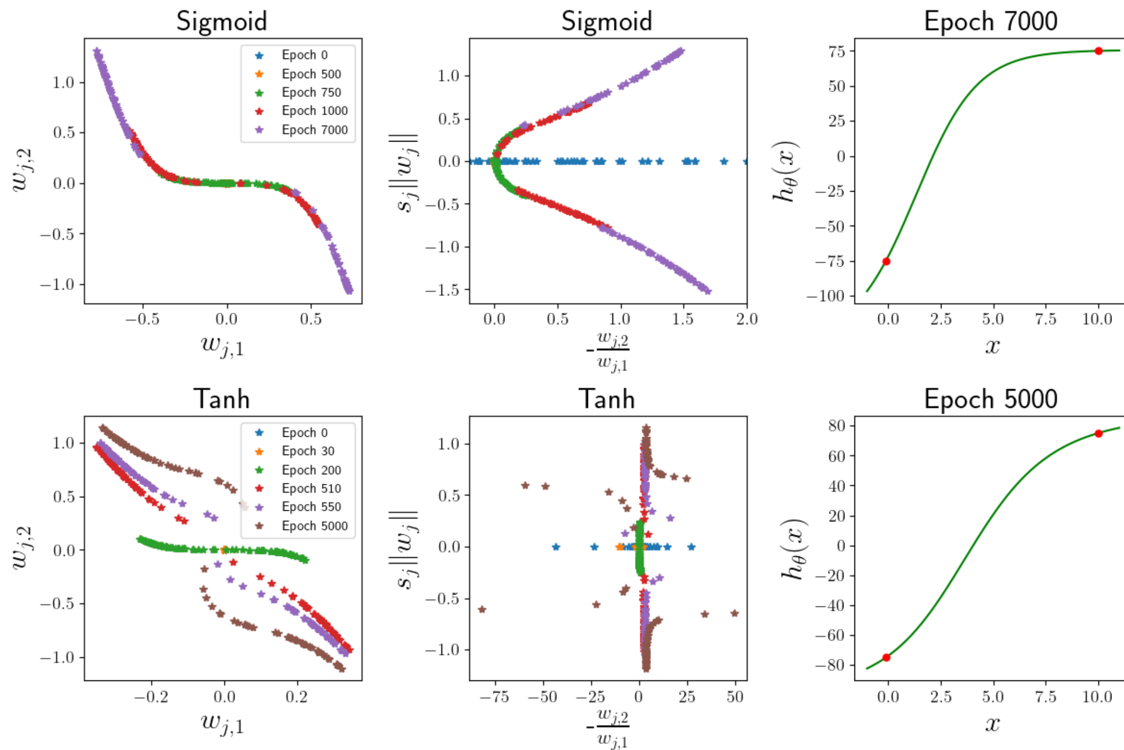



FIGURE 3.18: Training of one-hidden layer Sigmoid/Tanh ANNs with $m = 100$ for different epochs : (Left) neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ ; (Middle) neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$ ; (Right) Final interpolator $h_\theta(x)$. The label factor is $c = 75$.

Using a higher value of $c$, the neurons should grow more in norm, allowing us to observe more clearly the shapes the neurons form in different representations. In Figure 3.18, we observe the training for Sigmoid and Tanh ANNs. We superimpose the neuron dispositions from several epochs of the loss curve to see how they evolve during training. For Sigmoid, in the $(\frac{-w_{j,2}}{w_{j,1}}, s_j\|w_j\|)$ plane, we see clearly that neurons draw a bell shape inverted horizontally, and not a part of an ellipse as thought with previous numerical experiments. Interestingly, the neuron disposition forms an S-curve[4] in the plane, passing through the origin $(0, 0)$. We observe a similar pattern for Tanh ANNs[5].

We now summarize our observations on the training dynamics of one-hidden layer ANNs with Sigmoid or Tanh activation functions, and for orthogonal data minimizing the square loss. For the Sigmoid activation function, the training dynamics appear to follow these phases :

1. **Alignment Phase** : Neurons first align in one direction, maintaining a low norm.
2. **Labels Fitting Phase** : Neurons increase in norm to fit data with positive and negative labels. As they grow, neurons slightly adjust their direction, following an S-curve only dependent on the dataset.

[4] Sigmoid and Tanh are particular S-curves.

[5] In Figure 3.18 (lower middle), the upper part of the bell formed by the final epoch (brown stars) appears to shift left of the bell's main body. This is due to the orientation of the symmetry axis of the S-curve in the $(w_{j,1}, w_{j,2})$ plane. If we perform a well-chosen rotation of the axes, we can retrieve a non-broken bell shape.

3. **Final Convergence Phase** : Finally, we observe a convergence to zero loss.

The speed of these phases is proportional to $-\ln(\lambda)$, and inversely proportional to the norm of a value that depends on the data (which is difficult to deduce from our experiments).

For the Tanh activation function, the training dynamics seem to follow these phases :

1. **Alignment Phase** : Neurons initially align in one direction, maintaining a low norm.
2. **Positive Labels Fitting Phase** : Neurons increase in norm to fit the positive label data, adjusting their direction to follow an S-curve dependent on the dataset.
3. **Negative Labels Fitting Phase** : After fitting the positive label data, neurons continue to grow in norm and drastically change their direction while following a deformed S-curve[6] until they fit the negative label data.
4. **Final Convergence Phase** : Finally, we observe a convergence to zero loss.

For Tanh, we observe the same dependency on $\lambda$ and the data for the speed of the four phases as mentioned in [5] for ReLU activation function.

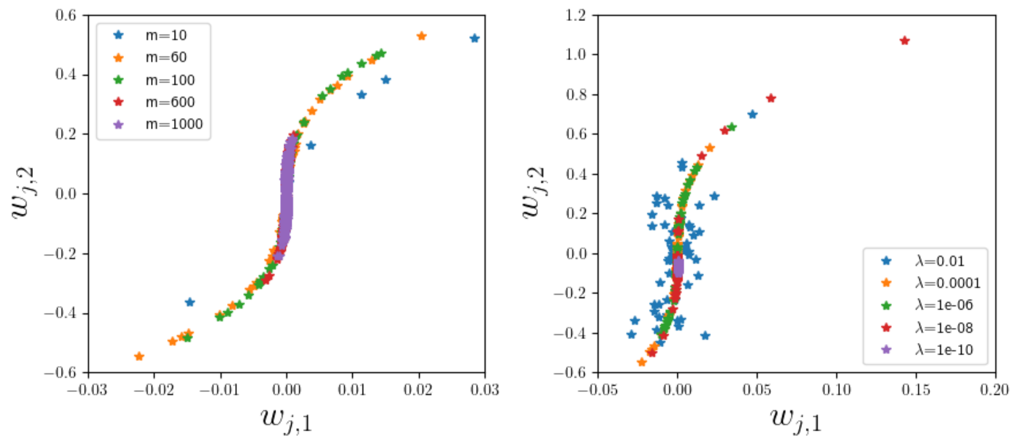



FIGURE 3.19: One-hidden layer Sigmoid ANN with : (Left) different values of $m$ and a fixed $\lambda = 10^{-6}$; (Right) different values of $\lambda$ and a fixed $m = 60$.

Moreover, the initialization scale $\lambda$ influences the variance of the neurons around the S-curve they trace, similar to the ReLU activation function. We also observe this behavior for Tanh. Additionally, we notice the impact of network width $m$ on the final hidden weights matrix $W^t_{\text{GD}}$. As with ReLU, we observe that, the more neurons present in the hidden layer, the less the neurons need to grow in norm to fit the data.

We can wonder why we observe S-curves in $(w_{j,1}, w_{j,2})$ plane and their corresponding bell shapes in $\left(\frac{-w_{j,2}}{w_{j,1}}, s_j \|w_j\|\right)$ plane. We attempt to answer this question in the following section.

[6] We will later see that this curve may correspond to a sum of Tanh functions.

### 3.1.4 S-curve of neurons in $(w_1, w_2)$ plane

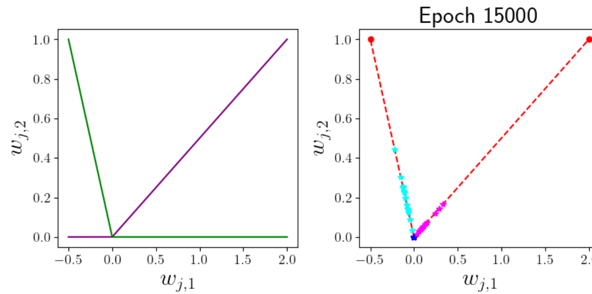



FIGURE 3.20: Example of neuron disposition in $(w_{j,1}, w_{j,2})$ plane which is a linear combination of two ReLU functions. (Left) Two ReLU functions, (Right) Neuron alignment for the final epoch of training drawn by blue, magenta and aqua blue stars, the directions $D_+$ and $D_-$ of alignment drawn by the red dashed line, and the input data $X$ drawn by the red dots.

We aim to characterize the final interpolator found by GD for different activation functions. For ReLU, as seen in Figure 3.20, the neurons align along two directions corresponding to the direction of the data (red points on the figure) as stated in [5]. Up to a planar rotation, the neuron disposition in space $(w_{j,1}, w_{j,2})$ can be viewed as a linear combination of two ReLU functions (left window of Figure 3.20). Moreover, as discussed in the previous section, we can observe that the neuron disposition for Sigmoid and Tanh activation functions draws an S-curve[7] in the plane, passing through the origin $(0, 0)$.

Based on these observations, we assume in this section that, up to a rotation in the $(w_{j,1}, w_{j,2})$ plane, the neurons of one-hidden layer Sigmoid/Tanh ANNs draw a shape in $(w_{j,1}, w_{j,2})$ plane that is a linear combination of two Sigmoid/Tanh functions respectively. Now, we analyze the neuron disposition in the parameter space for the Sigmoid activation function. The analysis for Tanh can be found below this one. We will try to fit a sum of Sigmoid $f(w)$ with $w \in \mathbb{R}$ to the neuron S-curve, and translate this into the $(\frac{-w_{j,2}}{w_{j,1}}, s_j \|w_j\|)$ plane to test our assumption. Each stage of this curve fitting process is illustrated in Figure 3.21. The function to fit the neuron S-curve can be expressed as follows :

$$f(w) = K_1 \text{ Sigmoid}(k_1 w) + K_2 \text{ Sigmoid}(k_2 w) \tag{3.1}$$

with $K_1, K_2, k_1, k_2 \in \mathbb{R}$.

[7] Sigmoid and Tanh are particular S-curves.

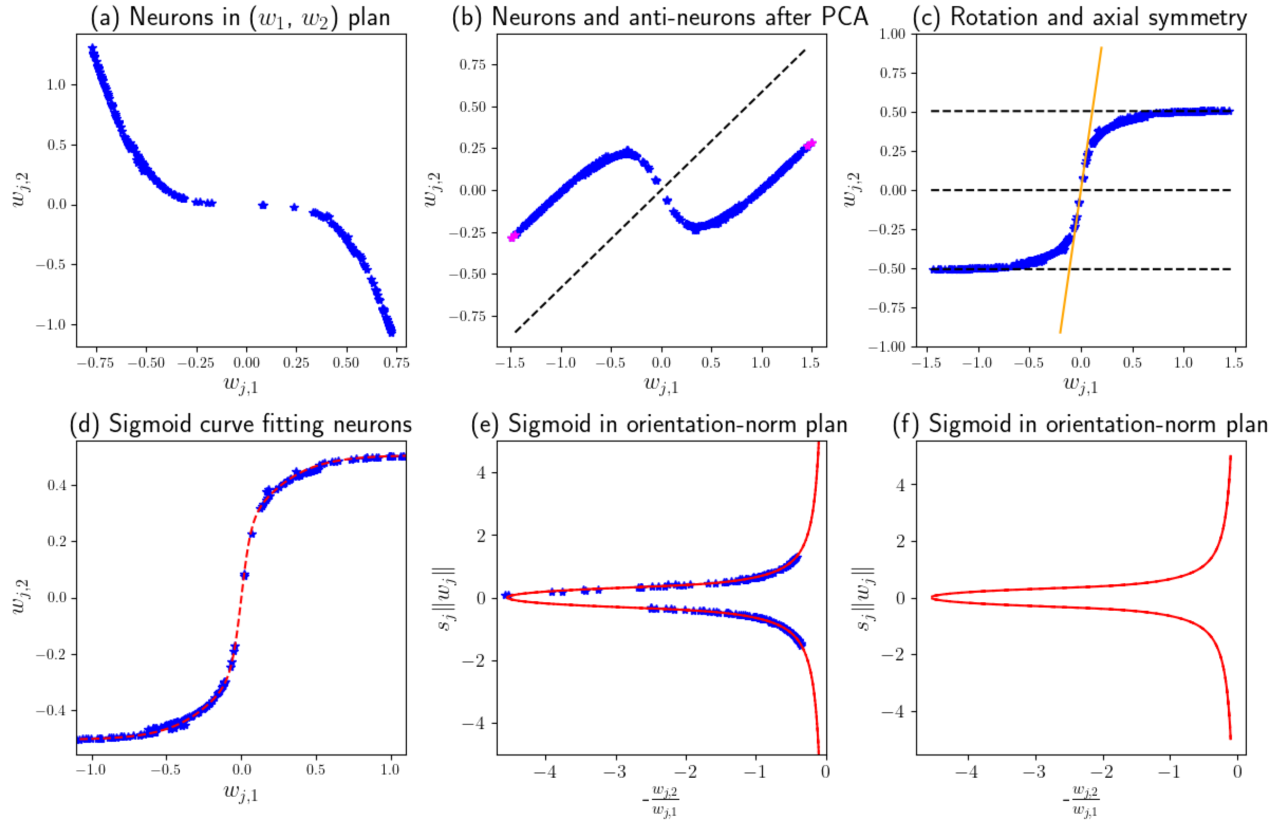



FIGURE 3.21: Curve fitting of $f(w)$ on the neuron S-curve in $(w_1, w_2)$ plane for a Sigmoid ANN trained on Dataset 1.1.4 with $c = 75$. (a) Neuron S-curve composed of several $W^t$. (b) Neurons and anti-neurons[8]pre-rotated with PCA. (c) Final rotation of the plane and axial symmetry. The black dotted lines are the axis of symmetry and the asymptotes of the S-curve, and the orange line is the tangent at the origin. (d) Curve fitting (red dotted line) of the neuron S-curve. (e) Equivalent representation in the orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$ (zoomed on the neurons). (f) Equivalent representation of $f(w)$ in the orientation-signed norm representation.

Starting from the final $\theta^*_{GD}$ found by GD (window (a))[9], our goal is to rotate the plane so that the axis of symmetry of the S-curve aligns with the $w_{j,1}$-axis. To achieve this, we perform a PCA on the plane to avoid special cases and we duplicate the neurons by performing a central symmetry rotation, ensuring a perfect symmetry of the data to fit (window (b)). We approximate the axis of symmetry passing through the origin (black dashed line) by computing the derivative of the linear lines passing through the two most distant neuron pairs (in magenta), and averaging these two values. In window (c), you observe the final rotated plot[10]. From this plot, we approximate the derivative $k$ of the S-curve at the origin (in orange) and the values of the two asymptotes $w_{j,2} = \pm K$ of the S-curve. Because of the positiveness of the Sigmoid function and the symmetry of the S-curve, we know that $K = -K_1 = K_2$. Additionally, $f'(0) = \frac{K_1 k_1}{4} + \frac{K_2 k_2}{4} = k$. So, by choosing one of the parameters $k_1$ or $k_2$, we can determine the other. In the example, we choose $k_1 = -5$. The resulting $f(w)$ is

[8]We refer to "anti-neurons" as the images of neurons under central symmetry.

[9]Since $\theta^t$ lies on the same S-curve for every $t$ with the Sigmoid activation function, we can merge several $\theta^t$ to increases the number of neuron positions and improve curve fitting. In the example, we add $\theta^{1000}$ to the final $\theta^*_{GD}$ to include neurons near the origin.

[10]With an axial symmetry to orient the S-curve in the same direction as the classical Sigmoid, although it is not an obligation.

plotted in red in the $(w_{j,1}, w_{j,2})$ plane (window (d)), and in the $(\frac{-w_{j,2}}{w_{j,1}}, s_j\|w_j\|)$ plane (windows (e) and (f)).
Here, the fit appears very good. However, this is not always the case, as shown by examples in Appendix C.1.5, where we can observe that the resulting $f(w)$ does not perfectly fit the data. This is probably due, in part, to the fitting method.

How can we theoretically explain that the neuron disposition in $(w_{j,1}, w_{j,2})$ plane draws an S-curve, which is likely a sum of Sigmoid functions, at the end of training with a small initialization scale ? We now propose an intuitive explanation. For simplicity, let us assume that the final interpolator $h_\theta(x)$ is almost a linear between the two input data $x_k$ with small initialization scale. Since $h_\theta(x) = \sum_{j=1}^m a_j\sigma\left(w_j^T x\right)$, we need $z_j = \sigma\left(w_j^T x\right)$ to be linear, implying $w_j^T x = \sigma^{-1}\left(z_j\right)$. Because $\sigma^{-1}$ is essentially $\sigma$ up to a an axial symmetry, we could recover our S-curve in the $(w_1, w_2)$ plane. However, although the assumption of a linear interpolator between input data holds for Dataset 1.1.0, it does not hold for Dataset 1.1.4 (see Figure 3.18 upper right). It suggests that, to achieve a perfect curve fitting, we need to use a function of the form $f(w) = K_1 \text{Sigmoid}(k_1 g(w)) + K_2 \text{Sigmoid}(k_2 g(w))$, where $g$ is a nonlinear function.

Below, we provide the shapes of the two terms of $f(w)$ used for curve fitting. This combination seems to be the most effective for accurately fitting $f(w)$ to the S-curve drawn by the neurons.

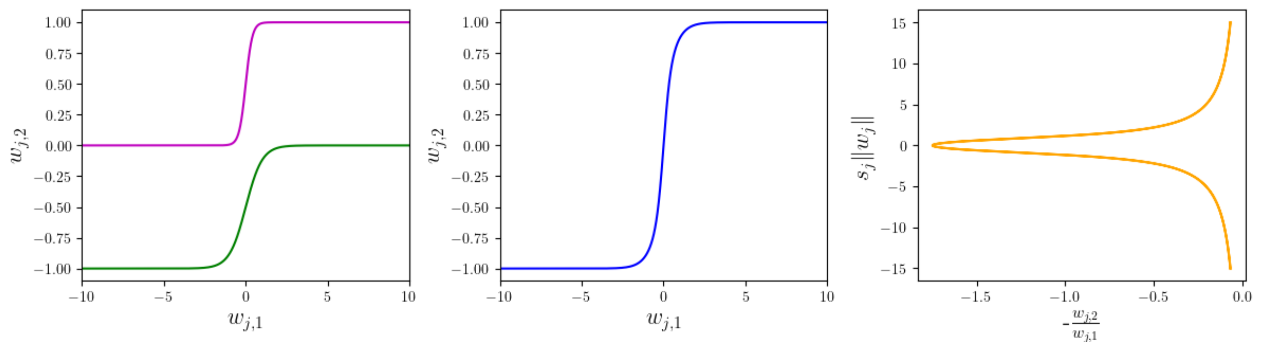



FIGURE 3.22: Illustration of neuron disposition in $(w_1, w_2)$ plane which is a linear combination of two Sigmoid functions, and its equivalent representation in the orientation-signed norm representation. (Left) Plots of $f_1(w) = K_1 \text{ Sigmoid}(k_1 w)$ and $f_2(w) = K_2 \text{ Sigmoid}(k_2 w)$, (Middle) plot of $f(w) = f_1(w) + f_2(w)$, (Right) equivalent representation of $(w, f(w))$ in $(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|)$.

For Tanh, fitting the curve is more complex. By adapting the method explained above and using the relationship between Sigmoid and Tanh (i.e. $\text{Sigmoid}(x) = (1 + \text{Tanh}\left(\frac{x}{2}\right))/2$), we attempt to fit the following function to the neuron curve :

$$f(w) = \frac{K_1}{2}\text{Tanh}\left(\frac{k_1}{2}w\right) + \frac{K_2}{2}\text{Tanh}\left(\frac{k_2}{2}w\right) \tag{3.2}$$

$$= K_1\left(\text{Sigmoid}(k_1 w) - \frac{1}{2}\right) + K_2\left(\text{Sigmoid}(k_2 w) - \frac{1}{2}\right), \tag{3.3}$$

which results in Figure 3.23.

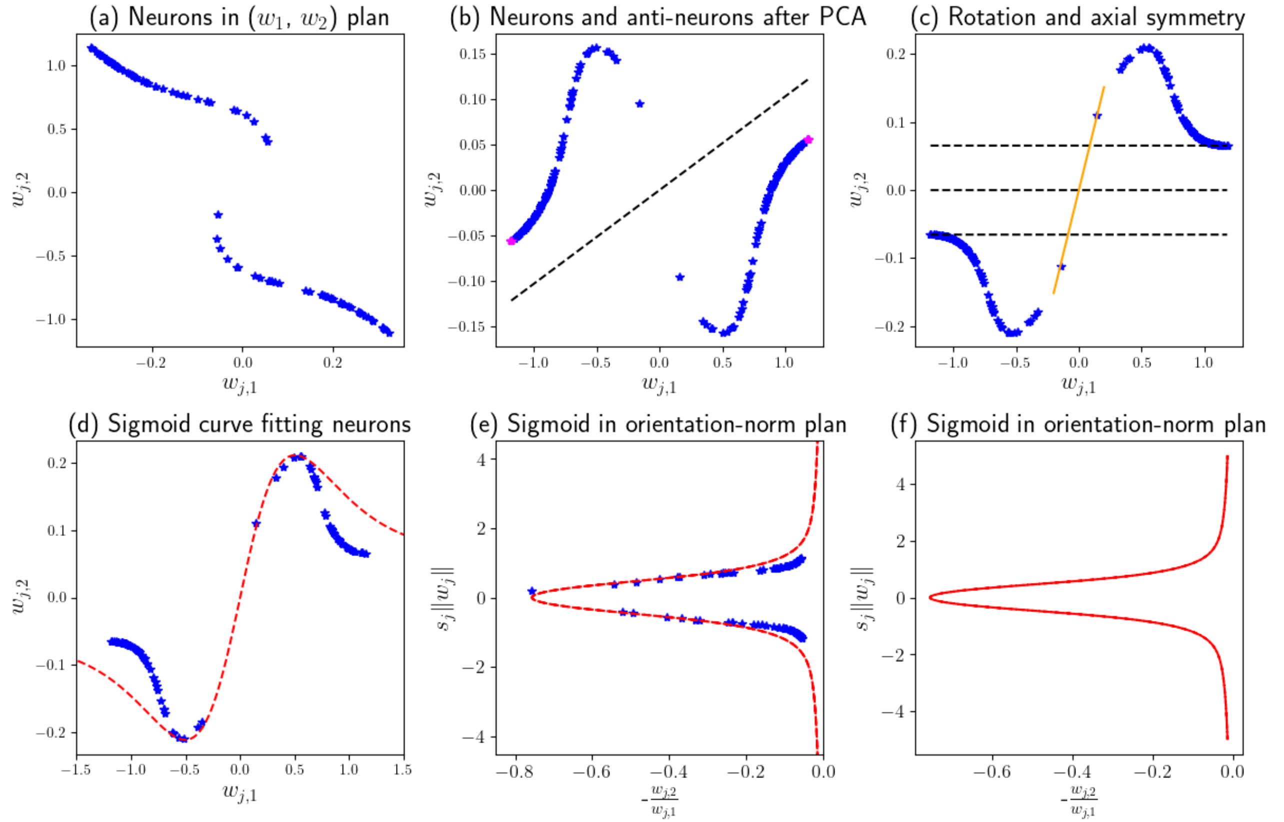



FIGURE 3.23: Curve fitting of $f(w)$ on the neuron S-curve in $(w_1, w_2)$ plane for a Tanh ANN trained on Dataset 1.1.4 with $c = 75$. (a) Neuron S-curve composed of several $W^t$. (b) Neurons and anti-neurons pre-rotated with PCA. (c) Final rotation of the plane and axial symmetry. The black dotted lines are the axis of symmetry and the asymptotes of the S-curve, and the orange line is the tangent at the origin. (d) Curve fitting (red dotted line) on the neuron S-curve. (e) Equivalent representation in the orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$ (zoomed on the neurons). (f) Equivalent representation of $f(w)$ in the orientation-signed norm representation.

From our experiments, we can deduce the following combinations of Tanh functions that we have encountered. The first combination appears during the positive label fitting phase in the training of Tanh ANNs. As the training progresses from fitting positive labels to fitting negative labels, we observe the second combination (which is the one chosen to fit the S-curve in the previous example). Lastly, we provide a third combination that we encountered in numerical experiments with non-orthogonal datasets.

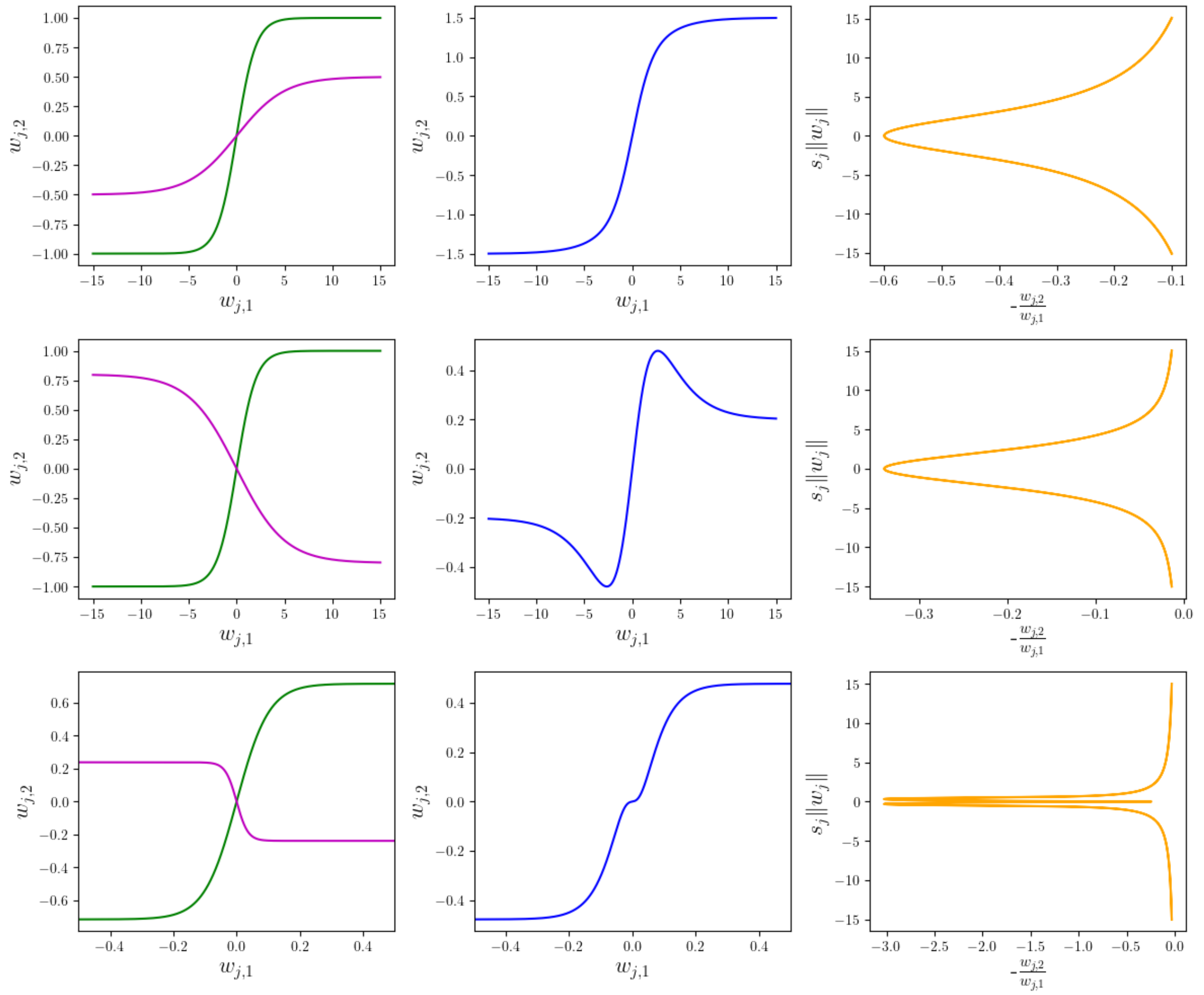


FIGURE 3.24: Illustration of three neuron dispositions (one by row) in the $(w_{j,1}, w_{j,2})$ plane which are linear combinations of two Tanh functions, and their equivalent representation in the orientation-signed norm representation.

## 3.2 Unidimensional quasi-orthogonal data

Since they argue in [5] that their results hold for quasi-orthogonal data when $\delta$ is of order $\lambda$, we use the following dataset :

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -0.5 \pm \frac{\delta}{2} & 1 \\ 2 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

which ensures that $|x_1^T x_2| \leq \delta$.

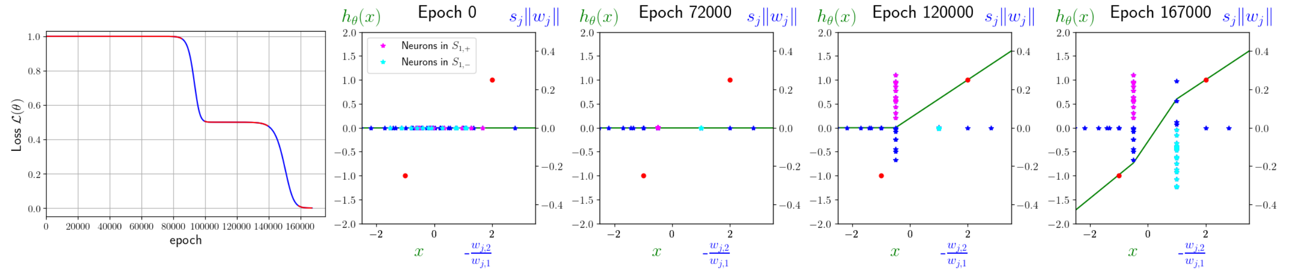


FIGURE 3.25: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-10}$. Quasi-orthogonal input data with $\delta = 1$.

We discuss the case of quasi-orthogonal data for ReLU. For all tested values of $\delta$ where one $x_{k,1}$ is positive and the other negative, the training loss curve profile remains unchanged, even for very high value of $\delta$[11]. We observe saddle-to-saddle dynamics with initial and intermediate plateaus, as well as the four phases presented in [5] for orthogonal data. However, with quasi-orthogonal data, neuron alignment directions differ from those of the data, leading to breakpoints in the interpolator $h_\theta(x)$ that do not align with the data. Indeed, from our observations, we can state that, for unidimensional data with $n = 2$ with bias element, opposite signed inputs and opposite signed labels, the alignment directions are still $D_+ = y_k x_k$ for $y_k > 0$ and $D_- = y_k x_k$ for $y_k < 0$ as stated in [5] for orthogonal data, corresponding to

$$\tilde{D}_+ = -\frac{x_{k,2}}{x_{k,1}} \quad \text{for } y_k > 0,$$
$$\tilde{D}_- = -\frac{x_{k,2}}{x_{k,1}} \quad \text{for } y_k < 0$$

in the plane $(\frac{-w_{j,2}}{w_{j,1}}, s_j\|w_j\|)$. For example, recalling that $x_{k,2} = 1$ for all $k$, if $x_{1,1} = -1$ and $x_{2,1} = 2$ with $y_1 = -1$ and $y_2 = 1$, the two directions of alignment are $\tilde{D}_+ = 1$ and $\tilde{D}_- = -0.5$ (It is what we observe on Figure 3.25). Orthogonal data with opposite signed labels are just a particular case where $x_{1,1} = -\frac{1}{x_{2,1}}$. Thus, the alignment direction of $x_{1,1}$ creates a breakpoint in the interpolator at $x = x_{2,1}$, and conversely. This aligns the breakpoints with the data on the different figures. This explains why the authors of [5] suggests that results for orthogonal data can be extended to quasi-orthogonal data with $\delta = \mathcal{O}(\lambda)$, since the direction difference caused by the deviation $\delta$ is merged into the variance around the neuron alignment directions created by $\lambda$. Another difference between quasi-orthogonal data and orthogonal data is that there are both neurons with $s_j = \text{sign}(a_j^0) = \pm 1$ which grow in norm along both alignment directions.

[11] We test $\delta \in \{100, 1, 10^{-1}, 10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}\}$, and some intermediate values when necessary.

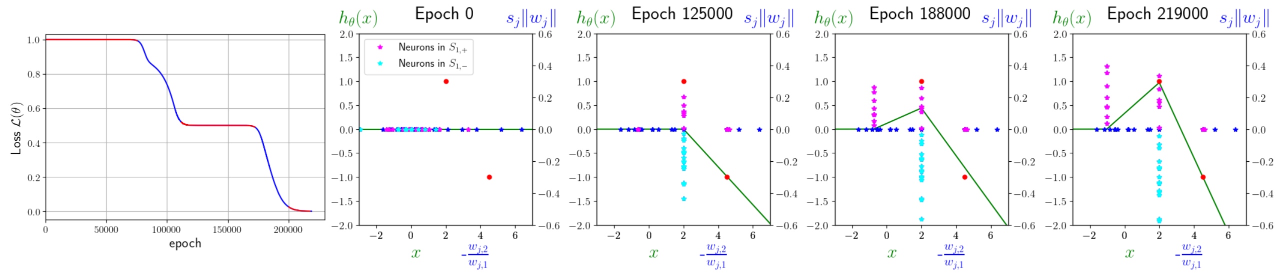


FIGURE 3.26: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-10}$. Quasi-orthogonal input data with $\delta = 10$.

When both input data $x_{k,1}$ are positive, the training dynamics change (See Figure 3.26). A new slowdown in the loss curve appears between the initial and the intermediate plateaus. Additionally, while two directions of neuron alignment are maintained, their dependence on the data is less clear. One direction aligns with the point $k$ having the smaller abscissa $x_k$, and the other remains in the negative part of the axis. Another key difference is that the training does not follow the same phases as stated for orthogonal data. In our example, the model first fits the data with negative labels. After fitting this negative label data, the model temporarily moves away from the data it just fitted when the second group of neurons grows in norm to reach the positive label data. Here, it is clear that this type of data should be considered non-orthogonal rather than quasi-orthogonal.

For Sigmoid and Tanh, we do not observe fundamental differences in the training dynamics compared to orthogonal data.

## 3.3 Unidimensional non-orthogonal data

Now, we experimentally investigate whether the behaviors we observed for orthogonal data are still valid for non-orthogonal data. Before, we notice that, for small values of $\lambda$, we often encountered vanishing gradient before final convergence after several millions of epochs. This vanishing often occurred at the start of the training, especially for Sigmoid activation function. Therefore, in this section, we used $\alpha < 10^{-3}$ for dataset with $n = 5$, and $\alpha < 10^{-4}$ for datasets with $n = 10$ when it was possible. Additionally, we used $\lambda \geq 10^{-6}$ (or greater value if we encountered vanishing gradient anyway). We also lowered our accuracy goal to $10^{-3}$ for dataset with $n = 5$ and to $10^{-5}$ for dataset with $n = 10$.

### 3.3.1 Dataset 1.3.3 : non-orthogonal data with alternate labels

For our first non-orthogonal dataset, we use five data with equidistant abscissas and alternating labels. More precisely,

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} -2 & 1 \\ -1 & 1 \\ 0 & 1 \\ 1 & 1 \\ 2 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}.$$

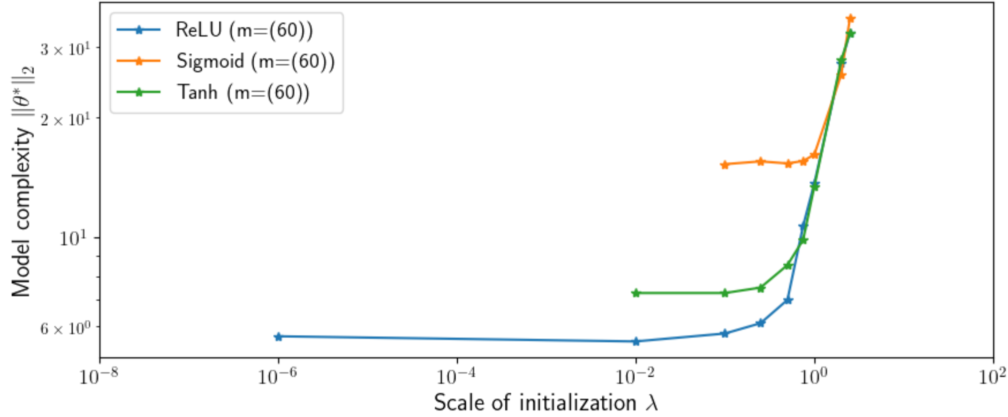



FIGURE 3.27: Model complexity with respect to $ell_2$-norm (i.e. $\|\theta^*_{\text{GD}}\|$) in function of $\lambda$ for a width of 60 neurons and a balanced initialization for non-orthogonal input data.

On the figure above, we show that, even for non-orthogonal data with $n > d$, the phenomenon of reduced model complexity when decreasing the initialization scale $\lambda$ persists, similar to what is observed with orthogonal data for the three activation functions. However, as the number of data points $n$ increases, gradient vanishing often occurs, preventing us from investigating this behavior for very small $\lambda$.

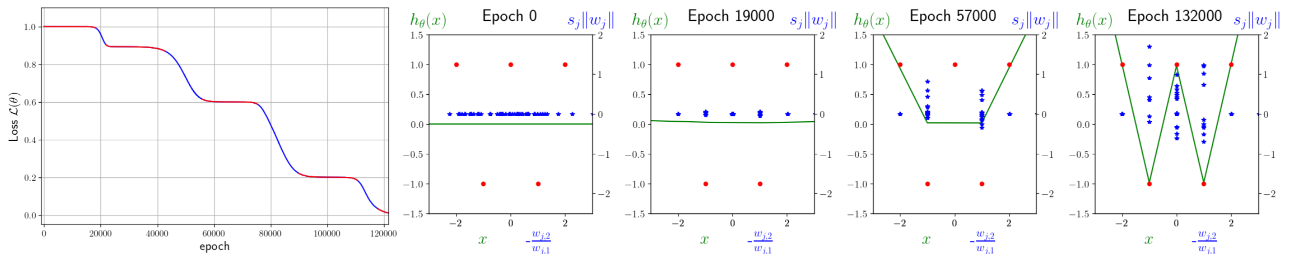



FIGURE 3.28: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-6}$.

For ReLU, we observe that for small $\lambda$, neuron alignment occurs at a finite number of directions corresponding here to the data's abscissas. The final interpolator looks like the linear interpolation of the data, as mentioned in [34]. We also observe training dynamics closed to those for orthogonal data. First, there is an alignment phase in which the neurons stay small in norm. Secondly, we observe that the norms of certain groups of neurons grow to fit specific data, while others remain small. Once these data points are fitted, different groups of neurons increase in norm, and this process repeats until the model fits all the data. This results in a training loss curve characterized by alternating parts of fast and slow convergence, showing a saddle-to-saddle dynamic.

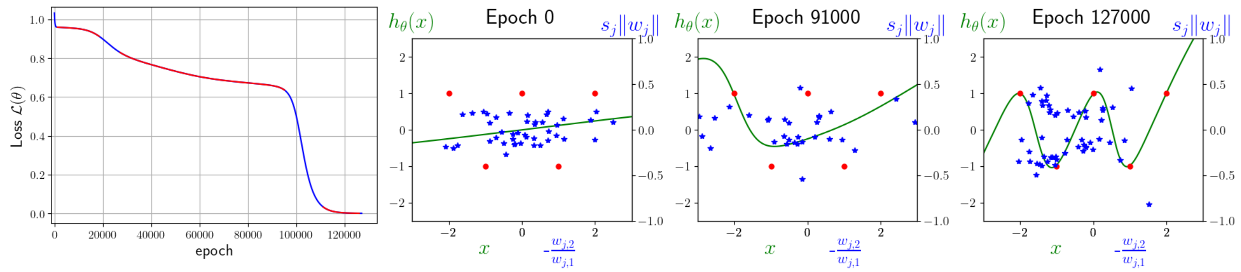



FIGURE 3.29: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a Tanh one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-1}$.

For Sigmoid and Tanh, we do not observe clear bell shape in $(\frac{-w_{j,2}}{w_{j,1}}, s_j\|w_j\|)$ plane or S-curve in $(w_{j,1}, w_{j,2})$ plane for the values of $\lambda$ that allow successful training with no premature vanishing. However, as we can observe on Figure 3.29 for Tanh, the neurons for Tanh form a shape that might be the start of a bell at the end of training, but further investigation is needed to confirm this. Similar to ReLU, the training loss curve exhibits saddle-to-saddle dynamics.

### 3.3.2 Dataset 1.3.2 : non-orthogonal data with random labels

For our last unidimensional non-orthogonal dataset, we randomly generated 10 input data points between -2 and 2, and we assigned a label of 1 or -1. The dataset is shown below.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{bmatrix} = \begin{bmatrix} 0.99864822 & 1 \\ -0.58678793 & 1 \\ -1.19866293 & 1 \\ 1.59530385 & 1 \\ -0.96831112 & 1 \\ -1.69267688 & 1 \\ 0.13815776 & 1 \\ -0.30366272 & 1 \\ 0.31415876 & 1 \\ 1.93938106 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

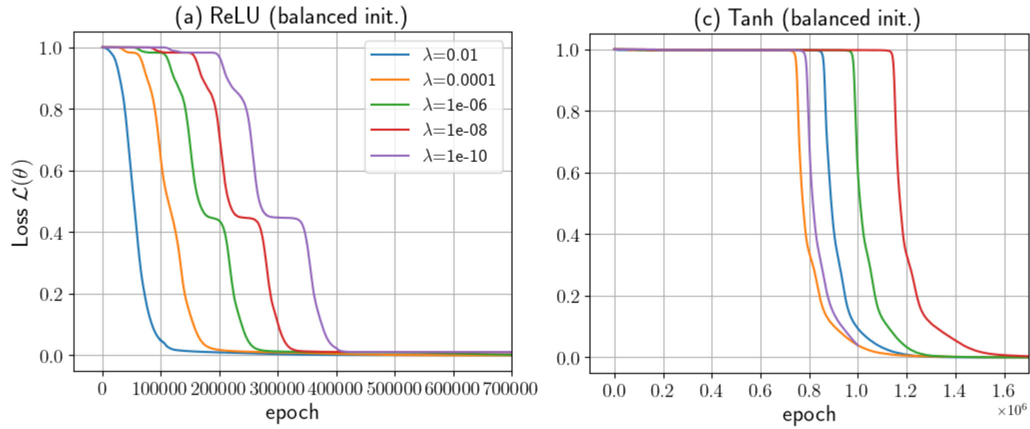



FIGURE 3.30: Training loss curves for one-hidden layer model with 60 neurons, $\alpha = 10^{-3}$. Comparison with respect to the activation function, the initialization type and scale (in rich regime).

On Figure 3.30, we plot the training loss curve for different values of $\lambda$, similar to what was done for orthogonal data[12]. We observe the same dependence on $-\ln(\lambda)$ in the speed of convergence as with orthogonal data. Additionally, increasing $\lambda$ leads to the disappearance of the intermediate plateau. Combined with the previous observations on model complexity, we see the existence of the same three regimes: Lazy, Rich, and the transition regime between them. In Appendix C.3.2, you can also find the plot of training loss curves for different values of $m$. The same observations as for orthogonal data can be made.

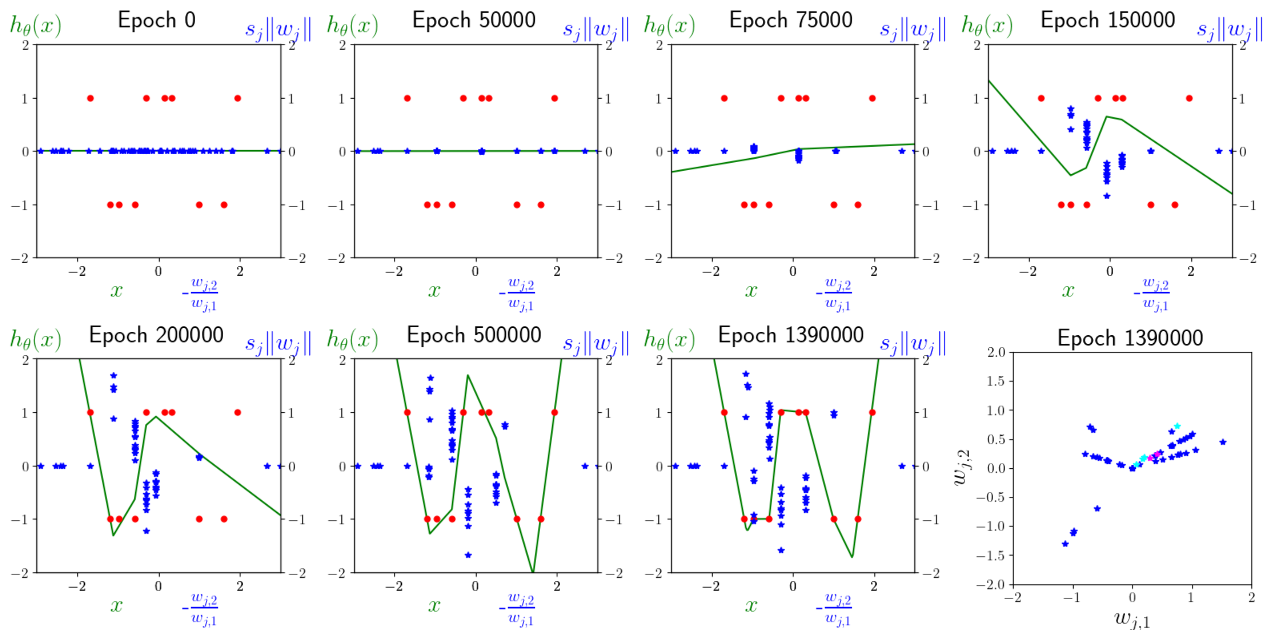



FIGURE 3.31: Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-6}$ for non-orthogonal data. We also plot the final neuron alignment in the $(w_{j,1}, w_{j,2})$ plane in the last window.

[12]There is no data for Sigmoid, as the loss did not decrease at all after $10^6$ epochs for $\lambda = 10^{-6}$.

For ReLU ANNs trained on non-orthogonal data, an interesting phenomenon is observed in Figure 3.31. By comparing several epochs, we can see that some groups of aligned neurons change their direction together during training. This behavior differs from the one described in [34], which states that neurons first align while remaining small in norm and then grow in norm while maintaining their orientation.

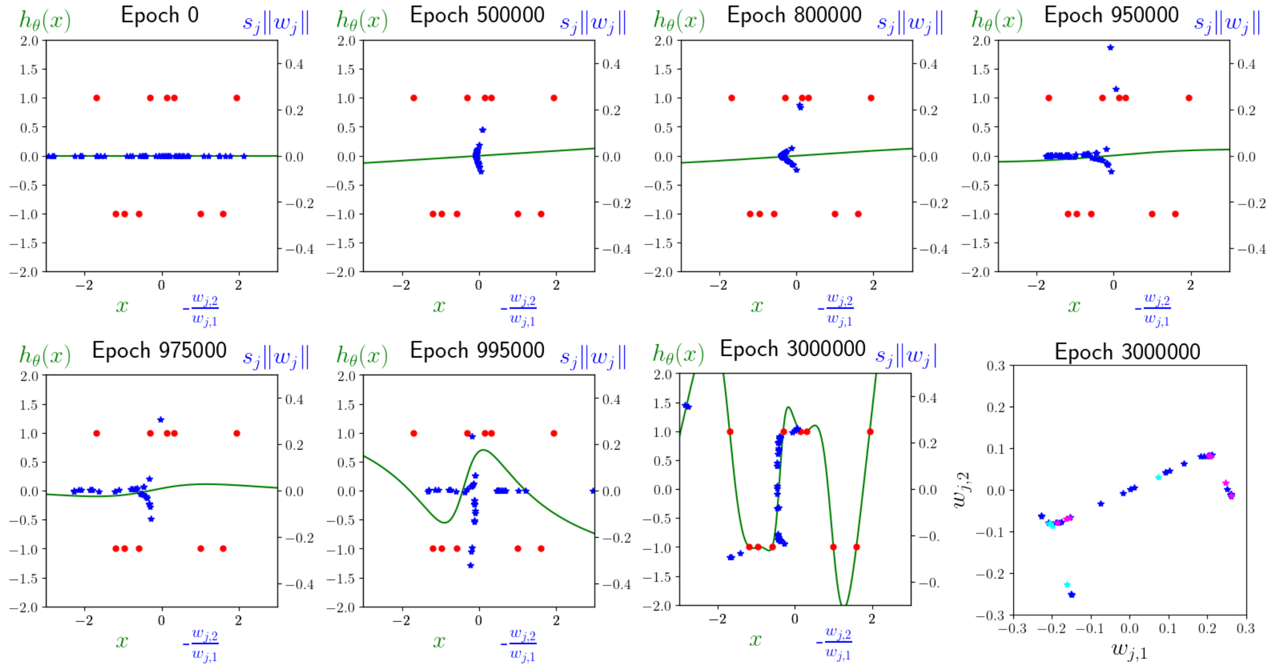



FIGURE 3.32: (Seven first windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a Tanh one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-6}$ for non-orthogonal data. (Last window) Neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ for the final epoch.

For Tanh ANNs, the initial phases of training follow the same pattern as for orthogonal data (see Figures 3.32. Initially, neurons align together in a single orientation while remaining small in norm (Epochs < 400,000). Then, they grow in norm, forming the bell shape described earlier (Epochs < 950,000). Afterward, the training dynamics change, deforming the bell to perfectly fit the data. On the last window of Figure 3.32, we observe the corresponding neuron disposition in the $(w_{j,1}, w_{j,2})$ plane that forms a deformed S-curve at the end of the training.

In Appendix C.3.1, you can find figures showing results for another non-orthogonal dataset with $n = 10$, where positive labels correspond to positive $x_k$ and negative labels to negative $x_k$.

## 3.4 Multidimensional data

Finally, we conduct numerical experiments with higher-dimensional data. Here, we present selected results for $d = 5$ and the methods used to generate them. We begin with orthogonal input data and we conclude with non-orthogonal data.

### 3.4.1 Orthogonal

For a fixed dimension $d$, we want to generate a dataset $X$ of $n = d$ data with orthogonal inputs containing a bias term ($x_{k,d} = 1$ for each $k$), and random scalar labels $y$. We follow this process :

1. We generate an orthogonal matrix $X \in \mathbb{R}^{d\times d}$. Then, we divide each row by its last element to ensure $x_{k,d} = 1$.

2. To generate the labels, we use a teaching network as done in [5]. The teaching network is a one-hidden layer ANN with a small number of neurons (we use $m = 10$), and it is set with a random parameter vector $\tilde{\theta}$. We generate $\tilde{\theta}$ from a normal distribution with fixed variance (typically 0.1). We then set the labels as $y_k = h_{\tilde{\theta}}(x_k)$.

Using this method, we obtain the dataset used to generate the figures in this section. Other trials have produced similar results in the training dynamics, but with less readable figures due to differences in scale between positive and negative labels data generated by the methods above.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1.06230214 & 0.31117178 & -0.34853043 & -0.34071425 & 1 \\ -1.60140948 & -1.45217752 & -0.08119978 & -3.30117333 & 1 \\ -0.99692421 & 1.51269658 & 1.08896864 & 0.09431674 & 1 \\ -0.18304326 & -1.14347546 & 0.42595479 & 0.884252 & 1 \\ -2.00910441 & 0.64220423 & -3.74383553 & 1.08712951 & 1 \end{bmatrix}$$

$$y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 0.05644127 \\ 0.01888487 \\ -0.0035589 \\ 0.05251718 \\ -0.07106081 \end{bmatrix}.$$

As observed in the following figures, the training dynamics described for undimensional orthogonal data seem to stay valid for the three activation functions. To visualize neuron dispositions for multidimensional data, we apply PCA to the final weight matrix $W^*_{\mathrm{GD}}$ and we project the neurons on the two principal component $\tilde{w}_{j,1}$, $\tilde{w}_{j,2}$ at each considered epoch. The explained variance ratio, which measures how much of the total variance in the original parameter space $(w_{j,1}, \cdots, w_{j,d})$ is captured by each principal component $\tilde{w}_{j,i}$, indicates that the network effectively reduces to a network with two neurons for the three activation functions. Indeed, the first two dimensions in which we project the neurons fully explain the total variance in the original space. This is very clear for ReLU, but less so for Sigmoid and Tanh, where it might appear that the entire variance is captured by only the first component (as shown in Figures 3.35 and 3.36). However, the explained variance of the second component, although small, is non-zero.

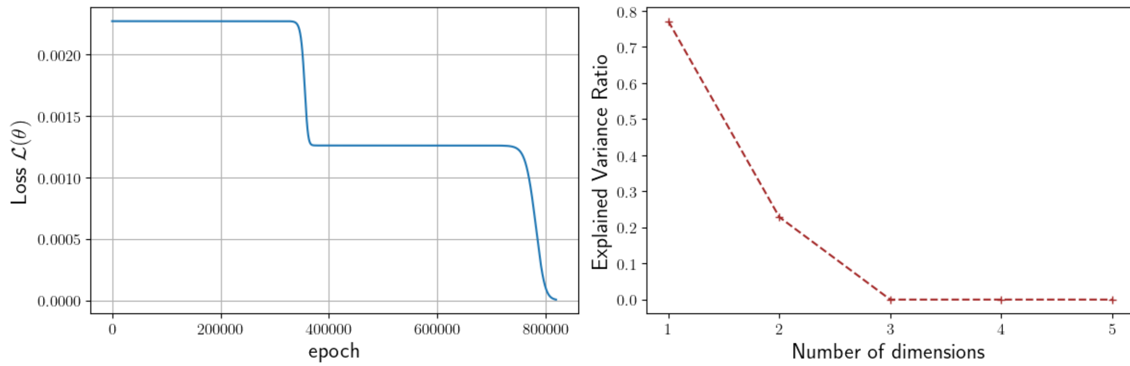



FIGURE 3.33: (Left) Training loss curve for multidimensional orthogonal data ($n = 5$), ReLU network with $m = 200$ and small initialization scale $\lambda$ (here, $\lambda = 10^{-22}$). (Right) Explained variance ratio of the PCA projection fitted on the final weight vector $W^*_{\mathrm{GD}}$.

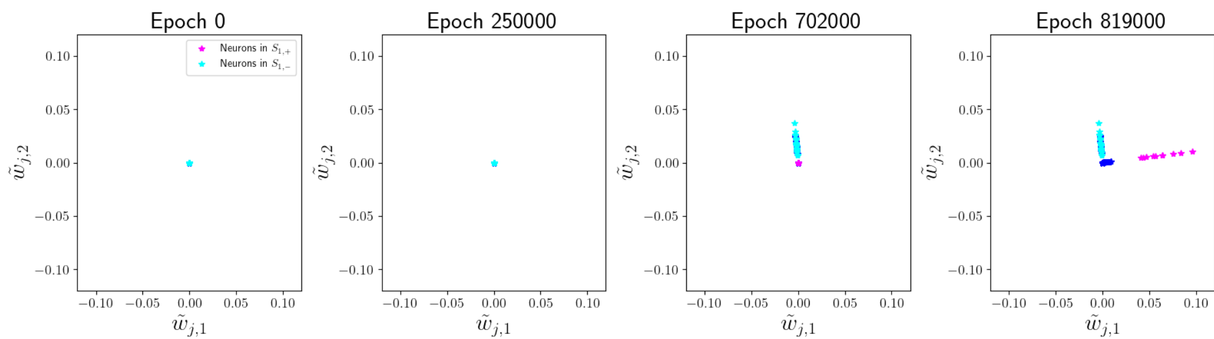



FIGURE 3.34: Neurons in the projected plane $(\tilde{w}_{j,1}, \tilde{w}_{j,2})$ with PCA fitted on the final weight vector $W^*_{\mathrm{GD}}$ for a ReLU one-hidden layer ANN with width 200, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-22}$ for multidimensional orthogonal data.

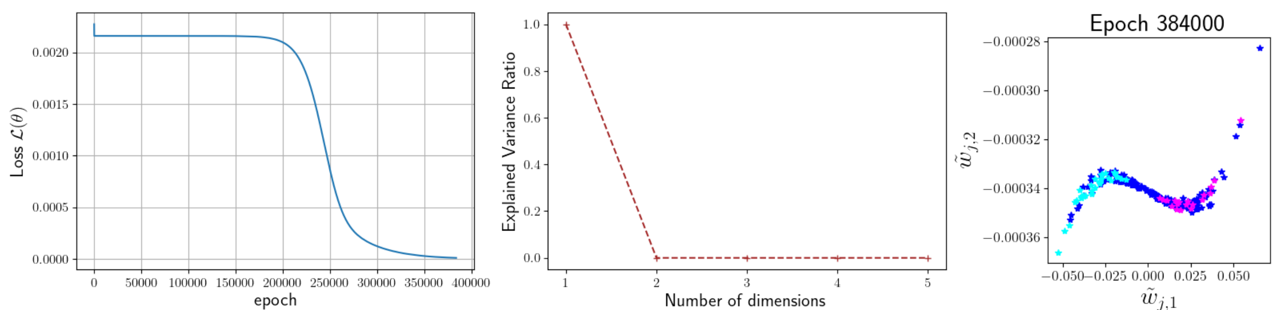



FIGURE 3.35: (Left) Training loss curve for multidimensional orthogonal data ($n = 5$), Sigmoid network with $m = 200$ and small initialization scale $\lambda$ (here, $\lambda = 10^{-6}$). (Middle) Explained variance ratio of the PCA projection fitted on the final weight vector $W^*_{\mathrm{GD}}$. (Right) Final neuron disposition in the $(\tilde{w}_{j,1}, \tilde{w}_{j,2})$ plane.

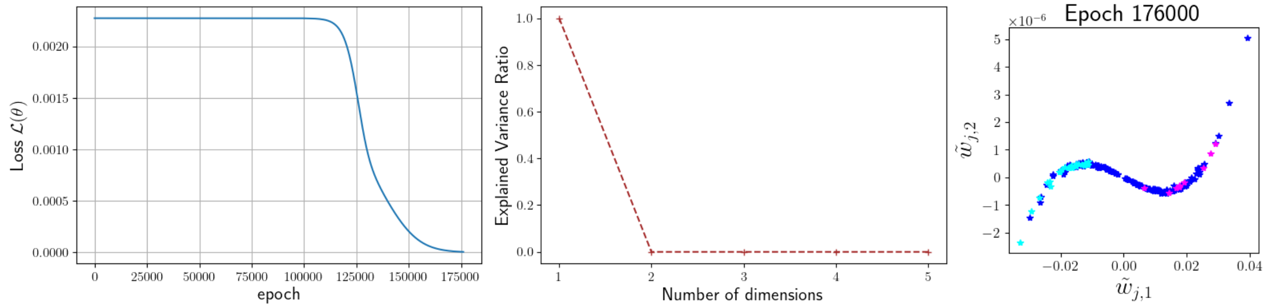



FIGURE 3.36: (Left) Training loss curve for multidimensional orthogonal data ($n = 5$), Tanh network with $m = 200$ and small initialization scale $\lambda$ (here, $\lambda = 10^{-10}$). (Middle) Explained variance ratio of the PCA projection fitted on the final weight vector $W^*_{\text{GD}}$. (Right) Final neuron disposition in the $(\tilde{w}_{j,1}, \tilde{w}_{j,2})$ plane.

### 3.4.2 Non-orthogonal

For comparison, we also conduct numerical experiments with non-orthogonal data. Analyzing the training process is more challenging in this case. However, even though the explained variance ratio for most principal components is very small, it suggests that the reduction to a two-neuron network, observed with orthogonal data, does not hold in higher dimensions with non-orthogonal data.

To generate the dataset for the following results, we created a random weight vector $\theta$ such that $\theta_i \sim \mathcal{U}(-1, 1)$, and we then divided each row by its last element in order to have $x_{k,d} = 1$.

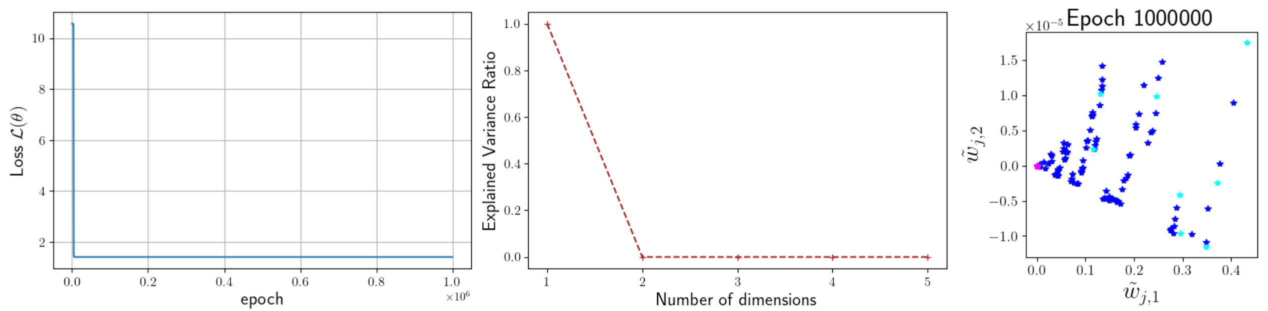



FIGURE 3.37: (Left) Training loss curve for multidimensional non-orthogonal data ($n = 5$), ReLU network with $m = 200$ and small initialization scale $\lambda$ (here, $\lambda = 10^{-12}$). (Middle) Explained variance ratio of the PCA projection fitted on the final weight vector $W^*_{\text{GD}}$. (Right) Final neuron disposition in the $(\tilde{w}_{j,1}, \tilde{w}_{j,2})$ plane.

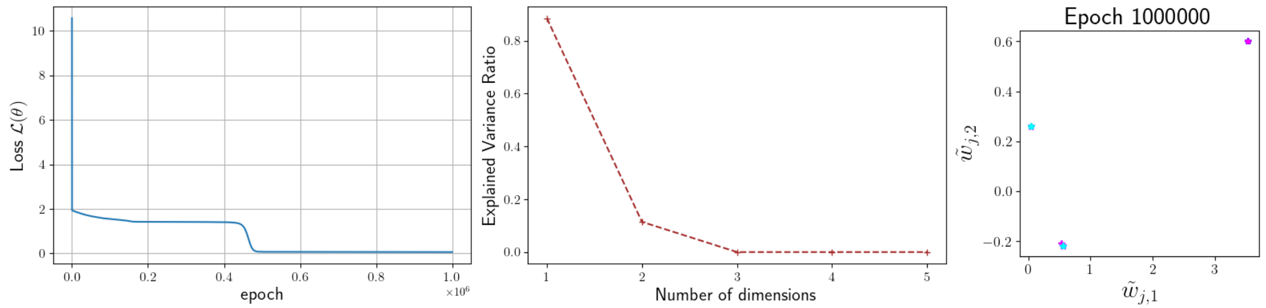



FIGURE 3.38: (Left) Training loss curve for multidimensional non-orthogonal data ($n = 5$), Sigmoid network with $m = 200$ and small initialization scale $\lambda$ (here, $\lambda = 10^{-12}$). (Middle) Explained variance ratio of the PCA projection fitted on the final weight vector $W^*_{\mathrm{GD}}$. (Right) Final neuron disposition in the $(\tilde{w}_{j,1}, \tilde{w}_{j,2})$ plane.

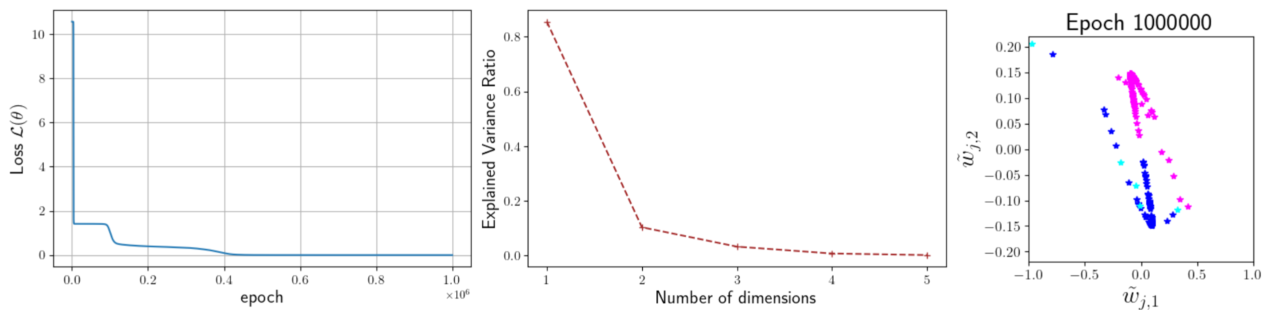



FIGURE 3.39: (Left) Training loss curve for multidimensional non-orthogonal data ($n = 5$), Tanh network with $m = 200$ and small initialization scale $\lambda$ (here, $\lambda = 10^{-12}$). (Middle) Explained variance ratio of the PCA projection fitted on the final weight vector $W^*_{\mathrm{GD}}$. (Right) Final neuron disposition in the $(\tilde{w}_{j,1}, \tilde{w}_{j,2})$ plane.

## 3.5 Summary of Numerical Experiments Results and Conjectures

To summarize, our numerical experiments aimed to discern the limitations of the postulated assumptions of results in [5], which characterize the training of one-hidden ReLU ANNs minimizing square loss with Gradient Flow and using orthogonal input data, under the assumptions that the sets $S_{1,+}$ and $S_{1,-}$ are non-empty and the initialization is balanced.

To achieve this, we first reproduced their experiments for unidimensional orthogonal data, and we verified their consistency with their theoretical findings. Next, we performed our own numerical experiments using unidimensional and multidimensional orthogonal and non-orthogonal datasets. We used a wide variety of hyperparameter settings to explore interesting behaviors of GD in the training of one-hidden layer ANNs. Driven by the questions of convergence to zero loss, training dynamics, implicit bias and training regimes, we experimentally showed that the findings of [5] hold for orthogonal and quasi-orthogonal data with $\delta = \mathcal{O}(\lambda)$. We also showed that the neuron alignment phenomenon occurs for non-orthogonal input data, with some differences in the training dynamics. For instance, with unidimensional non-orthogonal input data, neurons first align in a finite number of directions while staying small in norm, and then grow in norm while slightly adjusting their

direction in groups[13].

In parallel with these investigations for the ReLU activation function, we conducted numerical experiments for Sigmoid and Tanh activation functions. We observed similar behavior in training dynamics, leading us to propose the following conjectures :

**Conjecture 1.** ***(Training dynamics of one-hidden layer Sigmoid ANNs)*** *For orthogonal input data, the training dynamics of one-hidden layer ANNs with Sigmoid activation function, minimizing the square loss with Gradient Method and balanced initialization with an initialization scale $\lambda < \lambda^*$, follows the three following phases :*

1. ***Alignment Phase*** *: Neurons first align in one direction, maintaining a low norm.*
2. ***Labels Fitting Phase*** *: Neurons increase in norm to fit data with positive and negative labels. As they grow, neurons slightly adjust their direction, following an S-curve only dependent on the dataset.*
3. ***Final Convergence Phase*** *: Finally, we observe a convergence to zero loss.*

*Moreover, we conjecture that, for $\{\theta^t\}$ generated with Gradient Method and low step-size $\alpha$, we have*

$$\lim_{\lambda \to 0} \lim_{t \to +\infty} \theta^t \in \arg\min_{\mathcal{L}(\theta)=0} ||\theta||_2^2. \tag{3.4}$$

The speed of these different phases is proportional to $-\ln(\lambda)$, and inversely proportional to the norm of a value that depends on the data.

**Conjecture 2.** ***(Training dynamics of one-hidden layer Tanh ANNs)*** *For orthogonal input data, the training dynamics of one-hidden layer ANNs with Tanh activation function, minimizing the square loss with Gradient Method and balanced initialization with an initialization scale $\lambda < \lambda^*$, follows the four following phases :*

1. ***Alignment Phase*** *: Neurons initially align in one direction, maintaining a low norm.*
2. ***Positive Labels Fitting Phase*** *: Neurons increase in norm to fit the positive label data, adjusting their direction to follow an S-curve dependent on the dataset.*
3. ***Negative Labels Fitting Phase*** *: After fitting the positive label data, neurons continue to grow in norm and drastically change their direction while following a deformed S-curve until they fit the negative label data.*
4. ***Final Convergence Phase*** *: Finally, we observe a convergence to zero loss.*

*Moreover, we conjecture that, for $\{\theta^t\}$ generated with Gradient Method and low step-size $\alpha$, we have*

$$\lim_{\lambda \to 0} \lim_{t \to +\infty} \theta^t \in \arg\min_{\mathcal{L}(\theta)=0} ||\theta||_2^2. \tag{3.5}$$

For Tanh, we observed the same dependency on $-\ln(\lambda)$ and the data for the speed of the four phases as mentioned in [5] for the ReLU activation function (see Timeline 2.6). These conjectures can be extended to the case of quasi-orthogonal data and normal initialization,

[13] This differs slightly from the theoretical predictions of [34], where the authors suggest that neurons maintain their direction while growing in norm. However, their analysis applies to Gradient Flow and to infinitesimal initialization scale $\lambda$, while our experiments use the Gradient Method with a finite step size and initialization scale, which may explain the discrepancy.

as for ReLU activation function.

Based on the assumption that the shape formed by neurons in the $(w_{j,1}, w_{j,2})$ plane is a linear combination of two ReLU functions up to a well-chosen plannar rotation made from the observations during the training of ReLU ANNs, we similarly characterize the S-curves formed in the $(w_{j,1}, w_{j,2})$ plane by the neuron disposition with Sigmoid and Tanh ANNs.

**Conjecture 3.** ***(Neuron Disposition for One-Hidden Layer ANNs)*** *For one-hidden layer ANNs with ReLU, Sigmoid or Tanh activation functions $\sigma$, trained using the Gradient Method with an initialization scale $\lambda < \lambda^*$, the neuron disposition $f(w)$ draws a shape that can be expressed as a linear combination of two functions of the same form as $\sigma$, i.e. there exists $K_1, K_2, k_1, k_2 \in \mathbb{R}$*

$$f(w) = K_1\sigma(k_1 w) + K_2\sigma(k_2 w). \tag{3.6}$$

If this conjecture is true, it implies that the final interpolator of a network with $m$ neurons is equivalent to that of a network with only two neurons, which is indeed the case for the ReLU activation function as mentioned in [5]. However, as previously discussed, this conjecture may be incomplete, although satisfactory in many cases. Further investigation would be necessary to fully understand its validity.

For non-orthogonal data, we observed the existence of Rich regime, Lazy regime, and transition regime for the three activation functions, as previously observed for orthogonal data. Related to this, the model complexity $\|\theta^*_{\text{GD}}\|$ decreases linearly with $\lambda$ in the Lazy regime, reaching a minimum at a certain value of $\lambda$ (which we associate to the beginning of the Rich regime). In the Rich regime, we observed that the training loss curve exhibits alternating fast and slow convergence phases. This phenomenon is linked to saddle-to-saddle dynamics for the ReLU activation function as noted in [5], and we can suppose that this is also the case for Sigmoid and Tanh activation functions. For all the three activation functions, we observed similar neuron dispositions at the end of training as seen for orthogonal data. For ReLU, neurons align and grow in a finite number of directions, while for Sigmoid and Tanh, deformed S-curves appear in the $(w_{j,1}, w_{j,2})$ plane.

Finally, we conducted numerical experiments with multidimensional data to confirm our observations from the unidimensional case.

# Chapter 4

# Theoretical exploration of the loss landscape

In Chapter 2, we summarized the findings of [5] stating notably that the Gradient Flow converges to zero loss for one-hidden layer ReLU networks for orthogonal data and other mild conditions. In Chapter 3, we explored training dynamics of such over-parameterized networks, showing that, in the most cases, the Gradient Method converges to zero loss for sufficiently small step-size in many parameter settings. Here, we aim to find an alternative explanation of this fact for Sigmoid and Tanh activation functions, using the notion of Polyak-Łojasiewicz (PL) condition. To achieve this, we first calculate the gradient and the Hessian of the loss, we then introduce the notion of PL condition and show that one-hidden layer neural networks do not satisfy the PL condition on the whole parameter space. After that, we explore some theoretical results of the literature about the application of PL condition for neural networks. Besides all this, at the end of the chapter, we provide a proof of the Lemma 5 from [5] about the balancedness of iterates $\theta^t$ over the training process for one-hidden layer ReLU neural networks, since the authors do not provide it.

## 4.1 Preliminary calculation

Because we need it in the formulas of $\nabla_\theta \mathcal{L}$, we compute the subgradients of the ReLU function, and the derivatives of Sigmoid and Tanh. We obtain

$$\sigma'_R(x) = \begin{cases} 1 & \text{if } x > 0 \\ g \in [-1, 1] & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases} \qquad \sigma''_R(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\sigma'_S(x) = \frac{e^{-x}}{(1 + e^{-x})^2} \qquad \sigma''_S(x) = \frac{e^{-x}\,(e^{-x} - 1)}{(1 + e^{-x})^3}$$

$$\sigma'_T(x) = 1 - \sigma_T(x)^2 \qquad \sigma''_T(x) = -2\,\sigma_T(x)\left(1 - \sigma_T(x)^2\right)$$

Besides, as noted in [18], we can rewrite the square loss as follows

$$\mathcal{L}(\theta) = \frac{1}{2n}\sum_{k=1}^{n}\left(y_{pred,k} - y_k\right)^2 = \frac{1}{2n}\sum_{k=1}^{n} r_k(\theta)^2 = \frac{1}{2n}\|r(\theta)\|^2$$

with $r(\theta) \in \mathbb{R}^n$ the residuals of the nonlinear least squares problem. We can explicit the residuals as follows

$$r(\theta) = \begin{bmatrix} r_1(\theta) \\ \vdots \\ r_k(\theta) \\ \vdots \\ r_n(\theta) \end{bmatrix} = \begin{bmatrix} y_{pred,1} - y_1 \\ \vdots \\ y_{pred,k} - y_k \\ \vdots \\ y_{pred,n} - y_n \end{bmatrix} \tag{4.1}$$

We recall that the weights are noted as $\theta = (W, a) \in \mathbb{R}^{m\times d} \times \mathbb{R}^m$. We can compute the partial derivatives of each $r_k(\theta)$ with respect to each weight $\theta_i$ :

$$\begin{aligned} \frac{\partial r_k}{\partial a_I} &= \frac{\partial\left(y_{pred,k} - y_k\right)}{\partial a_I} \\ &= \frac{\partial\left(y_{pred,k}\right)}{\partial a_I} \\ &= \frac{\partial}{\partial a_I}\left(\sum_{i=1}^{m} a_i\, \sigma\left(\sum_{j=1}^{d} w_{i,j}x_{k,j}\right)\right) \\ &= \sigma\left(\sum_{j=1}^{d} w_{I,j}x_{k,j}\right) \end{aligned}$$

$$\begin{aligned} \frac{\partial r_k}{\partial w_{I,J}} &= \frac{\partial\left(y_{pred,k} - y_k\right)}{\partial w_{I,J}} \\ &= \frac{\partial\left(y_{pred,k}\right)}{\partial w_{I,J}} \\ &= \frac{\partial}{\partial w_{I,J}}\left(\sum_{i=1}^{m} a_i\, \sigma\left(\sum_{j=1}^{d} w_{i,j}x_{k,j)}\right)\right) \\ &= a_I\, \sigma'\left(\sum_{j=1}^{d} w_{I,j}x_{k,j}\right) x_{k,J} \end{aligned}$$

We also calculate the second derivatives of $r_k(\theta)$ :

$$\frac{\partial^2 r_k}{\partial a_{I_1}\partial a_{I_2}} = \frac{\partial}{\partial a_{I_2}}\left(\sigma\left(\sum_{j=1}^{d} w_{I_1,j}x_{k,j}\right)\right) = 0$$

$$\frac{\partial^2 r_k}{\partial a_{I_1}\partial w_{I_2,J_2}} = \frac{\partial}{\partial w_{I_2,J_2}}\left(\sigma\left(\sum_{j=1}^{d} w_{I_1,j}x_{k,j}\right)\right) = \begin{cases} 0 & \text{if } I_1 \neq I_2 \\ \sigma'\left(\sum_{j=1}^{d} w_{I,j}x_{k,j}\right) x_{k,J_2} & \text{if } I_1 = I_2 = I \end{cases}$$

$$\frac{\partial^2 r_k}{\partial w_{I_1,J_1}\partial w_{I_2,J_2}} = \frac{\partial}{\partial w_{I_2,J_2}}\left(a_{I_1}\, \sigma'\left(\sum_{j=1}^{d} w_{I_1,j}x_{k,j}\right) x_{k,J_1}\right) = \begin{cases} 0 & \text{if } I_1 \neq I_2 \\ a_I\, \sigma''\left(\sum_{j=1}^{d} w_{I,j}x_{k,j}\right) x_{k,J_1}x_{k,J_2} & \text{if } I_1 = I_2 = I \end{cases}$$

### 4.1.1 Gradient of the square loss

Now, we can calculate the gradient with the following formulas :

$$\nabla_\theta \mathcal{L}(\theta) = \frac{1}{n}\mathcal{J}_r(\theta)^T r(\theta) = \frac{1}{n}\begin{bmatrix} \frac{\partial r_1}{\partial \theta_1} & \cdots & \frac{\partial r_1}{\partial \theta_{m(d+1)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial r_n}{\partial \theta_1} & \cdots & \frac{\partial r_n}{\partial \theta_{m(d+1)}} \end{bmatrix}^T \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} = \left[\frac{1}{n}\sum_{k=1}^{n}\frac{\partial r_k}{\partial \theta_i} r_k\right]_{i=1,\cdots,m(d+1)}$$

with $\mathcal{J}_r(\theta)$ the Jacobian of $r(\theta)$. Calculating all elements of the gradient using the formulas of the partial derivative of $r$, we obtain

$$\nabla_\theta \mathcal{L}(\theta) = \begin{bmatrix} \vdots \\ \frac{\partial \mathcal{L}}{\partial w_{I,J}} \\ \vdots \\ \hline \vdots \\ \frac{\partial \mathcal{L}}{\partial a_I} \\ \vdots \end{bmatrix} = \begin{bmatrix} \vdots \\ \frac{1}{n}\sum_{k=1}^{n}\left(y_{pred,k} - y_k\right) a_I\, \sigma'\left(\sum_{j=1}^{d} w_{I,j} x_k^{(j)}\right) x_k^{(J)} \\ \vdots \\ \hline \vdots \\ \frac{1}{n}\sum_{k=1}^{n}\left(y_{pred,k} - y_k\right) \sigma\left(\sum_{j=1}^{d} w_{I,j} x_k^{(j)}\right) \\ \vdots \end{bmatrix}$$

### 4.1.2 Hessian of the square loss

The hessian of $\mathcal{L}$ can be written as

$$\nabla_\theta^2 \mathcal{L}(\theta) = \frac{1}{n}\left(J_r(\theta)^T J_r(\theta) + \sum_{k=1}^{n} r_k(\theta)\nabla^2 r_k(\theta)\right) = \left[\frac{1}{n}\sum_{k=1}^{n}\frac{\partial r_k}{\partial \theta_i}\frac{\partial r_k}{\partial \theta_j} + \left(y_{pred,k} - y_k\right)\frac{\partial^2 r_k}{\partial \theta_i \partial \theta_j}\right]_{i,j=1,\cdots,m(d+1)}$$

Again, using the residuals formulas, we obtain

$$\nabla_\theta^2 \mathcal{L}(\theta) = \left[\begin{array}{ccc|ccc} \ddots & \vdots & & \ddots & \vdots & \\ \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_{I_1,J_1}\partial w_{I_2,J_2}} & \cdots & \cdots & \frac{\partial^2 \mathcal{L}}{\partial w_{I_1,J_1}\partial a_{I_2}} & \cdots \\ & \vdots & \ddots & & \vdots & \ddots \\ \hline \ddots & \vdots & & \ddots & \vdots & \\ \cdots & \frac{\partial^2 \mathcal{L}}{\partial a_{I_1}\partial w_{I_2,J_2}} & \cdots & \cdots & \frac{\partial^2 \mathcal{L}}{\partial a_{I_1}\partial a_{I_2}} & \cdots \\ & \vdots & \ddots & & \vdots & \ddots \end{array}\right]$$

with

$$\frac{\partial^2 \mathcal{L}}{\partial a_{I_1} \partial a_{I_2}} = \frac{1}{n} \sum_{k=1}^{n} \sigma\left(\sum_{j=1}^{d} w_{I_1,j} x_{k,j}\right) \sigma\left(\sum_{j=1}^{d} w_{I_2,j} x_{k,j}\right)$$

$$\frac{\partial^2 \mathcal{L}}{\partial w_{I_1,J_1} \partial w_{I_2,J_2}} = \frac{1}{n} \sum_{k=1}^{n} a_{I_1} a_{I_2} \sigma'\left(\sum_{j=1}^{d} w_{I_1,j} x_{k,j}\right) \sigma'\left(\sum_{j=1}^{d} w_{I_2,j} x_{k,j}\right) x_{k,J_1} x_{k,J_2} \quad \text{if } I_1 \neq I_2$$

$$\frac{\partial^2 \mathcal{L}}{\partial w_{I,J_1} \partial w_{I,J_2}} = \frac{1}{n} \sum_{k=1}^{n} a_I \left[ a_I \sigma'\left(\sum_{j=1}^{d} w_{I,j} x_{k,j}\right)^2 + \left[y_{pred,k} - y_k\right] \sigma''\left(\sum_{j=1}^{d} w_{I,j} x_{k,j}\right)\right] x_{k,J_1} x_{k,J_2}$$

$$\frac{\partial^2 \mathcal{L}}{\partial a_{I_1} \partial w_{I_2,J_2}} = \frac{1}{n} \sum_{k=1}^{n} a_{I_2} \sigma\left(\sum_{j=1}^{d} w_{I_1,j} x_{k,j}\right) \sigma'\left(\sum_{j=1}^{d} w_{I_2,j} x_{k,j}\right) x_{k,J_2} \quad \text{if } I_1 \neq I_2$$

$$\frac{\partial^2 \mathcal{L}}{\partial a_I \partial w_{I,J}} = \frac{1}{n} \sum_{k=1}^{n} \left[ a_I \sigma\left(\sum_{j=1}^{d} w_{I,j} x_{k,j}\right) + \left[y_{pred,k} - y_k\right]\right] \sigma'\left(\sum_{j=1}^{d} w_{I,j} x_{k,j}\right) x_{k,J}$$

## 4.2 PL condition

As mentioned in Section 2.1.4, the PL condition provides a better framework than classical convexity-based analysis for studying over-parameterized neural networks. Here, we define the PL condition, discuss its properties, and present related results. Additionally, we briefly investigate whether the square loss of a one-hidden layer neural network satisfies the PL condition for the model defined in Section 2.2.1.1 for ReLU, Sigmoid and Tanh activation functions.

### 4.2.1 Definition and properties

In [20], the authors summarize convergence results for GD and many variants of SGD for different types of functions. Notably, they define and discuss results for functions that satisfy the PL condition. In this section, we present the definition and some properties associated with this concept, beginning with the formal definition of the PL condition.

**Definition 5.** ***(PL condition)*** *Let $F : \mathbb{R}^D \to \mathbb{R}$ be differentiable, and $\mu > 0$. We say that $F$ is $\mu$-PL if it is bounded from below, and if $\forall \theta \in \mathbb{R}^D$*

$$F(\theta) - \inf_{\theta} F(\theta) \leq \frac{1}{2\mu} \|\nabla F(\theta)\|^2. \tag{4.2}$$

For example, all $\mu$-strongly convex functions are $\mu$-PL. Another example of a unidimensional nonconvex function that satisfies the PL condition is $F(\theta) = \theta^2 + 3\sin(\theta)^2$. This function is shown on Figure 4.1.

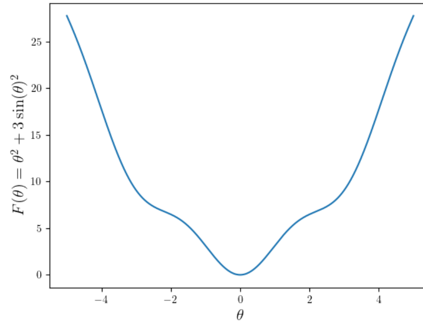



FIGURE 4.1: Example of nonconvex function that satisfies the PL condition : $F(\theta) = \theta^2 + 3\sin(\theta)^2$.

They also provide the following sufficient condition to prove that a nonlinear least square problem satisfies the PL condition. Because training a neural network by minimizing the square loss is a nonlinear least square problem, we are particularly interested in this condition.

**Theorem 7.** *(Sufficient condition for PL condition)*
*For the square loss $F(\theta) = \frac{1}{2n}\|r(\theta)\|^2 = \frac{1}{2n}\|h(\theta) - y\|^2$, the Jacobian of the residuals $\mathcal{J}_r(\theta)$, where $h(\theta) : \mathbb{R}^D \to \mathbb{R}^n$ is differentiable. Then $F$ is PL if $\mathcal{J}_r(\theta)$ is uniformly injective :*

$$\exists\mu > 0 : \forall\theta \in \mathbb{R}^D, \quad \lambda_{\min}\left(\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T\right) \geq \mu.$$

Using the fact that $F$ is non-negative, this sufficient condition holds since

$$\|\nabla F(\theta)\|^2 = \|\mathcal{J}_r(\theta)^T(h(\theta) - y)\|^2 \geq \mu\|h(\theta) - y\|^2 = 2\mu F(\theta) \geq 2\mu(F(\theta) - \inf_\theta F(\theta)).$$

The authors report that for neural networks with differentiable activation functions (such as Sigmoid and Tanh), this condition can hold only if $D \geq n$, meaning that it can apply to over-parameterized networks. Indeed, for an under-parameterized model, since $\mathcal{J}_r(\theta) \in \mathbb{R}^{n\times D}$, we have that

$$\text{rank}\left(\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T\right) \leq n.$$

So, thanks to the Rank-Nullity Theorem, we have that

$$\text{rank}\left(\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T\right) + \dim\left(\text{kernel}\left(\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T\right)\right) = n$$

$$\implies \dim\left(\text{kernel}\left(\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T\right)\right) \geq n - D$$

Because the kernel is

$$\text{kernel}\left(\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T\right) = \{v \mid \mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T v = 0\},$$

the dimension of the kernel is the number of zero eigenvalues of $\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T$. So, for $n > D$, there is at least one zero eigenvalue for $\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T$. Thus, we cannot use the Theorem 7 to prove that $\mathcal{L}(.)$ is a PL function.

An interesting property of PL functions is stated in the following lemma. It states that, for any differentiable PL function, every stationary point is a global minimizer of the function.

**Lemma 8.** *Let $F : \mathbb{R}^D \to \mathbb{R}$ be a differentiable PL function. Then $\theta^* \in \arg\min_{\theta \in \mathbb{R}^D} F(\theta)$ if and only if $\nabla F(\theta^*) = 0$.*

This implies that, if we find a stationary point of a differentiable function $F$ that is not a global minimizer, we can conclude that $F$ is not a PL function.

### 4.2.2 PL condition for Shallow Neural Networks

Now, we want to investigate whether the square loss $\mathcal{L}(\theta) = \frac{1}{2n}|h(\theta) - y|^2$, used to train one-hidden layer neural networks, satisfies the PL condition stated above. If it does, this would imply that every stationary point of $\mathcal{L}$ is also a global minimizer, partially explaining why GD is effective for training such network architectures. Moreover, as reported in [20], to reach a given precision $\epsilon$, we need $\mathcal{O}(\log(\epsilon^{-1}))$ iterations. We explore this for the model defined in Section 2.2.1.1, i.e.

$$h_\theta(x_k) = \sum_{j=1}^{m} a_j \, \sigma(w_j^T x_k).$$

From our observations in Chapter 3, we can suspect that there is a stationary point in $\theta = 0$, since for very small initialization around 0, the training loss curve starts with a wide plateau. To verify this intuition, it is sufficient to verify if $\nabla \mathcal{L}(0) = 0$. Using the formulas of $\nabla \mathcal{L}(\theta)$ derived at the beginning of this chapter, we obtain

$$\nabla_\theta \mathcal{L}(0) = \left.\left[\begin{array}{c} \vdots \\ \frac{1}{n}\sum_{k=1}^{n}\left(y_{pred,k} - y_k\right) a_I \, \sigma'\left(\sum_{j=1}^{d} w_{I,j} x_k^{(j)}\right) x_k^{(J)} \\ \vdots \\ \hline \vdots \\ \frac{1}{n}\sum_{k=1}^{n}\left(y_{pred,k} - y_k\right) \sigma\left(\sum_{j=1}^{d} w_{I,j} x_k^{(j)}\right) \\ \vdots \end{array}\right]\right|_{\theta=0} = \left[\begin{array}{c} \vdots \\ 0 \\ \vdots \\ \hline \vdots \\ -\frac{\sigma(0)}{n}\sum_{k=1}^{n} y_k \\ \vdots \end{array}\right]$$

For ReLU activation function, we already know from [5] that the square loss is not PL on the whole parameter space. Indeed, to show it, it is sufficient to remark that $\nabla_\theta \mathcal{L}(0) = 0$ and $\mathcal{L}(0) > 0$ if we use the particular subgradient $\sigma_R(0) = 0$, and if we have at least one nonzero label $y_k$.

For the Tanh activation function, $\sigma_T(0) = 0$, leading to $\nabla \mathcal{L}(0) = 0$. Because $h_x(\theta)|_{\theta=0} = 0$, the square loss is strictly positive in $\theta = 0$ if at least one label $y_k$ is different to zero. So, for the Tanh activation function, we have proved that the origin is a non-optimal stationary point of the square loss function for the model defined in 2.2.1.1. When we compute the eigenvalues of $\nabla^2 \mathcal{L}(0)$, we obtain positive and negative eigenvalues, showing that $\theta = 0$ is a saddle point of $\mathcal{L}$.

For Sigmoid, we can follow the same reasoning as for Tanh if the sum of the label is zero, implying that the network is not PL for this kind of dataset. When $\sum_k y_k \neq 0$, the zero vector is not a saddle point anymore, since the $\nabla_\theta \mathcal{L}(\theta) \neq 0$. Since there is no trivial saddle anymore, we can try to apply the sufficient condition for the PL condition 7. For simplicity, consider a

dataset with two data. First, we need to determine the form of the matrix $\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T$ :

$$\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T = \begin{bmatrix} \frac{\partial r_1}{\partial \theta_1} & \cdots & \frac{\partial r_1}{\partial \theta_D} \\ \frac{\partial r_2}{\partial \theta_1} & \cdots & \frac{\partial r_2}{\partial \theta_D} \end{bmatrix} \begin{bmatrix} \frac{\partial r_1}{\partial \theta_1} & \frac{\partial r_2}{\partial \theta_1} \\ \vdots & \vdots \\ \frac{\partial r_1}{\partial \theta_D} & \frac{\partial r_2}{\partial \theta_D} \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{D} \left(\frac{\partial r_1}{\partial \theta_i}\right)^2 & \sum_{i=1}^{D} \frac{\partial r_1}{\partial \theta_i}\frac{\partial r_2}{\partial \theta_i} \\ \sum_{i=1}^{D} \frac{\partial r_1}{\partial \theta_i}\frac{\partial r_2}{\partial \theta_i} & \sum_{i=1}^{D} \left(\frac{\partial r_2}{\partial \theta_i}\right)^2 \end{bmatrix}$$

In order to know if $\exists \mu > 0 : \forall \theta \in \mathbb{R}^D, \quad \lambda_{\min}\left(\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T\right) \geq \mu$, we can use the Sylvester criterion.

**Theorem 9.** *(Sylvester criterion)*
*For $A \in \mathbb{R}^{n \times n}$ square and symmetric, $A$ is positive-definite if and only if all main dominant minors of $A$ are positive, i.e.*

$$\det(A_p) > 0 \tag{4.3}$$

*for all $p \in \{1, \cdots, n\}$, with $A_p = \left[a_{i,j}\right]_{i,j=1,\cdots,p}$.*

Using this criterion and the fact that all eigenvalues of a positive-definite matrix are strictly greater than 0, we are able to determine if all eigenvalues of $\mathcal{J}_r(\theta)\mathcal{J}_r(\theta)^T$ are strictly greater than zero or not for every $\theta$.

- $p = 1$ :

$$\begin{aligned} \det([\mathcal{J}\mathcal{J}^T]_1) &= \sum_{i=1}^{D} \left[\frac{\partial r_1}{\partial \theta_i}\right]^2 \\ &= \sum_{I=1}^{m}\sum_{J=1}^{d} \left[a_I\, \sigma_S'\left(w_I^T x_1\right) x_{1,J}\right]^2 + \sum_{I=1}^{m} \left[\sigma_S\left(w_I^T x_1\right)\right]^2 \end{aligned}$$

  To have this value equals to zero, we need to have every term equals to zero, which is impossible since $\sigma_S(x) > 0$ for all $x$. So, $\det([\mathcal{J}\mathcal{J}^T]_1) > 0$ is ensured for all $\theta$.

- $p = 2$ :

$$\begin{aligned} \det([\mathcal{J}\mathcal{J}^T]_2) &= \sum_{i=1}^{D} \left[\frac{\partial r_1}{\partial \theta_i} - \frac{\partial r_2}{\partial \theta_i}\right]^2 \\ &= \sum_{I=1}^{m}\sum_{J=1}^{d} \left[a_I\, \left(\sigma_S'\left(w_I^T x_1\right) x_{1,J} - \sigma_S'\left(w_I^T x_2\right) x_{2,J}\right)\right]^2 + \sum_{I=1}^{m} \left[\sigma_S\left(w_I^T x_1\right) - \sigma_S\left(w_I^T x_2\right)\right]^2 \end{aligned}$$

  Here, it is sufficient to take $\theta = (W, a) = 0$ to have $\det([\mathcal{J}\mathcal{J}^T]_2) = 0$. So, even in this simple setting, the sufficient condition is not powerful enough to prove that this function is PL.

### 4.2.3 PL condition for Neural Networks in the literature

Because the loss function of a neural network often contains saddle points and sub-optimal minimizers in its landscape, a relaxed version of the PL condition exists, known as the *PL* condition* (also called the *local PL condition*). Below, we present its definition, sourced from [32], for a nonnegative function:

**Definition 6.** ***(PL* or local PL condition)*** *A nonnegative function $\mathcal{L}$ is $\mu$-PL* on a set $\mathcal{S} \subset \mathbb{R}^D$ for $\mu > 0$ if*

$$\mathcal{L}(\theta) \leq \frac{1}{2\mu} \|\nabla \mathcal{L}(\theta)\|^2 \tag{4.4}$$

*for all $\theta \in \mathcal{S}$.*

The authors of [5] (summarized in Section 2.2.1) used a form of the PL* condition to prove that the square loss is locally PL around global and balanced minimizers found by the gradient flow with a small initialization scale for a one-hidden layer ReLU ANN with orthogonal data.

In [32], the authors show that for sufficiently wide feedforward neural networks with any number of layers, the PL* condition holds for the square loss within a ball around the initial weight vector $\theta^0$, provided the activation function is smooth. This implies that the PL* condition is satisfied in most, but not all, of the parameter space. They assume a random normal initialization with zero mean and unit variance ($w_{i,j}^{(l)} \sim \mathcal{N}(0,1)$) and some other mild conditions. Moreover, they demonstrate that this condition guarantees the existence of solutions and a linear convergence rate for (S)GD. More precisely, the PL* condition ensures the existence of solutions, and the convergence of (S)GD, if it holds in a ball of sufficient radius. Indeed, if the $\mathcal{L}$ is $\mu$-PL* in a ball of center $\theta^0$ with radius $\mathcal{O}\left(\frac{1}{\mu}\right)$, then there exists a global minimum in the ball and (S)GD starting from $\theta^0$ converges linearly to this global minimum. Additionally, they show that the optimization path is bounded by the size of the ball in which the PL* condition is satisfied. Since this ball is finite and the path remains within it, it ensures that the length of the path from the initial point $\theta^0$ to the global minimum $\theta^*$ is finite. Moreover, the closest global minimum to the initial point must be at least $\frac{|h_{\theta^0} - y|}{L_h}$ away from $\theta^0$, where $L_h$ is the Lipschitz constant of the model $h$.

## 4.3 Proof of Lemma 5 about Balancedness of iterates $\theta^t$

Since the authors of [5] do not provide any proof, we provide a proof of Lemma 5, which is restated just below.

**Lemma 10.** *For all $t \geq 0$ and $j \in \{1, \cdots, m\}$, $(a_j^t)^2 - \|w_j^t\|^2 = (a_j^0)^2 - \|w_j^0\|^2$.*
*Assume furthermore that for all $j \in \{1, \cdots, m\}$, the initialization is balanced and non-zero : $|a_j^0| = \|w_j^0\| > 0$. Then $|a_j^t| = \|w_j^t\| > 0$, and letting $s = sign\left(a^0\right) \in \{-1,1\}^m$, we have that $a_j^t = s_j \|w_j^t\|$ for all $t \geq 0$.*

**Proof.** Thanks to the gradient flow formulas 2.40 and the computation of $\nabla_\theta \mathcal{L}$ done at the begining of this chapter, we deduce the following equations

$$\frac{\mathrm{d}a_i^t}{\mathrm{d}t} = -\frac{1}{n} \sum_{k=1}^{n} \left(h_{\theta^t}(x_k) - y_k\right) \sigma\left((w_i^t)^T x_k\right) \tag{4.5}$$

$$\frac{\mathrm{d}w_{i,j}^t}{\mathrm{d}t} = -\frac{1}{n} \sum_{k=1}^{n} \left(h_{\theta^t}(x_k) - y_k\right) a_i^t \, \sigma'\left((w_i^t)^T x_k\right) x_{k,j}. \tag{4.6}$$

- First, we calculate :

$$\begin{aligned}
\|w_i^t\|^2 - \|w_i^0\|^2 &= \int_0^t \frac{\mathrm{d}}{\mathrm{d}\tau}\left(\|w_i^\tau\|^2\right) \mathrm{d}\tau \\
&= \int_0^t \sum_{j=1}^d \frac{\mathrm{d}}{\mathrm{d}\tau}\left((w_{i,j}^\tau)^2\right) \mathrm{d}\tau \\
&= \int_0^t \sum_{j=1}^d \frac{\mathrm{d}\left((w_{i,j}^\tau)^2\right)}{\mathrm{d}w_{i,j}^\tau} \frac{\mathrm{d}w_{i,j}^\tau}{\mathrm{d}\tau} \mathrm{d}\tau \\
&= \int_0^t \sum_{j=1}^d 2w_{i,j}^\tau \left[-\frac{1}{n}\sum_{k=1}^n \left(h_{\theta^\tau}(x_k) - y_k\right) a_i^\tau\, \sigma'\left((w_i^\tau)^T x_k\right) x_{k,j}\right] \mathrm{d}\tau \\
&= -\frac{2}{n}\sum_{k=1}^n \int_0^t a_i^\tau \left(h_{\theta^\tau}(x_k) - y_k\right)\, \sigma'\left((w_i^\tau)^T x_k\right) (w_i^\tau)^T x_k \,\mathrm{d}\tau
\end{aligned}$$

- In the same way, we calculate

$$(a_i^t)^2 - (a_i^0)^2 = -\frac{2}{n}\sum_{k=1}^n \int_0^t a_i^\tau \left(h_{\theta^\tau}(x_k) - y_k\right)\, \sigma\left((w_i^\tau)^T x_k\right) \mathrm{d}\tau$$

Since, for ReLU activation function $\sigma$, we have $\sigma(x) = \sigma'(x)x$, we deduce that

$$\begin{aligned}
&\|w_i^t\|^2 - \|w_i^0\|^2 = (a_i^t)^2 - (a_i^0)^2 \\
\Longrightarrow &(a_i^t)^2 - \|w_i^t\|^2 = (a_i^0)^2 - \|w_i^0\|^2
\end{aligned}$$

which proves the first part of the lemma.

For the second part, we assume that $|a_j^0| = \|w_j^0\| > 0$ for all $j \in \{1, \cdots, m\}$. Thus, since

$$\begin{aligned}
(a_i^t)^2 - \|w_i^t\|^2 &= (a_i^t - \|w_i^t\|)(a_i^t + \|w_i^t\|) \\
&= (a_i^0)^2 - \|w_i^0\|^2 \\
&= 0
\end{aligned}$$

we have $a_i^t = s_i^t \|w_i^t\|$ with $s_i^t = \text{sign}\left(a_i^t\right)$. Moreover, by continuity of the loss function and the gradient flow, and by the Intermediate Value Theorem, if $a_i^\tau = k$ with $k \in \mathbb{R}$, $a_i^t$ must take all the intermediate value between $a_i^0$ and $k$ for $t \in [0, \tau]$. Because of equations 4.5 and 4.6, when $a_i^t = 0$, we have $\frac{\mathrm{d}a_i^t}{\mathrm{d}t} = 0$ and $\frac{\mathrm{d}w_{i,j}^t}{\mathrm{d}t} = 0$ since $a_i^t = 0 \Rightarrow \|w_i^t\| = 0 \Rightarrow w_i^t = 0$. Thus, if $a_i^0$ is positive, $a_i^t$ is at least nonnegative, and if $a_i^0$ is negative, $a_i^t$ is at least nonpositive. It shows that $s_j^t = s_j = \text{sign}\left(a_i^0\right)$ for all $t \geq 0$.

Finally, using the Cauchy-Schwartz inequality on equation 4.5, we obtain

$$\left|\frac{\mathrm{d}a_i^t}{\mathrm{d}t}\right| = |(D_i^t)^T w_i^t| \leq \|D_i^t\| \|w_i^t\| = \|D_i^t\| |a_i^t| \leq K |a_i^t|$$

with $D_i^t = -\frac{1}{n}\sum_{k=1}^n \left(h_{\theta^t}(x_k) - y_k\right)\right) \sigma'\left((w_i^t)^T x_k\right) x_k \in \mathbb{R}^d$ and $K$ a constant such that $\|D_i^t\| \leq K < +\infty$. This constant is easy to derive, since the loss at initialization is finite. So, $\frac{\mathrm{d}a_i^t}{\mathrm{d}t}$ is bounded by exponential functions of $t$, showing that, in the worst case, if $|a_i^0| > 0$, $|a_i^t|$ can

asymptotically approach 0, but never reach 0. This implies that

$$|a_i^t| = \|w_i^t\| > 0$$

for all $t \geq 0$. This concludes the prove.

# Chapter 5

# Conclusion

In this thesis, after a few preliminaries, we made a State-of-The-Art review of training in Deep Learning. We explored different facets of Deep Learning training, such as the explosion/vanishing gradient problem, initialization methods, optimization algorithms used to train the neural network models, as well as convergence analysis and landscape approach for characterizing the behavior of Gradient-based optimization algorithms in this field. The most important results we found are the fact that, for overparameterized networks, the loss landscape is highly nonconvex, even locally. This implies that the classical convergence analysis of (S)GD based on local convexity is inadequate for studying their convergence for this kind of problems. We also talk about implicit bias and their related training regimes, which we highly used in the rest of our work. At the end of Chapter 2, we summarized the article [5] characterizing the convergence of the Gradient Flow to zero loss, the implicit bias and the training dynamics for one-hidden layer ReLU neural networks for orthogonal input data minimizing square loss. They prove that, under small initialization, the Gradient Flow of such networks converges to zero loss, implicitly choosing an optimal solution with the smallest possible $\ell_2$-norm. They also show a neuron alignment resulting in a reduction of the network to a two-neurons network in four distinct phases, and a saddle-to-saddle dynamics.

After replicating the numerical experiments made in [5] on toy unidimensional dataset, we performed our own numerical experiments, showing that their theoretical findings are coherent. We also aimed to discern the limitations of their postulated assumptions. In this view, we performed our own numerical experiments using unidimensional and multidimensional orthogonal and non-orthogonal datasets. We used a wide variety of hyperparameter settings to explore interesting behaviors of GD in the training of one-hidden layer ANNs. Driven by the questions of convergence to zero loss, training dynamics, implicit bias and training regimes, we experimentally showed that the findings of [5] hold for orthogonal and quasi-orthogonal data with $\delta = \mathcal{O}(\lambda)$. We also showed that the neuron alignment phenomenon occurs for non-orthogonal input data, with some differences in the training dynamics. For instance, with unidimensional non-orthogonal input data, neurons first align in a finite number of directions while staying small in norm, and then grow in norm while slightly adjusting their direction in groups.

In parallel with these investigations for the ReLU activation function, we conducted numerical experiments for Sigmoid and Tanh activation functions. We observed similar behavior in training dynamics, leading us to propose three conjectures. The two first characterized the training dynamics of one-hidden layer Sigmoid and Tanh ANNs respectively. They stated in both cases that, first, there is an alignment phase of neurons in one direction, maintaining their norm small. Then, neurons grow in norm to fit positive and negative labels data (this second phase is separated in two for Tanh activation function). Finally, there is a convergence phase to zero loss. All these phases are proportional to $-\ln(\lambda)$ with $\lambda$ the

initialization scale. Moreover, the final interpolator generated by the Gradient Method is the smallest possible one with respect to the $\ell_2$-norm, as for ReLU activation function. In the third conjecture, we predicted that, during the training of one-hidden layer neural network with either ReLU, Sigmoid or Tanh activation function, the neuron disposition draws a shape that can be expressed as a linear combination of two functions of the same form as the used activation function when using small initialization scale. For Sigmoid and Tanh, it results in an S-curve drawn in the parameter space of the hidden layer. We also briefly discussed the limit of this conjecture, and why we thought that is incomplete. However, completing this conjecture would be the work of a future project.

For orthogonal and non-orthogonal input data, we observed the existence of the Rich, Lazy, and transition regimes between the two for the three activation functions. Related to this, the model complexity $\|\theta^*_{\text{GD}}\|$ decreases linearly with the initialization scale $\lambda$ in the Lazy regime, reaching a minimum at a certain value of $\lambda$ (which we associate to the beginning of the Rich regime). In the Rich regime, we observed that the training loss curve exhibits alternating fast and slow convergence phases, often associated to saddle-to-saddle dynamics in the literature. For all the three activation functions and non-orthogonal input data, we observed similar neuron dispositions at the end of training as seen for orthogonal data. For ReLU, neurons align and grow in a finite number of directions, while for Sigmoid and Tanh, deformed S-curves appear in the parameter space of the hidden layer.

In Chapter 4, we briefly explore the notion of PL condition, which seems to be a better framework than convexity to study the property of the Gradient Method minimizing nonconvex loss functions used in deep learning. We notably show that one-hidden neural networks with ReLU and Tanh never satisfy the PL condition on the whole parameter space since the zero vector is a sub-optimal stationary point. For Sigmoid activation function, the zero vector is not always a stationary point. Thus, we attempted to prove that the loss function of one-hidden layer Sigmoid ANNs satisfies the sufficient condition for being $\mu$-PL. But, unfortunately, this sufficient condition does not hold at the origin. After that, we briefly explored the literature about the PL condition and its use in deep learning field. Finally, since the authors of [5] do not provide any proof of Lemma 5 about the balancedness of iterates generated by the Gradient Flow during the training, we proposed our own proof.

We are aware that our contributions present some limits. First, the majority of our contributions is based on numerical experiments. Even though that numerical experiments are useful to explore and to understand certain behaviors in the training dynamics of neural networks, they do not prove formally that what we observe always occurs. Moreover, we performed our experiments with a finite number of hyperparameter settings in a certain range for each of these hyperparameters. When increasing and decreasing, certain hyperparameters could change drastically what we observe. Our numerical results may be useful as starting point for attempts to formally prove and complete our observations and conjectures.

From our work, several further directions are possible. Firstly, we could continue the numerical experiments, trying to better understand what we already observed. Secondly, we could try to formally prove the stated conjectures in Chapter 3. Lastly, we could continue our exploration of the loss landscape via the PL condition. To do so, we could use the notion of local PL condition and prove that, unless if the PL condition does not hold on the whole parameter space, one-hidden layer Neural Networks are locally PL in a ball around the final weights vector found by the Gradient Method. This can be helped by the results of [32], which proves that wide enough neural network are PL on the most of the parameter space, and of [37], which states that iterates of GD never leave a neighborhood of a certain radius

for one-hidden layer neural network when the number of data is less than the dimension of input vectors (which is the case for orthogonal data). Proving that the loss function is PL in these neighborhood could explain why we observe convergence to zero loss in this settings.

# Appendix A

# Generative AI for Report Improvement

Since we are native French speakers, our English skills for writing a complete report in English can be improved using generative AI, particularly for spelling mistakes and turns of phrase. To be transparent about the writing process of this work, we detail our use of these tools. Specifically, we used ChatGPT from OpenAI in the following manner:

- First, we wrote each part of the report ourselves (without the help of any tool).
- After that, paragraph by paragraph, we asked ChatGPT to "correct spelling mistakes and turn of phrases."
- Finally, we corrected sentence by sentence, comparing ChatGPT's proposed corrections with the original text.

It is important to note that only the English language was corrected using generative AI. The structure of the work, all information written in the text, and all presented results do not come from this type of tool, but from our own work.

# Appendix B

# Calculation of $\nabla^2\mathcal{L}(\theta)$ for the square loss

Let us calculate the Hessian of $L(\theta)$. We take $l_2 \geq l_1$ in order to have some quantities of layer $l_1$ independent of $w^{(l_2)}_{i_2,j_2}$. We use the same techniques as for deriving the gradient.

$$\frac{\partial^2 \mathcal{L}}{\partial w^{(l_1)}_{i_1,j_1} \partial w^{(l_2)}_{i_2,j_2}} = \frac{\partial}{\partial w^{(l_2)}_{i_2,j_2}} \left( \frac{\partial \mathcal{L}}{\partial w^{(l_1)}_{i_1,j_1}} \right).$$

Here, we have several different cases :

1. When $l_1 = l_2 = L+1$ :

$$\begin{aligned}
\frac{\partial^2 \mathcal{L}}{\partial w^{(L+1)}_{1,j_2} \partial w^{(L+1)}_{1,j_1}} &= \frac{\partial}{\partial w^{(L+1)}_{1,j_2}} \left( \frac{1}{n} \sum_{k=1}^{n} (y_{pred,k} - y_k)\, \sigma'\left(s^{(L+1)}_{1,k}\right) z^{(L)}_{j_1,k} \right) \\
&= \frac{1}{n} \sum_{k=1}^{n} \frac{\partial}{\partial w^{(L+1)}_{1,j_2}} \left( (y_{pred,k} - y_k)\, \sigma'\left(s^{(L+1)}_{1,k}\right) z^{(L)}_{j_1,k} \right) \\
&= \frac{1}{n} \sum_{k=1}^{n} \left[ \frac{\partial (y_{pred,k} - y_k)}{\partial w^{(L+1)}_{1,j_2}}\, \sigma'\left(s^{(L+1)}_{1,k}\right) + (y_{pred,k} - y_k) \frac{\partial}{\partial w^{(L+1)}_{1,j_2}} \left( \sigma'\left(s^{(L+1)}_{1,k}\right)\right) \right] z^{(L)}_{j_1,k} \\
&= \frac{1}{n} \sum_{k=1}^{n} \left[ \frac{\partial y_{pred,k}}{\partial y_{pred,k}} \frac{\partial y_{pred,k}}{\partial s^{(L+1)}_{1,k}} \frac{\partial s^{(L+1)}_{1,k}}{\partial w^{(L+1)}_{1,j_2}}\, \sigma'\left(s^{(L+1)}_{1,k}\right) + (y_{pred,k} - y_k) \frac{\partial}{\partial w^{(L+1)}_{1,j_2}} \left( \sigma'\left(s^{(L+1)}_{1,k}\right)\right) \right] z^{(L)}_{j_1,k} \\
&= \frac{1}{n} \sum_{k=1}^{n} \left[ \sigma'\left(s^{(L+1)}_{1,k}\right) z^{(L)}_{j_2,k}\, \sigma'\left(s^{(L+1)}_{1,k}\right) + (y_{pred,k} - y_k) \frac{\partial}{\partial w^{(L+1)}_{1,j_2}} \left( \sigma'\left(s^{(L+1)}_{1,k}\right)\right) \right] z^{(L)}_{j_1,k} \\
&= \frac{1}{n} \sum_{k=1}^{n} \left[ \sigma'\left(s^{(L+1)}_{1,k}\right)^2 z^{(L)}_{j_2,k} + (y_{pred,k} - y_k) \frac{\partial}{\partial s^{(L+1)}_{1,k}} \left( \sigma'\left(s^{(L+1)}_{1,k}\right)\right) \frac{\partial s^{(L+1)}_{1,k}}{\partial w^{(L+1)}_{1,j_2}} \right] z^{(L)}_{j_1,k} \\
&= \frac{1}{n} \sum_{k=1}^{n} \left[ \sigma'\left(s^{(L+1)}_{1,k}\right)^2 z^{(L)}_{j_2,k} + (y_{pred,k} - y_k)\, \sigma''\left(s^{(L+1)}_{1,k}\right) \frac{\partial}{\partial w^{(L+1)}_{1,j_2}} \left( \sum_{p=1}^{m^{(L)}} w^{(L+1)}_{1,p} z^{(L)}_{p,k} \right) \right] z^{(L)}_{j_1,k} \\
&= \frac{1}{n} \sum_{k=1}^{n} \left[ \sigma'\left(s^{(L+1)}_{1,k}\right)^2 z^{(L)}_{j_2,k} + (y_{pred,k} - y_k)\, \sigma''\left(s^{(L+1)}_{1,k}\right) z^{(L)}_{j_2,k} \right] z^{(L)}_{j_1,k} \\
&= \frac{1}{n} \sum_{k=1}^{n} \left[ \sigma'\left(s^{(L+1)}_{1,k}\right)^2 + (y_{pred,k} - y_k)\, \sigma''\left(s^{(L+1)}_{1,k}\right) \right] z^{(L)}_{j_1,k} z^{(L)}_{j_2,k}
\end{aligned}$$

$$\implies \frac{\partial^2 \mathcal{L}}{\partial w_{1,j_1}^{(L+1)} \partial w_{1,j_2}^{(L+1)}} = \frac{1}{n}\sum_{k=1}^{n}\left[\sigma'\left(s_{1,k}^{(L+1)}\right)^2 + (y_{pred,k} - y_k)\,\sigma''\left(s_{1,k}^{(L+1)}\right)\right] z_{j_1,k}^{(L)} z_{j_2,k}^{(L)}$$

2. When $l_1 \leq L$ and $l_2 = L+1$ :

$$\begin{aligned}
\frac{\partial^2 \mathcal{L}}{\partial w_{i_1,j_1}^{(l_1)} \partial w_{1,j_2}^{(L+1)}} &= \frac{\partial}{\partial w_{1,j_2}^{(L+1)}}\left(\sum_{k=1}^{n}\left[\sum_{p=1}^{m^{(l_1+1)}} \frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l_1+1)}} w_{p,i_1}^{(l_1+1)} \sigma'\left(s_{i_1,k}^{(l_1)}\right)\right] z_{j_1,k}^{(l_1-1)}\right) \\
&= \sum_{k=1}^{n}\sum_{p=1}^{m^{(l_1+1)}} \frac{\partial}{\partial w_{1,j_2}^{(L+1)}}\left(\frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l_1+1)}} w_{p,i_1}^{(l_1+1)}\right)\sigma'\left(s_{i_1,k}^{(l_1)}\right) z_{j_1,k}^{(l_1-1)} \\
&= \sum_{k=1}^{n}\sum_{p=1}^{m^{(l_1+1)}}\left[\frac{\partial}{\partial w_{1,j_2}^{(L+1)}}\left(\frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l_1+1)}}\right) w_{p,i_1}^{(l_1+1)} + \frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l_1+1)}} \frac{\partial w_{p,i_1}^{(l_1+1)}}{\partial w_{1,j_2}^{(L+1)}}\right]\sigma'\left(s_{i_1,k}^{(l_1)}\right) z_{j_1,k}^{(l_1-1)}
\end{aligned}$$

If $l_1 < L$, or $l_1 = L$, $j_2 \neq i_1$ :

$$\begin{aligned}
\frac{\partial^2 \mathcal{L}}{\partial w_{i_1,j_1}^{(l_1)} \partial w_{1,j_2}^{(L+1)}} &= \sum_{k=1}^{n}\sum_{p=1}^{m^{(l_1+1)}}\left[\frac{\partial}{\partial w_{1,j_2}^{(L+1)}}\left(\frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l_1+1)}}\right) w_{p,i_1}^{(l_1+1)}\right]\sigma'\left(s_{i_1,k}^{(l_1)}\right) z_{j_1,k}^{(l_1-1)} \\
&= \sum_{k=1}^{n}\sum_{p=1}^{m^{(l_1+1)}} \frac{\partial}{\partial z_{1,k}^{(L+1)}}\left(\frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l_1+1)}}\right)\frac{\partial z_{1,k}^{(L+1)}}{\partial s_{1,k}^{(L+1)}} \frac{\partial s_{1,k}^{(L+1)}}{\partial w_{1,j_2}^{(L+1)}} w_{p,i_1}^{(l_1+1)} \sigma'\left(s_{i_1,k}^{(l_1)}\right) z_{j_1,k}^{(l_1-1)} \\
&= \sum_{k=1}^{n}\sum_{p=1}^{m^{(l_1+1)}} \frac{\partial}{\partial y_{pred,k}}\left(\frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l_1+1)}}\right)\frac{\partial y_{pred,k}}{\partial s_{1,k}^{(L+1)}} \frac{\partial s_{1,k}^{(L+1)}}{\partial w_{1,j_2}^{(L+1)}} w_{p,i_1}^{(l_1+1)} \sigma'\left(s_{i_1,k}^{(l_1)}\right) z_{j_1,k}^{(l_1-1)} \\
&= \sum_{k=1}^{n}\sum_{p=1}^{m^{(l_1+1)}} \frac{\partial}{\partial s_{p,k}^{(l_1+1)}}\left(\frac{\partial \mathcal{L}}{\partial y_{pred,k}}\right)\frac{\partial y_{pred,k}}{\partial s_{1,k}^{(L+1)}} \frac{\partial s_{1,k}^{(L+1)}}{\partial w_{1,j_2}^{(L+1)}} w_{p,i_1}^{(l_1+1)} \sigma'\left(s_{i_1,k}^{(l_1)}\right) z_{j_1,k}^{(l_1-1)} \\
&= \sum_{k=1}^{n}\sum_{p=1}^{m^{(l_1+1)}} \frac{\partial}{\partial y_{pred,k}}\left(\frac{\partial \mathcal{L}}{\partial s_{p,k}^{(l_1+1)}}\right) w_{p,i_1}^{(l_1+1)} \sigma'\left(s_{i_1,k}^{(l_1)}\right)\sigma'\left(s_{1,k}^{(L+1)}\right) z_{j_1,k}^{(l_1-1)} z_{j_2,k}^{(L)} \\
&= \sum_{k=1}^{n}\sum_{p=1}^{m^{(l+1)}} \frac{\partial}{\partial s_{p,k}^{(l_1+1)}}\left(\frac{1}{n}(y_{pred,k} - y_k)\right) w_{p,i_1}^{(l_1+1)} \sigma'\left(s_{i_1,k}^{(l_1)}\right)\sigma'\left(s_{1,k}^{(L+1)}\right) z_{j_1,k}^{(l_1-1)} z_{j_2,k}^{(L)} \\
&= \frac{1}{n}\sum_{k=1}^{n}\sum_{p=1}^{m^{(l+1)}} \frac{\partial y_{pred,k}}{\partial s_{p,k}^{(l_1+1)}} w_{p,i_1}^{(l_1+1)} \sigma'\left(s_{i_1,k}^{(l_1)}\right)\sigma'\left(s_{1,k}^{(L+1)}\right) z_{j_1,k}^{(l_1-1)} z_{j_2,k}^{(L)}
\end{aligned}$$

$$\implies \frac{\partial^2 \mathcal{L}}{\partial w_{i_1,j_1}^{(l_1)} \partial w_{1,j_2}^{(L+1)}} = \frac{1}{n}\sum_{k=1}^{n}\sum_{p=1}^{m^{(l_1+1)}} \frac{\partial y_{pred,k}}{\partial s_{p,k}^{(l_1+1)}} w_{p,i_1}^{(l_1+1)} \sigma'\left(s_{i_1,k}^{(l_1)}\right)\sigma'\left(s_{1,k}^{(L+1)}\right) z_{j_1,k}^{(l_1-1)} z_{j_2,k}^{(L)}$$

If $l_1 = L$ and $j_2 = i_1$ :

$$\begin{aligned}
\frac{\partial^2\mathcal{L}}{\partial w^{(L)}_{i_1,j_1}\partial w^{(L+1)}_{1,i_1}} &= \sum_{k=1}^{n}\sum_{p=1}^{m^{(L+1)}}\left[\frac{\partial}{\partial w^{(L+1)}_{1,i_1}}\left(\frac{\partial\mathcal{L}}{\partial s^{(L+1)}_{p,k}}\right)w^{(L+1)}_{p,i_1} + \frac{\partial\mathcal{L}}{\partial s^{(L+1)}_{p,k}}\frac{\partial w^{(L+1)}_{p,i_1}}{\partial w^{(L+1)}_{1,i_1}}\right]\sigma'\left(s^{(L)}_{i_1,k}\right)z^{(L-1)}_{j_1,k}\\
&= \sum_{k=1}^{n}\left[\frac{\partial}{\partial w^{(L+1)}_{1,i_1}}\left(\frac{\partial\mathcal{L}}{\partial s^{(L+1)}_{1,k}}\right)w^{(L+1)}_{1,i_1} + \frac{\partial\mathcal{L}}{\partial s^{(L+1)}_{1,k}}\frac{\partial w^{(L+1)}_{1,i_1}}{\partial w^{(L+1)}_{1,i_1}}\right]\sigma'\left(s^{(L)}_{i_1,k}\right)z^{(L-1)}_{j_1,k}\\
&= \sum_{k=1}^{n}\left[\frac{\partial y_{pred,k}}{\partial s^{(L+1)}_{1,k}}w^{(L+1)}_{1,i_1}\sigma'\left(s^{(L+1)}_{1,k}\right)z^{(L)}_{i_1,k} + \frac{\partial\mathcal{L}}{\partial s^{(L+1)}_{1,k}}\right]\sigma'\left(s^{(L)}_{i_1,k}\right)z^{(L-1)}_{j_1,k}
\end{aligned}$$

$$\implies \frac{\partial^2\mathcal{L}}{\partial w^{(L)}_{i_1,j_1}\partial w^{(L+1)}_{1,i_1}} = \sum_{k=1}^{n}\left[\frac{\partial y_{pred,k}}{\partial s^{(L+1)}_{1,k}}w^{(L+1)}_{1,i_1}\sigma'\left(s^{(L+1)}_{1,k}\right)z^{(L)}_{i_1,k} + \frac{\partial\mathcal{L}}{\partial s^{(L+1)}_{1,k}}\right]\sigma'\left(s^{(L)}_{i_1,k}\right)z^{(L-1)}_{j_1,k}$$

3. When $l_1 \le l_2 \le L$ :

$$\begin{aligned}
\frac{\partial^2\mathcal{L}}{\partial w^{(l_1)}_{i_1,j_1}\partial w^{(l_2)}_{i_2,j_2}} &= \frac{\partial}{\partial w^{(l_2)}_{i_2,j_2}}\left(\sum_{k=1}^{n}\left[\sum_{p_1=1}^{m^{(l_1+1)}}\frac{\partial\mathcal{L}}{\partial s^{(l_1+1)}_{p_1,k}}w^{(l_1+1)}_{p_1,i_1}\sigma'\left(s^{(l_1)}_{i_1,k}\right)\right]z^{(l_1-1)}_{j_1,k}\right)\\
&= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l_1+1)}}\left[\frac{\partial}{\partial w^{(l_2)}_{i_2,j_2}}\left(\frac{\partial\mathcal{L}}{\partial s^{(l_1+1)}_{p_1,k}}\right)w^{(l_1+1)}_{p_1,i_1}\sigma'\left(s^{(l_1)}_{i_1,k}\right) + \frac{\partial\mathcal{L}}{\partial s^{(l_1+1)}_{p_1,k}}\frac{\partial w^{(l_1+1)}_{p_1,i_1}}{\partial w^{(l_2)}_{i_2,j_2}}\sigma'\left(s^{(l_1)}_{i_1,k}\right)\right.\\
&\qquad\left. + \frac{\partial\mathcal{L}}{\partial s^{(l_1+1)}_{p_1,k}}w^{(l_1+1)}_{p_1,i_1}\frac{\partial}{\partial w^{(l_2)}_{i_2,j_2}}\left(\sigma'\left(s^{(l_1)}_{i_1,k}\right)\right)\right]z^{(l_1-1)}_{j_1,k}
\end{aligned}$$

If $l_1 < l_2 - 1$, or if $l_1 = l_2 - 1$, $j_2 \neq i_1$, or if $l_1 = l_2$, $i_1 \neq i_2$ :

$$\begin{aligned}
\frac{\partial^2\mathcal{L}}{\partial w^{(l_1)}_{i_1,j_1}\partial w^{(l_2)}_{i_2,j_2}} &= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l_1+1)}}\left[\frac{\partial}{\partial w^{(l_2)}_{i_2,j_2}}\left(\frac{\partial\mathcal{L}}{\partial s^{(l_1+1)}_{p_1,k}}\right)w^{(l_1+1)}_{p_1,i_1}\sigma'\left(s^{(l_1)}_{i_1,k}\right)\right]z^{(l_1-1)}_{j_1,k}\\
&= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l_1+1)}}\frac{\partial}{\partial s^{(l_2)}_{i_2,k}}\left(\frac{\partial\mathcal{L}}{\partial s^{(l_1+1)}_{p_1,k}}\right)\frac{\partial s^{(l_2)}_{i_2,k}}{\partial w^{(l_2)}_{i_2,j_2}}w^{(l_1+1)}_{p_1,i_1}\sigma'\left(s^{(l_1)}_{i_1,k}\right)z^{(l_1-1)}_{j_1,k}\\
&= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l_1+1)}}\frac{\partial}{\partial s^{(l_2)}_{i_2,k}}\left(\frac{\partial\mathcal{L}}{\partial s^{(l_1+1)}_{p_1,k}}\right)w^{(l_1+1)}_{p_1,i_1}\sigma'\left(s^{(l_1)}_{i_1,k}\right)z^{(l_1-1)}_{j_1,k}z^{(l_2-1)}_{j_2,k}\\
&= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l_1+1)}}\sum_{p_2=1}^{m^{(l_2+1)}}\frac{\partial}{\partial s^{(l_2+1)}_{p_2,k}}\left(\frac{\partial\mathcal{L}}{\partial s^{(l_1+1)}_{p_1,k}}\right)\frac{\partial s^{(l_2+1)}_{p_2,k}}{\partial s^{(l_2)}_{i_2,k}}w^{(l_1+1)}_{p_1,i_1}\sigma'\left(s^{(l_1)}_{i_1,k}\right)z^{(l_1-1)}_{j_1,k}z^{(l_2-1)}_{j_2,k}\\
&= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l_1+1)}}\sum_{p_2=1}^{m^{(l_2+1)}}\frac{\partial}{\partial s^{(l_2+1)}_{p_2,k}}\left(\frac{\partial\mathcal{L}}{\partial s^{(l_1+1)}_{p_1,k}}\right)w^{(l_1+1)}_{p_1,i_1}w^{(l_2+1)}_{p_2,i_2}\sigma'\left(s^{(l_1)}_{i_1,k}\right)\sigma'\left(s^{(l_2)}_{i_2,k}\right)z^{(l_1-1)}_{j_1,k}z^{(l_2-1)}_{j_2,k}
\end{aligned}$$

$$\implies \frac{\partial^2\mathcal{L}}{\partial w^{(l_1)}_{i_1,j_1}\partial w^{(l_2)}_{i_2,j_2}} = \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l_1+1)}}\sum_{p_2=1}^{m^{(l_2+1)}}\frac{\partial^2\mathcal{L}}{\partial s^{(l_1+1)}_{p_1,k}\partial s^{(l_2+1)}_{p_2,k}}w^{(l_1+1)}_{p_1,i_1}w^{(l_2+1)}_{p_2,i_2}\sigma'\left(s^{(l_1)}_{i_1,k}\right)\sigma'\left(s^{(l_2)}_{i_2,k}\right)z^{(l_1-1)}_{j_1,k}z^{(l_2-1)}_{j_2,k}$$

If $l_1 = l_2 = l$ and $i_1 = i_2 = i$ :

$$\begin{aligned}
\frac{\partial^2 \mathcal{L}}{\partial w^{(l)}_{i,j_1}\partial w^{(l)}_{i,j_2}} &= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l+1)}}\left[\frac{\partial}{\partial w^{(l)}_{i,j_2}}\left(\frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\right) w^{(l+1)}_{p_1,i}\sigma'\left(s^{(l)}_{i,k}\right) + \frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}} w^{(l+1)}_{p_1,i}\frac{\partial}{\partial w^{(l)}_{i,j_2}}\left(\sigma'\left(s^{(l)}_{i,k}\right)\right)\right] z^{(l-1)}_{j_1,k} \\
&= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l+1)}}\left[\frac{\partial}{\partial w^{(l)}_{i,j_2}}\left(\frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\right)\sigma'\left(s^{(l)}_{i,k}\right) + \frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\frac{\partial}{\partial w^{(l)}_{i,j_2}}\left(\sigma'\left(s^{(l)}_{i,k}\right)\right)\right] w^{(l+1)}_{p_1,i} z^{(l-1)}_{j_1,k} \\
&= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l+1)}}\left[\frac{\partial}{\partial w^{(l)}_{i,j_2}}\left(\frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\right)\sigma'\left(s^{(l)}_{i,k}\right) + \frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\frac{\partial}{\partial s^{(l)}_{i,k}}\left(\sigma'\left(s^{(l)}_{i,k}\right)\right)\frac{\partial s^{(l)}_{i,k}}{\partial w^{(l)}_{i,j_2}}\right] w^{(l+1)}_{p_1,i} z^{(l-1)}_{j_1,k} \\
&= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l+1)}}\left[\frac{\partial}{\partial w^{(l)}_{i,j_2}}\left(\frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\right)\sigma'\left(s^{(l)}_{i,k}\right) + \frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\sigma''\left(s^{(l)}_{i,k}\right) z^{(l-1)}_{j_2,k}\right] w^{(l+1)}_{p_1,i} z^{(l-1)}_{j_1,k} \\
&= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l+1)}}\left[\sum_{p_2=1}^{m^{(l+1)}}\frac{\partial^2 \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}\partial s^{(l+1)}_{p_2,k}} w^{(l+1)}_{p_2,i}\sigma'\left(s^{(l)}_{i,k}\right)^2 + \frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\sigma''\left(s^{(l)}_{i,k}\right)\right] w^{(l+1)}_{p_1,i} z^{(l-1)}_{j_1,k} z^{(l-1)}_{j_2,k}
\end{aligned}$$

$$\Longrightarrow \frac{\partial^2 \mathcal{L}}{\partial w^{(l)}_{i,j_1}\partial w^{(l)}_{i,j_2}} = \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l+1)}}\left[\sum_{p_2=1}^{m^{(l+1)}}\frac{\partial^2 \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}\partial s^{(l+1)}_{p_2,k}} w^{(l+1)}_{p_2,i}\sigma'\left(s^{(l)}_{i,k}\right)^2 + \frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\sigma''\left(s^{(l)}_{i,k}\right)\right] w^{(l+1)}_{p_1,i} z^{(l-1)}_{j_1,k} z^{(l-1)}_{j_2,k}$$

If $l_1 = l_2 - 1 = l$ and $j_2 = i_1$ :

$$\begin{aligned}
\frac{\partial^2 \mathcal{L}}{\partial w^{(l)}_{i_1,j_1}\partial w^{(l+1)}_{i_2,i_1}} &= \sum_{k=1}^{n}\sum_{p_1=1}^{m^{(l+1)}}\left[\frac{\partial}{\partial w^{(l+1)}_{i_2,i_1}}\left(\frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\right) w^{(l+1)}_{p_1,i_1}\sigma'\left(s^{(l)}_{i_1,k}\right) + \frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\frac{\partial w^{(l+1)}_{p_1,i_1}}{\partial w^{(l+1)}_{i_2,i_1}}\sigma'\left(s^{(l)}_{i_1,k}\right)\right] z^{(l-1)}_{j_1,k} \\
&= \sum_{k=1}^{n}\left[\sum_{p_1=1}^{m^{(l+1)}}\frac{\partial}{\partial w^{(l+1)}_{i_2,i_1}}\left(\frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}}\right) w^{(l+1)}_{p_1,i_1}\sigma'\left(s^{(l)}_{i_1,k}\right) + \frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{i_2,k}}\sigma'\left(s^{(l)}_{i_1,k}\right)\right] z^{(l-1)}_{j_1,k} \\
&= \sum_{k=1}^{n}\left[\sum_{p_1=1}^{m^{(l+1)}}\sum_{p_2=1}^{m^{(l+2)}}\frac{\partial^2 \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}\partial s^{(l+2)}_{p_2,k}} w^{(l+1)}_{p_1,i_1} w^{(l+2)}_{p_2,i_2}\sigma'\left(s^{(l+1)}_{i_2,k}\right) z^{(l)}_{i_1,k} + \frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{i_2,k}}\right]\sigma'\left(s^{(l)}_{i_1,k}\right) z^{(l-1)}_{j_1,k}
\end{aligned}$$

$$\Longrightarrow \frac{\partial^2 \mathcal{L}}{\partial w^{(l)}_{i_1,j_1}\partial w^{(l+1)}_{i_2,i_1}} = \sum_{k=1}^{n}\left[\sum_{p_1=1}^{m^{(l+1)}}\sum_{p_2=1}^{m^{(l+2)}}\frac{\partial^2 \mathcal{L}}{\partial s^{(l+1)}_{p_1,k}\partial s^{(l+2)}_{p_2,k}} w^{(l+1)}_{p_1,i_1} w^{(l+2)}_{p_2,i_2}\sigma'\left(s^{(l+1)}_{i_2,k}\right) z^{(l)}_{i_1,k} + \frac{\partial \mathcal{L}}{\partial s^{(l+1)}_{i_2,k}}\right]\sigma'\left(s^{(l)}_{i_1,k}\right) z^{(l-1)}_{j_1,k}$$

# Appendix C

# Additional Figures to Chapter 3

In this appendix, we include figures that are not essential for understanding the main text, but may be helpful for a deeper understanding of the phenomena we discuss. These figures illustrate details that are not visible in the figures presented in Chapter 3. The structure of this appendix follows the same organization as Chapter 3. Specifically, we begin with results for unidimensional orthogonal data, followed by those for non-orthogonal data.

## C.1 Dataset 1.1.0 : original data

### C.1.1 Neuron representation in $(w_{j,1}, w_{j,2})$ plane

Here, we provide neuron representations in the $(w_{j,1}, w_{j,2})$ plane for Sigmoid and Tanh for Dataset 1.1.0. For the Tanh activation function, the neuron disposition draws an S-curve, but it is difficult to see due to the difference in scale between the two axes of the curve. These illustrations correspond to Section 3.1.1(iii).

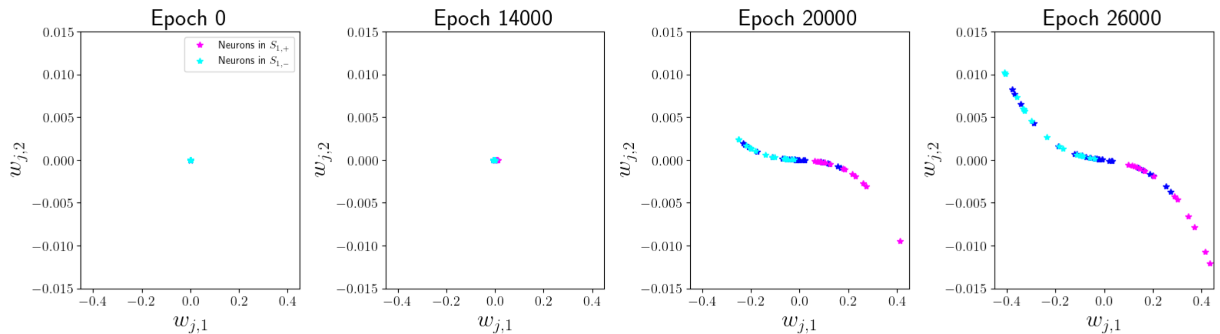



FIGURE C.1: Neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ for a Sigmoid one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-6}$.

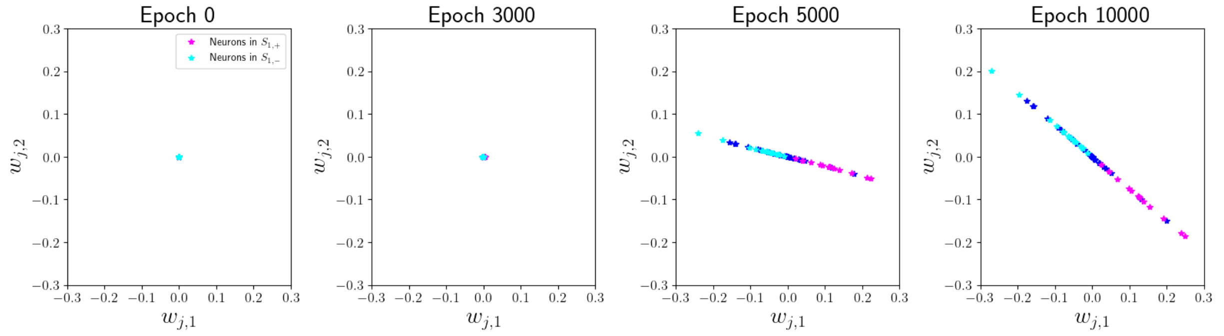



FIGURE C.2: Neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ for a Tanh one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-6}$.

### C.1.2 Balancedness of iterates $\theta^t$

We plot the evolution of $a_j^t$ over epochs for the three activation functions to understand how their values change during training. This experiment relates to the verification of balancedness of iterates $\theta^t$ over training in Section 3.1.1(iv).

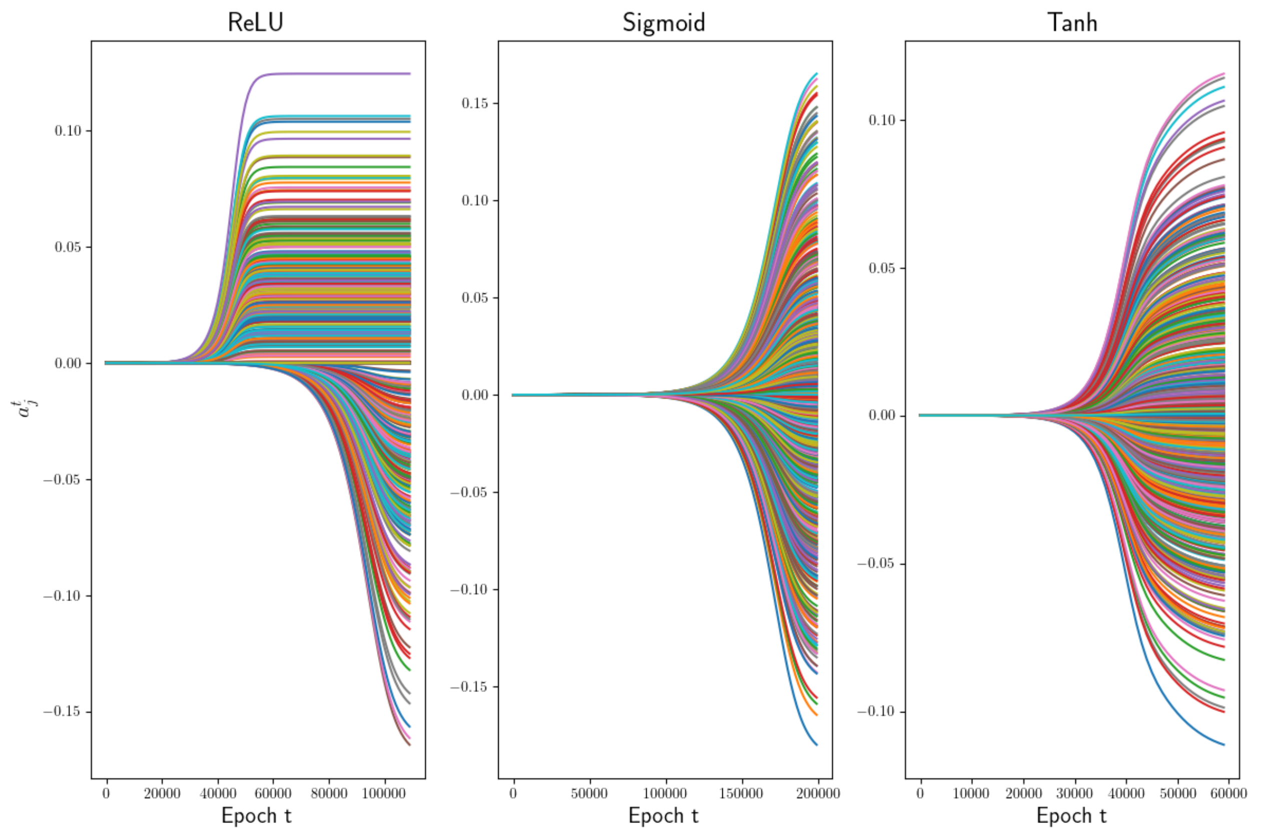



FIGURE C.3: Evolution of each $a_j^t$ over epochs $t$ training one-hidden layer ANNs for the three activation functions. We use Dataset 1.1.0, 1000 neurons and $\lambda = 10^{-6}$.

### C.1.3 Transition phase between Lazy and Rich regimes : illustrations for Sigmoid and Tanh

We display here the figures related to Section 3.1.1(vii), where only figures for ReLU were shown. Here, we provide illustrations for ANNs with Sigmoid and Tanh activation functions.

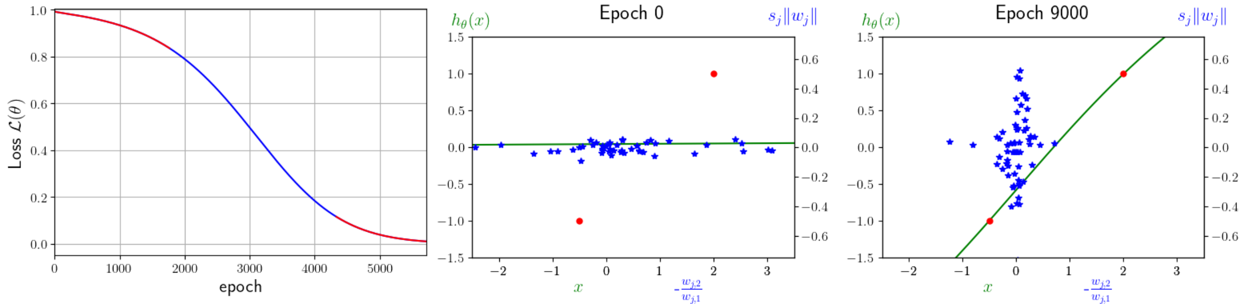



FIGURE C.4: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(X, y)$ for a Sigmoid one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 2.5 \times 10^{-2}$.

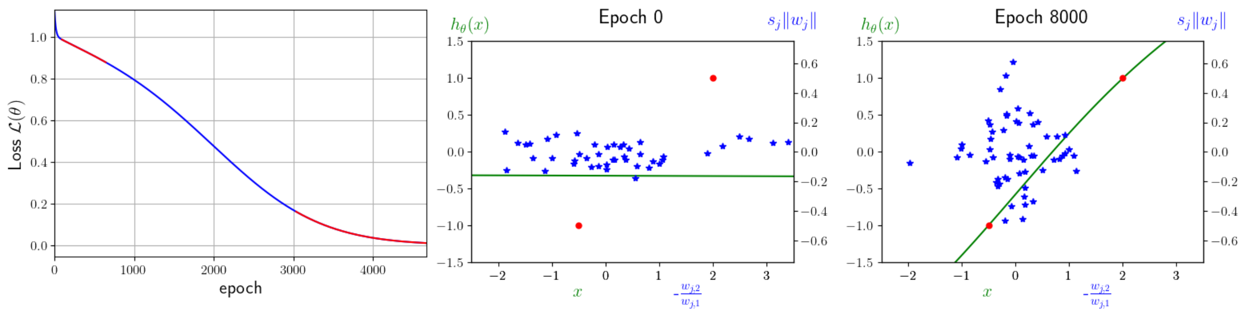



FIGURE C.5: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(X, y)$ for a Sigmoid one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 6 \times 10^{-2}$.

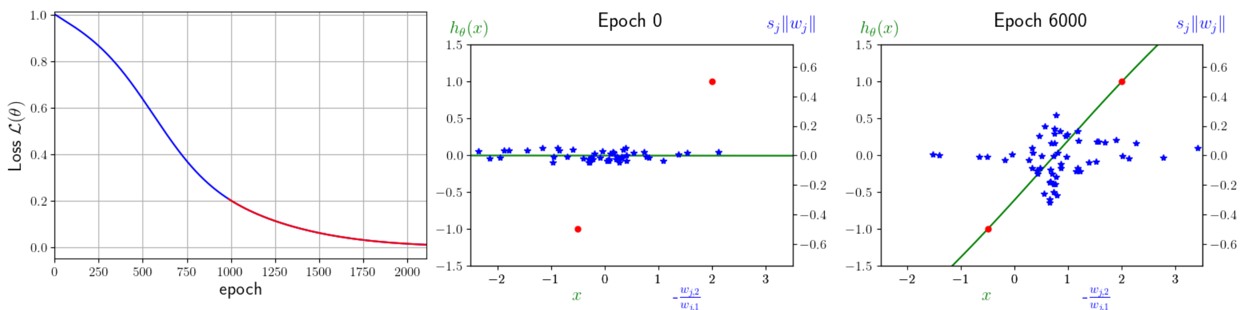



FIGURE C.6: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(X, y)$ for a Tanh one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 2 \times 10^{-2}$.

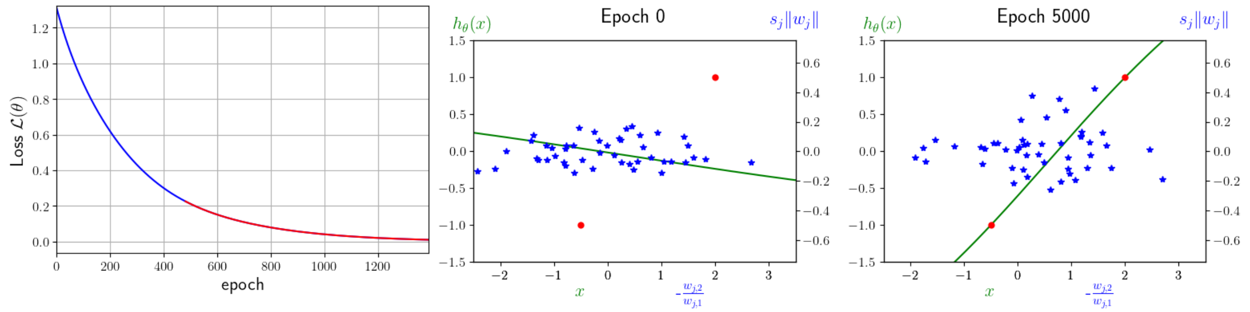



FIGURE C.7: (Left window) Training loss curve, (Right windows) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(X, y)$ for a Tanh one-hidden layer ANN with width 60, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 6 \times 10^{-2}$.

### C.1.4 Automatic processing of experiment results for all generated hyperparameter setups

The following figures are related to Section 3.1.1(vii), where we mention the automatic processing of our numerical experiments. Here, we enter details of the methods we used and their results.

To ensure that our observations are not simply particular cases obtained by chance, we now present the results of automatic processing. After generating all the experiment results obtained with hyperparameters mentioned in the introduction of this chapter, we wanted to verify whether similar results were obtained across different hyperparameter settings. We checked four aspects automatically :

- Convergence to zero loss : We used a threshold of 0.01 for the loss value, and we counted the number of epochs required to achieve $\mathcal{L}(\theta^t) < 0.01$. We denote $T$ as the number of epochs needed to reach this threshold.

- Number of plateaus : We detected the number of plateaus in the training loss curve. To do this, we defined a threshold to differentiate between slow and fast steps in Gradient Method. Because the notion of plateau is independent of the scale of the curve, we cannot use an absolute threshold like for zero loss. Instead, we considered an epoch slow if, for, $\Delta t = \min\left\{1, \frac{T}{100}\right\}$,

$$\frac{\mathrm{d}\mathcal{L}(\theta^t)}{\mathrm{d}t} \approx \frac{\theta^t - \theta^{t-\Delta t}}{\Delta t} < \frac{0.01}{\Delta t}. \tag{C.1}$$

  where $\mathcal{L}(\theta^t)$ represents the training loss curve along the optimization path. To detect a plateau, it is sufficient to detect several adjacent slow epochs. Two distinct plateaus are separated by fast epochs. If there is no convergence to zero loss, we do not attempt to detect plateaus in the curve, resulting in no data on the figure.

- Length of the starting plateau : Using the previous method, we count the number of epochs in the plateau starting at epoch 0.

- Length of the intermediate plateau : In general, in our experiments with orthogonal and quasi-orthogonal data, there is at most one intermediate plateau. We measure its length if it exists. In the rare cases where there are multiple intermediate plateaus, we simply note this in the caption below the figure, which will display the length of the first plateau.

Using these detection methods, we obtained the loss curves (see Figure C.8) with slow epochs marked in red and fast epochs in blue.

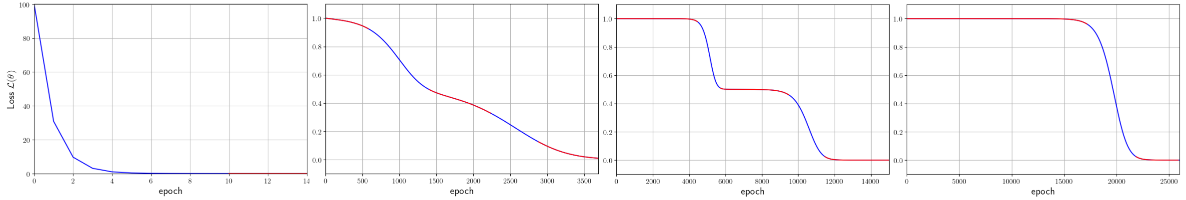



FIGURE C.8: Training loss curve for one-hidden layer ANN with $m = 60$, $\alpha = 0.001$. The three sub-figures on the left are for ReLU with $\lambda = 1, 10^{-2}, 10^{-6}$ respectively. The right sub-figure is for Sigmoid and $\lambda = 10^{-6}$. The red parts of the curves represent slow plateaus.

Below, you will find a summary of our numerical experiments using the detection methods described above. For each activation function, the first column of figures represents the number of epochs to reach zero loss, the second column shows the length of the initial plateau, and the last column indicates the length of the intermediate plateau. Each row corresponds to a different network width.

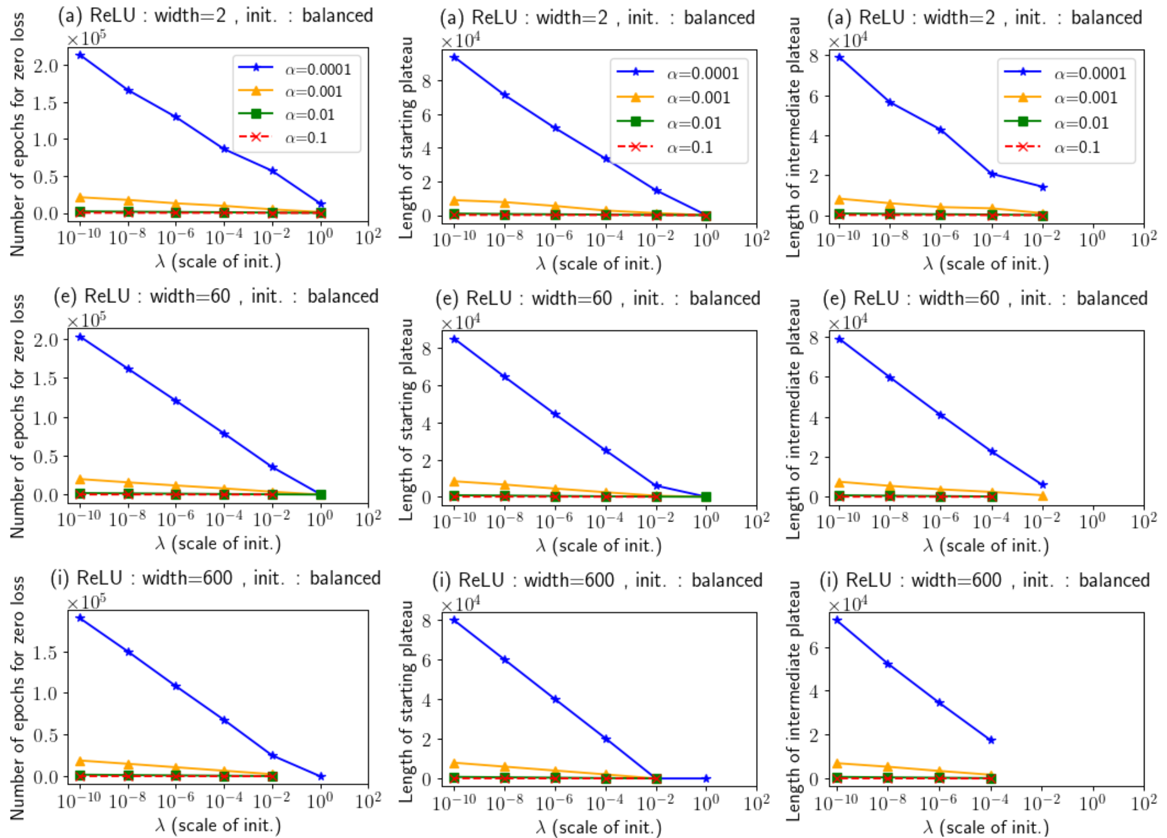



FIGURE C.9: Check of the zero loss and determination of the length of starting and intermediate plateaus for the training of ReLU network with different hyperparameters.[1]

[1]For ReLU activation, we forced the assumption 5 to hold, i.e. $S_{1,+}$ and $S_{1,-}$ non-empty. This ensures the convergence to zero loss as stated in [5]. If this assumption is not guaranteed, we often did not encounter convergence to zero loss for $m \leq 20$.

In Figure C.9 for the ReLU networks, we observe that the number of epochs needed to reach the threshold, as well as the lengths of the initial and intermediate plateaus with the same step-size $\alpha$ are all exponential in $\lambda$ (since the abscissa scale is logarithmic) for all widths. Another observation is that for $\lambda \leq 10^{-4}$, the training under each setting is similar, showing a convergence to zero loss with initial and intermediate plateaus. For higher $\lambda$, as discussed in the previous sections, the nonconvexities in the loss curve disappear. For high value of $\lambda$ (e.g. $\lambda = 100$), the outputs of the ReLU activation explode, starting the training with an initial loss value about several millions, which complicates the training of the ANN. As a result, the network does not converge to zero loss for these initialization scales.

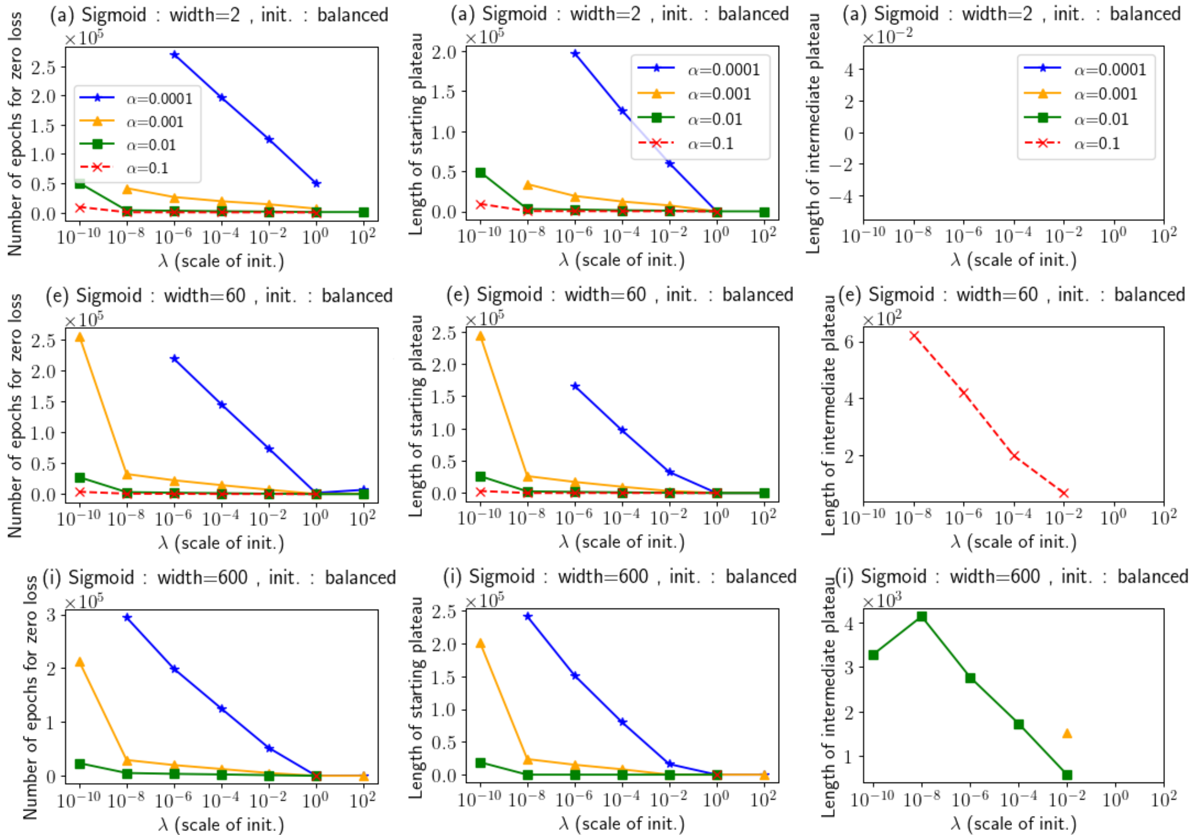



FIGURE C.10: Check of the zero loss and determination of the length of starting and intermediate plateaus for the training of Sigmoid network with different hyperparameters.

In Figure C.10 for the Sigmoid networks, we observe that for low values of the GD step-size $\alpha$, the resulting training dynamics are stable. The model always reaches zero loss and exhibits an initial plateau with no intermediate plateau. For lower values of $\alpha$, the GD becomes unstable, generally leading to an initial increase in the loss before eventually converging to zero loss (see Figure C.11).

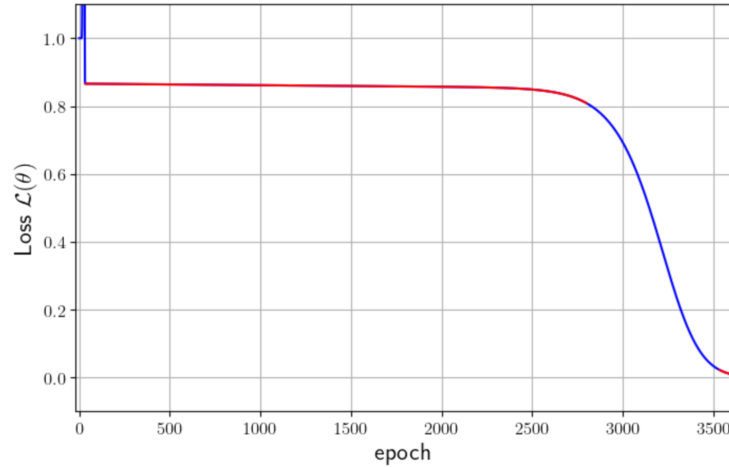



FIGURE C.11: Loss training curve of one-hidden layer Sigmoid ANN with 600 neurons, and trained with $\lambda = 10^{-6}$, and $\alpha = 10^{-2}$. The initial loss value is $\mathcal{L}(\theta^0) \approx 0.9999$ and the maximum loss value reaches $\mathcal{L}(\theta^{26}) \approx 19462.8417$, before converging to zero loss.

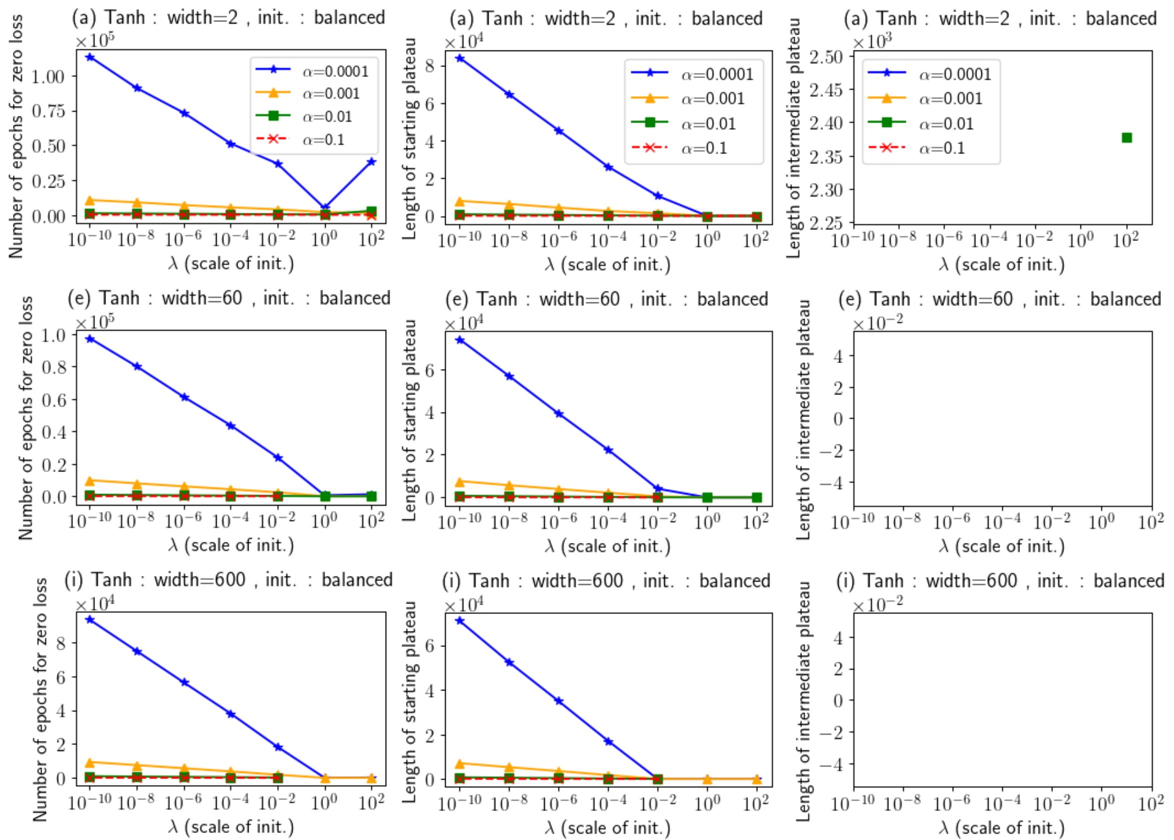



FIGURE C.12: Check of the zero loss and determination of the length of starting and intermediate plateaus for the training of Tanh network with different hyperparameters.

For the Tanh activation function, we observe more stable training dynamics across all the hyperparameter settings used. This confirms that the results presented so far are not coincidental, but occur with high probability, as no alternative behaviors were observed. For all subsequent datasets and hyperparameter settings, we applied the same type of automatic

processing. However, to avoid making the text overly long and repetitive, these results are not presented.

### C.1.5 Curve fitting on S-curves in $(w_{j,1}, w_{j,2})$ plane

We display the same curve fitting as done in Section 3.1.4, but here for Dataset 1.1.0. The rest of the settings remain the same as in the referred section, so we invite you to refer to that section for more details on the methods used.

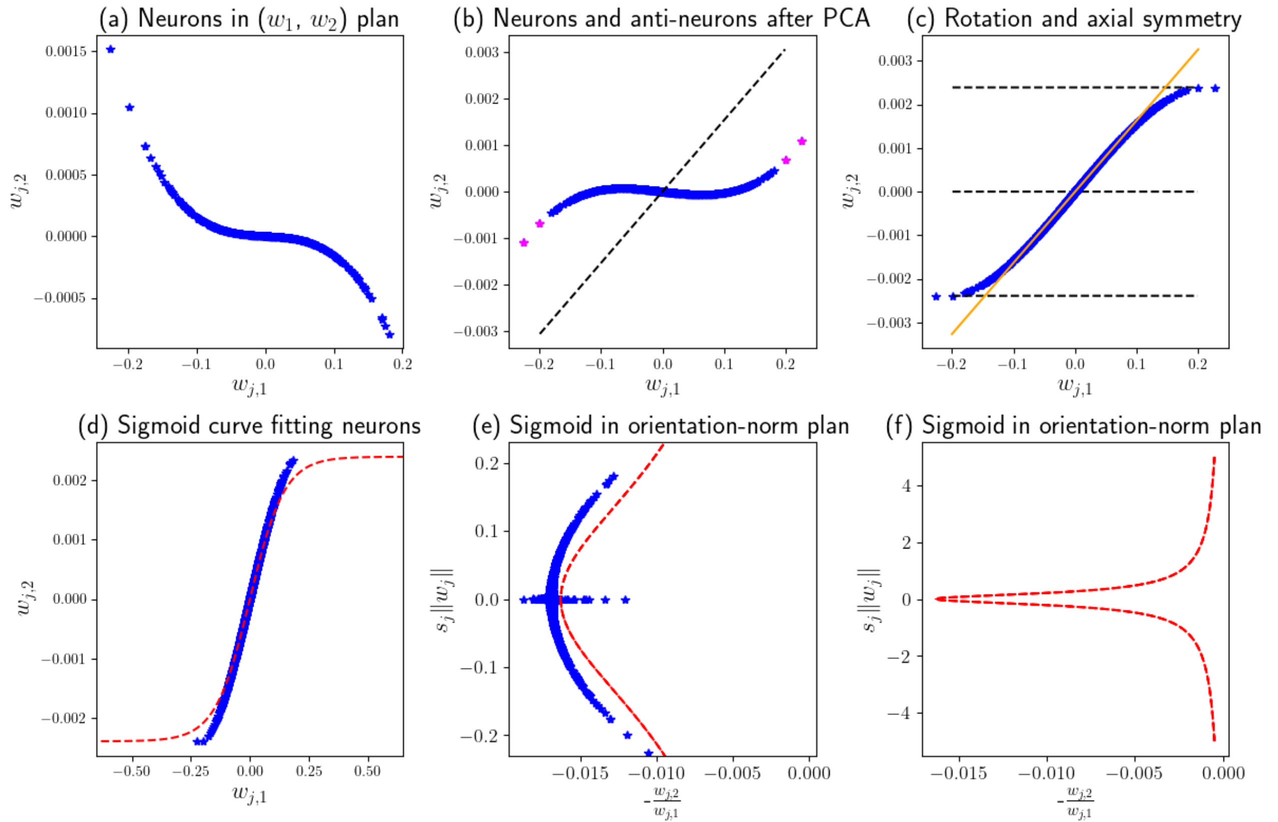



FIGURE C.13: Curve fitting of $f(w)$ on the neuron S-curve in $(w_1, w_2)$ plane for a Sigmoid ANN trained on Dataset 1.1.0. (a) Neuron S-curve composed of several $W^t$. (b) Neurons and anti-neurons[2] pre-rotated with PCA. (c) Final rotation of the plane and axial symmetry. The black dotted lines are the axis of symmetry and the asymptotes of the S-curve, and the orange line is the tangent at the origin. (d) Curve fitting (red dotted line) of the neuron S-curve. (e) Equivalent representation in the orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$ (zoomed on the neurons). (f) Equivalent representation of $f(w)$ in the orientation-signed norm representation.

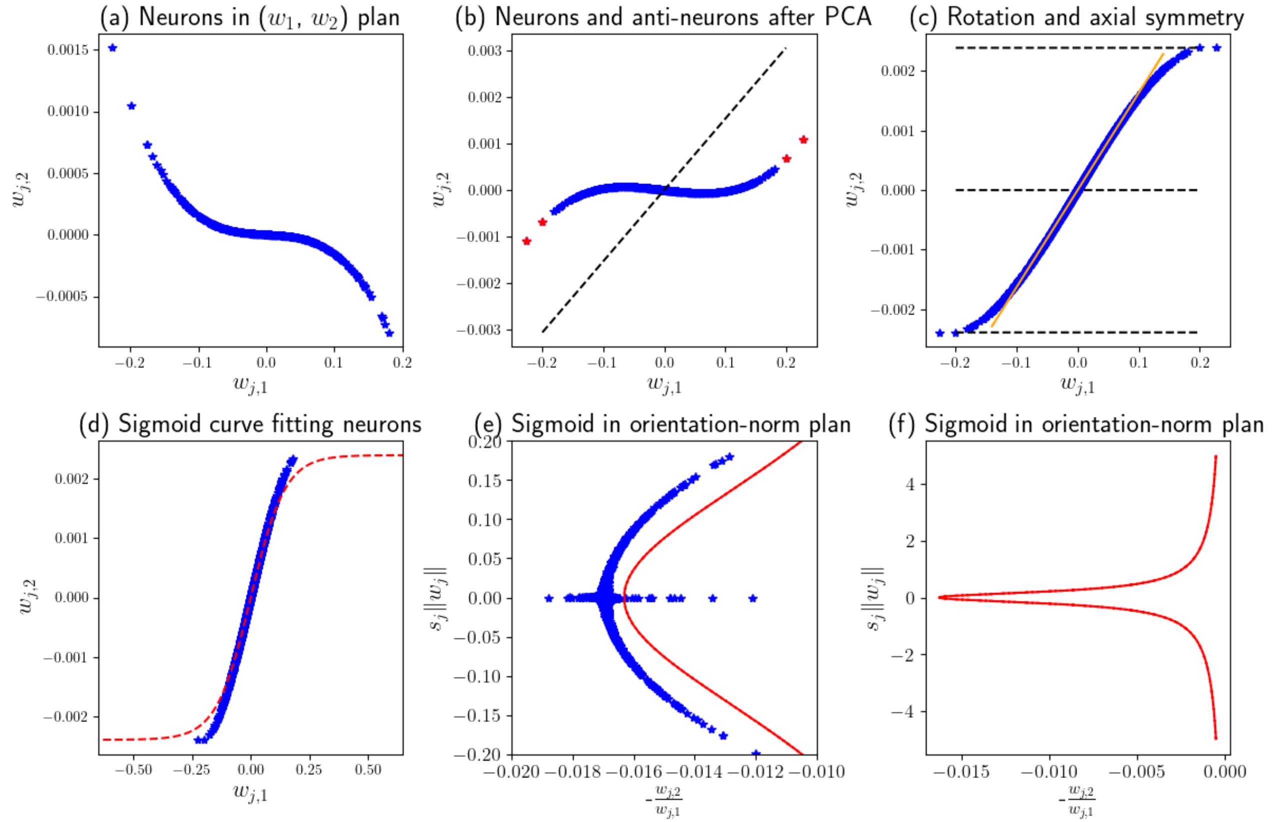



FIGURE C.14: Curve fitting of $f(w)$ on the neuron S-curve in $(w_1, w_2)$ plane for a Tanh ANN trained on Dataset 1.1.0. (a) Neuron S-curve composed of several $W^t$. (b) Neurons and anti-neurons[3] pre-rotated with PCA. (c) Final rotation of the plane and axial symmetry. The black dotted lines are the axis of symmetry and the asymptotes of the S-curve, and the orange line is the tangent at the origin. (d) Curve fitting (red dotted line) of the neuron S-curve. (e) Equivalent representation in the orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$ (zoomed on the neurons). (f) Equivalent representation of $f(w)$ in the orientation-signed norm representation.

## C.2 Dataset 1.1.4 : alternative orthogonal data

The following illustrations correspond to those in Section 3.1.3, but with a more traditional epoch-by-epoch representation. We also include a figure comparing the S-curve for different values of $m$ and $\lambda$ for Tanh (the comparison for Sigmoid is provided in the main text).

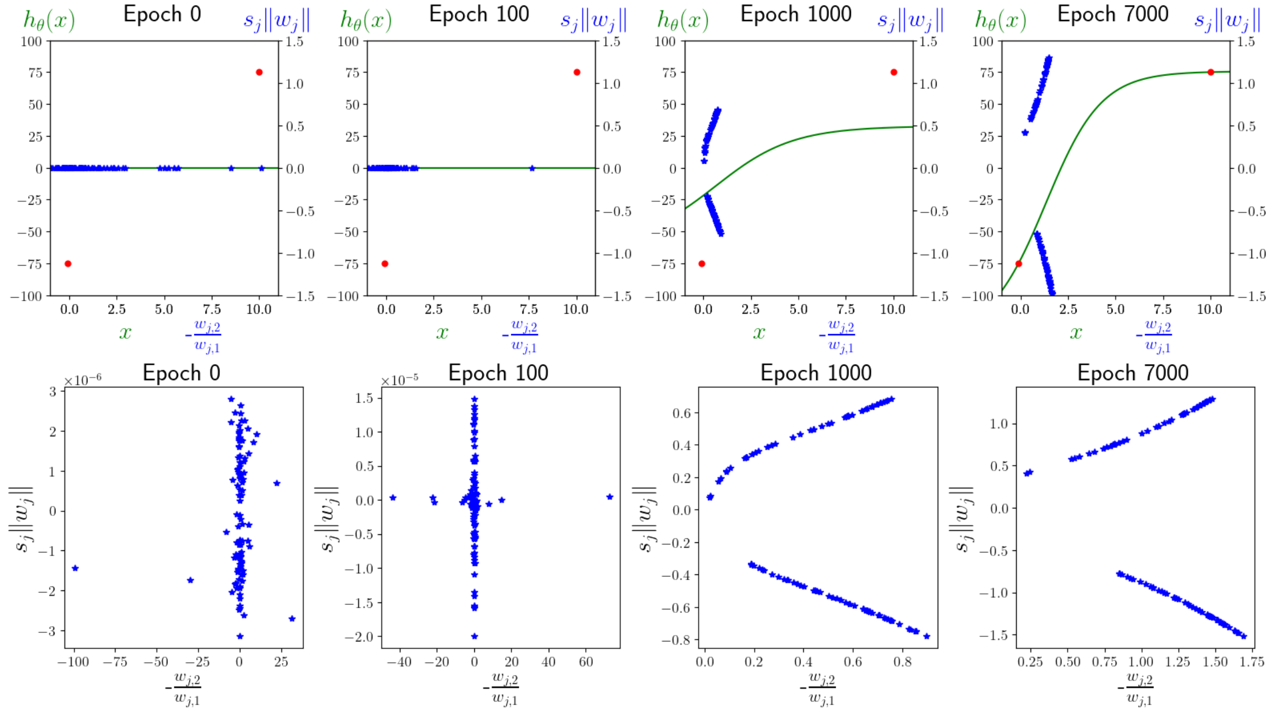



FIGURE C.15: (Dataset 1.1.4 with $c = 75$) Above : Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a Sigmoid one-hidden layer ANN with width 60, $\alpha = 10^{-4}$, and a balanced init. with $\lambda = 10^{-6}$. Below : A zoom on the neuron alignment to see better the shape of neuron alignment.

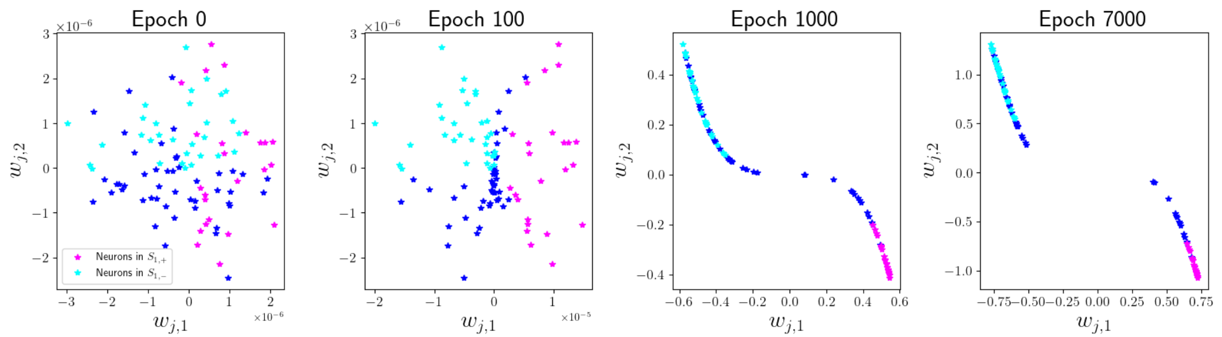



FIGURE C.16: (Dataset 1.1.4 with $c = 75$) Neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ for a Sigmoid one-hidden layer ANN with width 60, $\alpha = 10^{-4}$, and a balanced init. with $\lambda = 10^{-6}$.

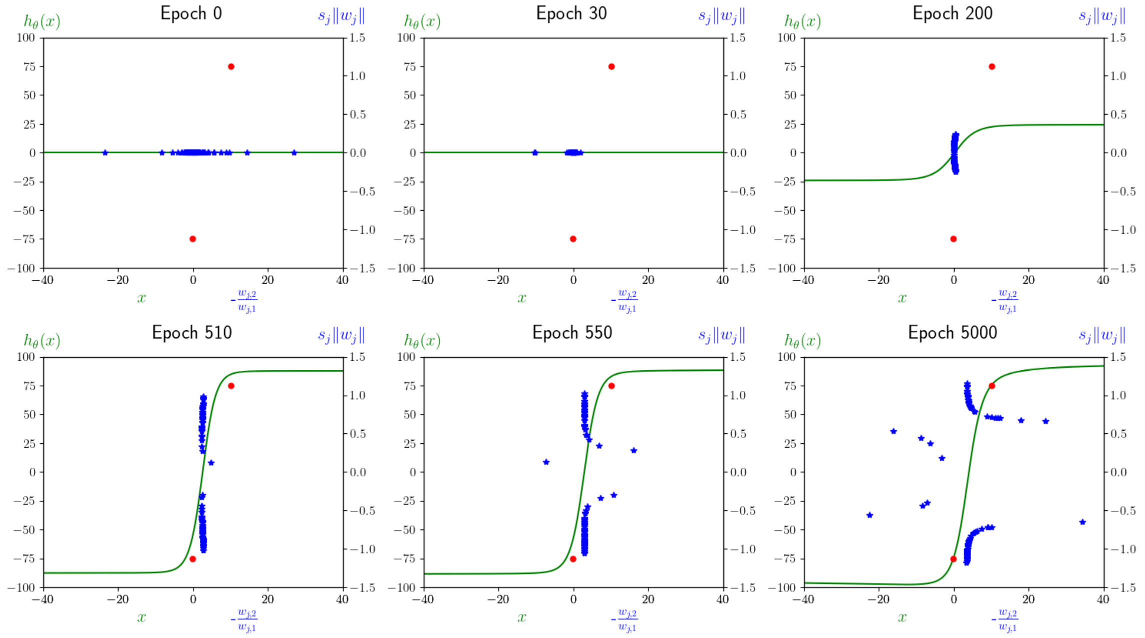



FIGURE C.17: (Dataset 1.1.4 with $c = 75$) Above : Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a Tanh one-hidden layer ANN with width 60, $\alpha = 10^{-4}$, and a balanced init. with $\lambda = 10^{-6}$. Below : A zoom on the neuron alignment to see better the shape of neuron alignment.

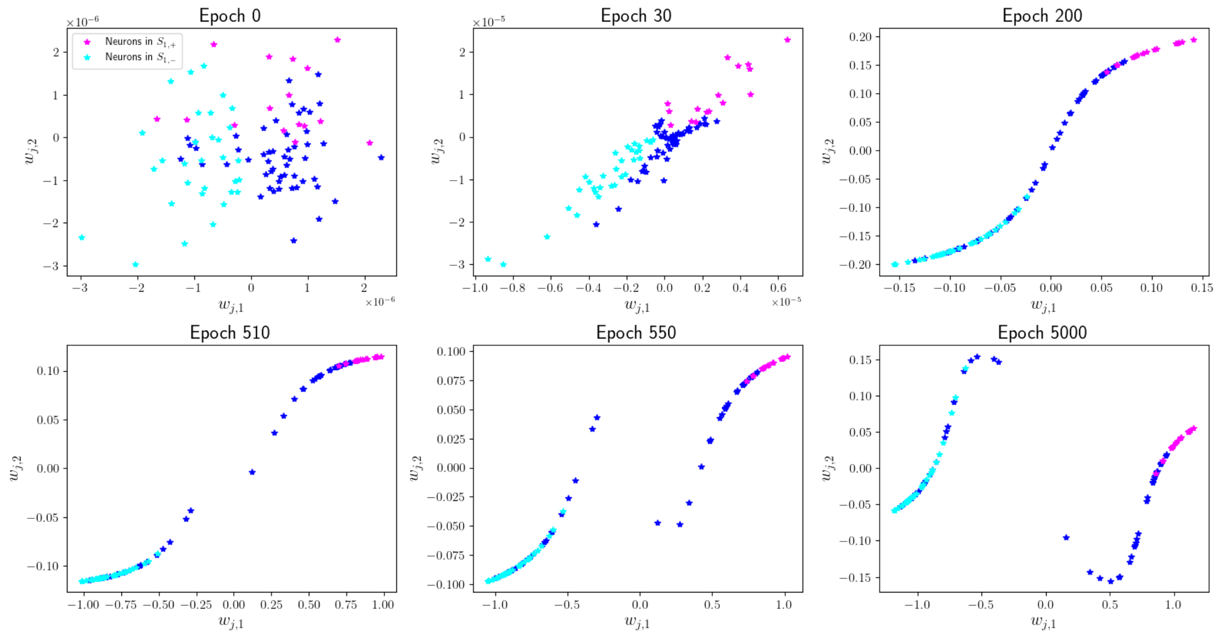



FIGURE C.18: (Dataset 1.1.4 with $c = 75$) Neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ for a Tanh one-hidden layer ANN with width 60, $\alpha = 10^{-4}$, and a balanced init. with $\lambda = 10^{-6}$.

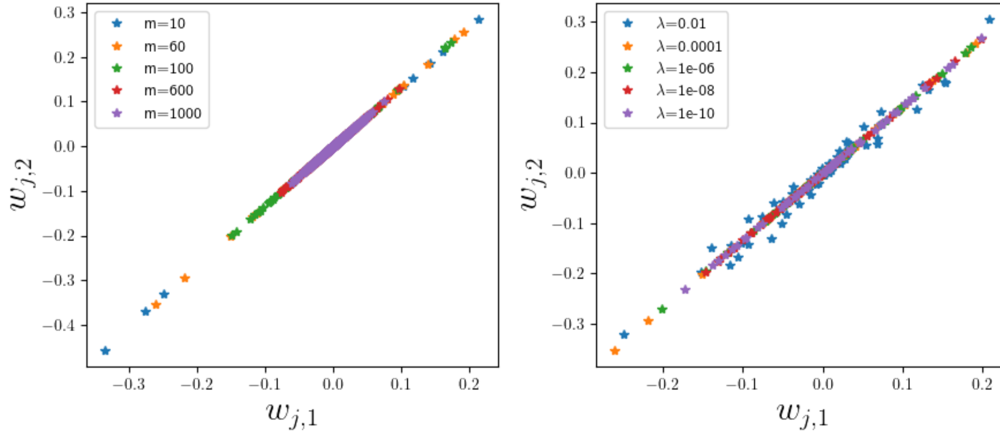



FIGURE C.19: One-hidden layer Tanh ANN with : (Left) different values of $m$ and a fixed $\lambda = 10^{-6}$; (Right) different values of $\lambda$ and a fixed $m = 60$.

## C.3 Unidimensional non-orthogonal data

### C.3.1 Dataset 1.3.1

Here, we provide results for another non-orthogonal dataset. This dataset has the same abscissas as Dataset 1.3.2, but in this case, the positive abscissas correspond to positive labels, and the negative abscissas correspond to negative labels. The dataset is shown below.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{bmatrix} = \begin{bmatrix} 0.99864822 & 1 \\ -0.58678793 & 1 \\ -1.19866293 & 1 \\ 1.59530385 & 1 \\ -0.96831112 & 1 \\ -1.69267688 & 1 \\ 0.13815776 & 1 \\ -0.30366272 & 1 \\ 0.31415876 & 1 \\ 1.93938106 & 1 \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ -1 \\ -1 \\ 1 \\ -1 \\ 1 \\ 1 \end{bmatrix}.$$

The following figures show illustrative results for the three activation functions. For the ReLU activation function, as discussed in Chapter 3, the initial phase of training is similar to that of orthogonal data. Initially, neurons align and then grow in norm. Subsequently, groups of neurons that share the same direction move together to enable the model to fit the data. For Sigmoid and Tanh, the interpretation is more challenging due to the large number of neurons. Although neurons need to move only slightly to fit the data, the parameter space representation reveals a tendency for neurons to align, which represents the first phase of training for these activation functions.

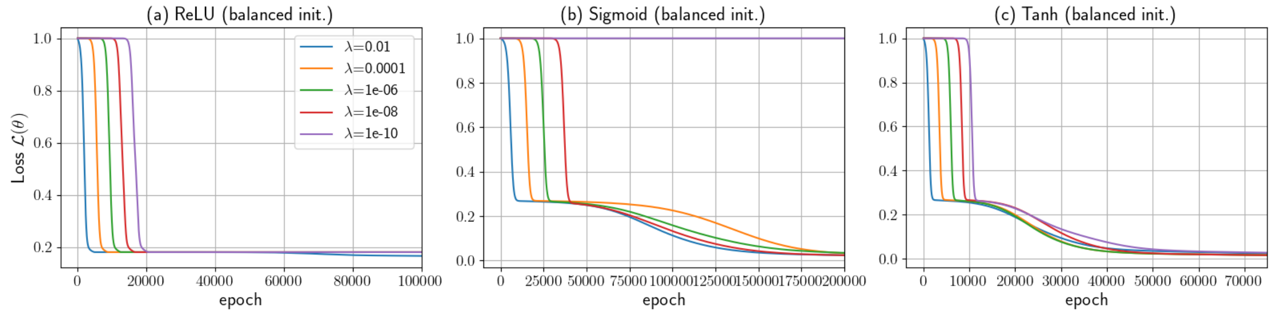



FIGURE C.20: (Dataset 1.3.1) Training loss function for one-hidden layer model with 60 neurons, $\alpha = 10^{-3}$. Comparison with respect to the activation function, the initialization type and scale (in the rich regime).

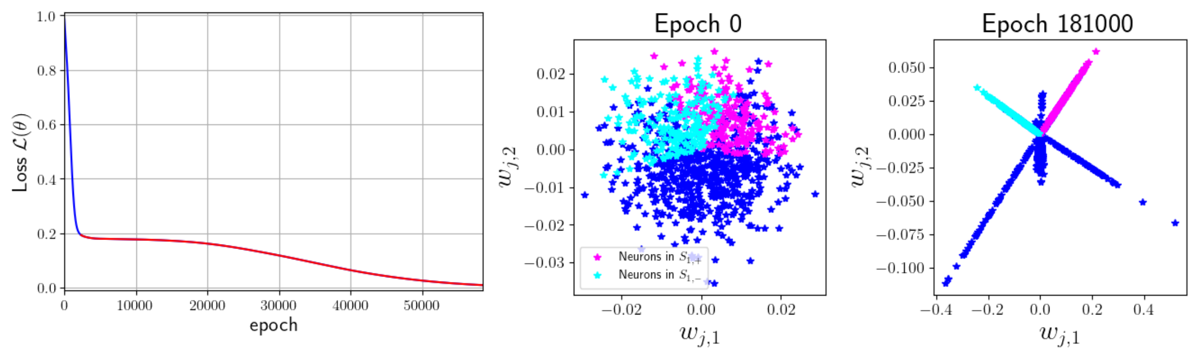



FIGURE C.21: (Dataset 1.3.1) (Left) Training loss curve, (Right) Neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ for a ReLU one-hidden layer ANN with width 1000, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-2}$.

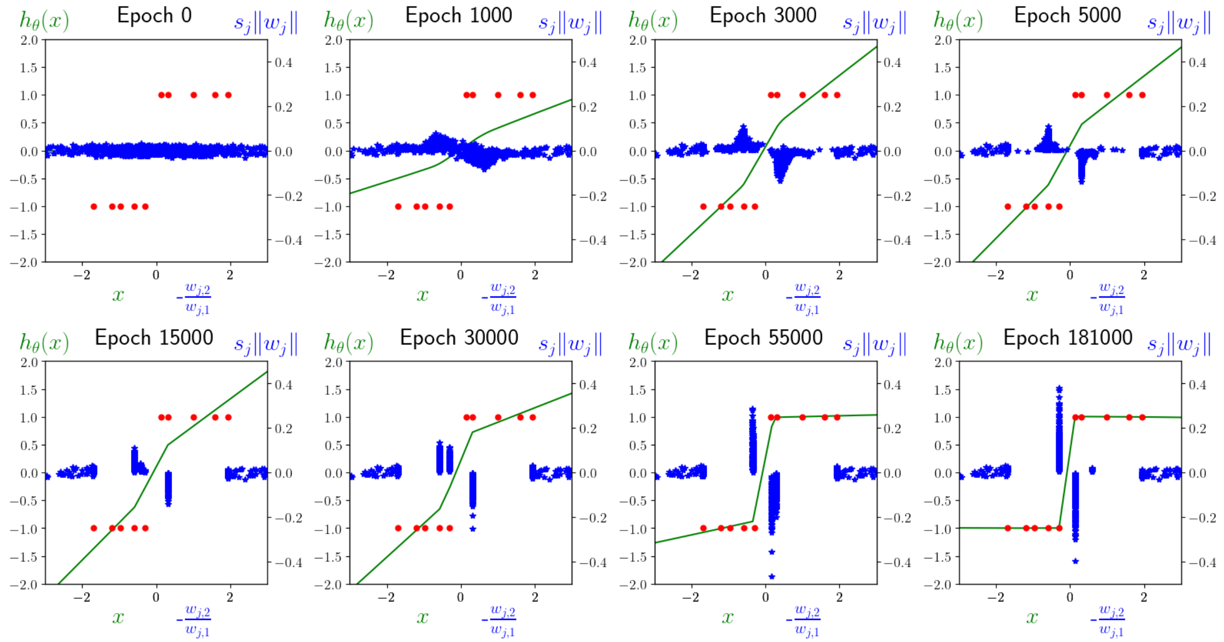



FIGURE C.22: (Dataset 1.3.1) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a ReLU one-hidden layer ANN with width 1000, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-2}$.

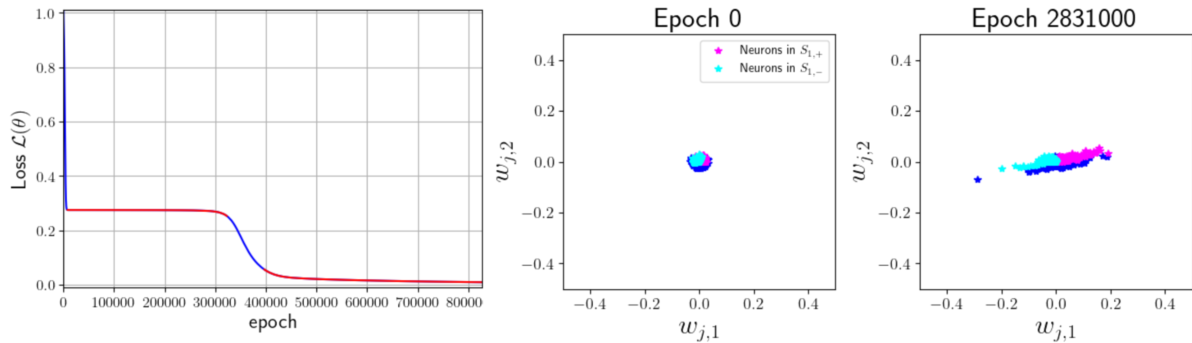



FIGURE C.23: (Dataset 1.3.1) (Left) Training loss curve, (Right) Neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ for a Sigmoid one-hidden layer ANN with width 1000, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-2}$.

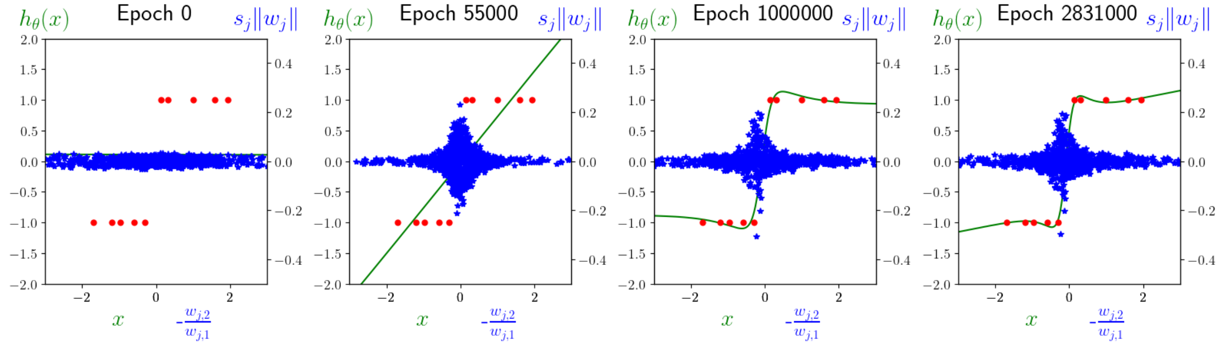



FIGURE C.24: (Dataset 1.3.1) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a Sigmoid one-hidden layer ANN with width 1000, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-2}$.

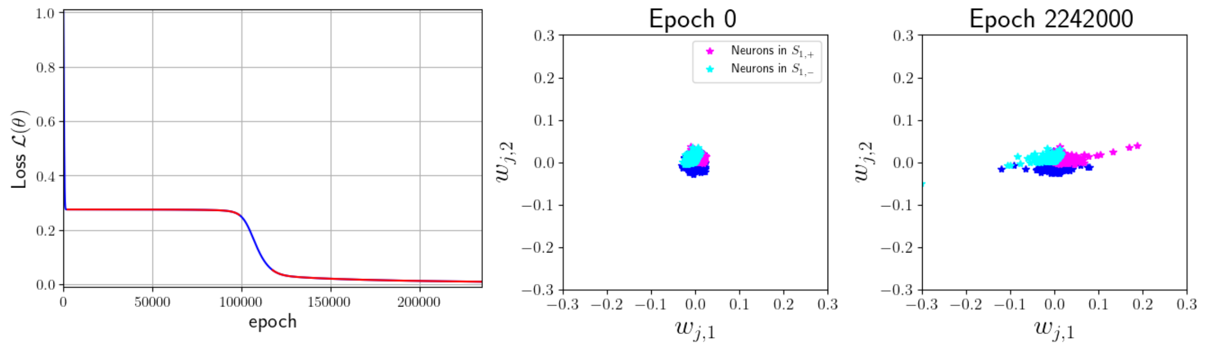



FIGURE C.25: (Dataset 1.3.1) (Left) Training loss curve, (Right) Neurons in their parameter space representation $(w_{j,1}, w_{j,2})$ for a Tanh one-hidden layer ANN with width 1000, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-2}$.

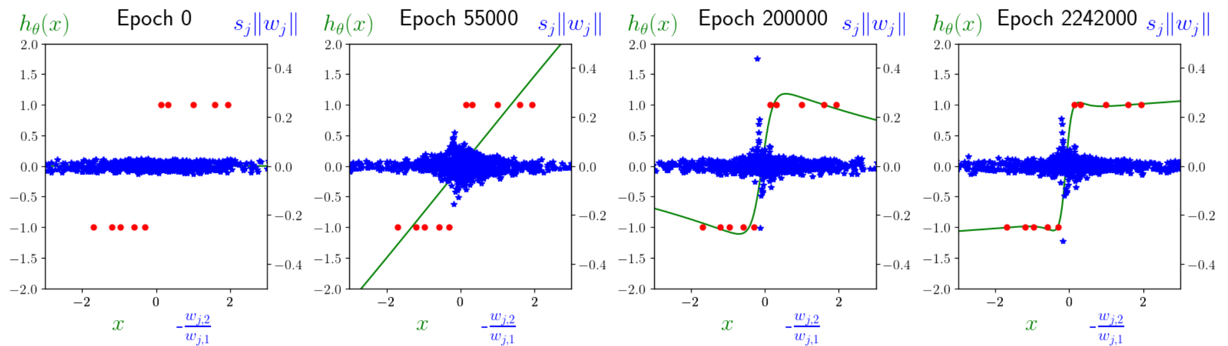



FIGURE C.26: (Dataset 1.3.1) Model $h_\theta(x)$, neurons in their orientation-signed norm representation $\left(-\frac{w_{j,2}}{w_{j,1}}, s_j\|w_j\|\right)$, and data points $(\bar{X}, y)$ for a Tanh one-hidden layer ANN with width 1000, $\alpha = 10^{-3}$, and a balanced init. with $\lambda = 10^{-2}$.

### C.3.2 Dataset 1.3.2

As mentioned in Section 3.3.2, we provide the training loss curve here for Dataset 1.3.2 with different values of $m$. We observe the same dependency of the speed of the different phases

on the width $m$ as seen with orthogonal input data.

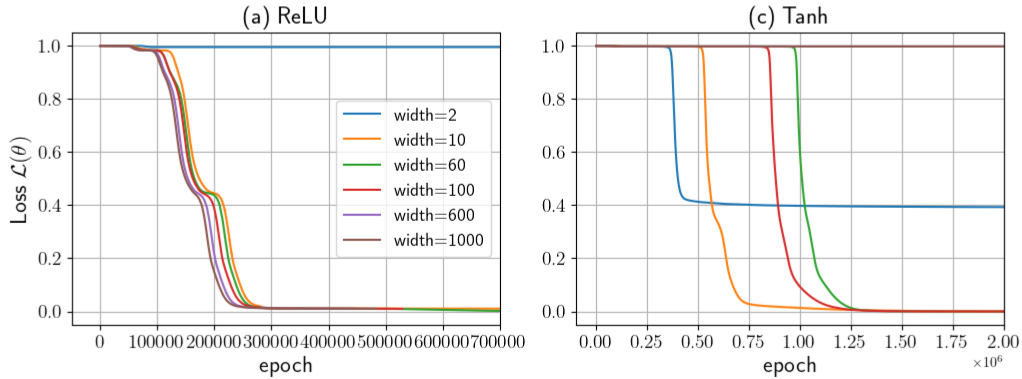



FIGURE C.27: (Dataset 1.3.2) Training loss curves for one-hidden layer model with $\alpha = 10^{-3}$, and balanced init. with $\lambda = 10^{-6}$. Comparison with respect to the width $m$ of the hidden layer (in rich regime).

We also check the balancedness for this non-orthogonal dataset. The results show similarities to those for orthogonal data, except that we observe instabilities in the training due to a step size that is too large to accurately approximate the Gradient Flow. For Sigmoid and Tanh activation functions, there are doubts about the balancedness of iterates throughout the entire training process. If balancedness does not hold, studying the training dynamics for one-hidden layer ANNs with these activation functions must also include examining the output weights, unless another relationship between hidden and output weights is found.

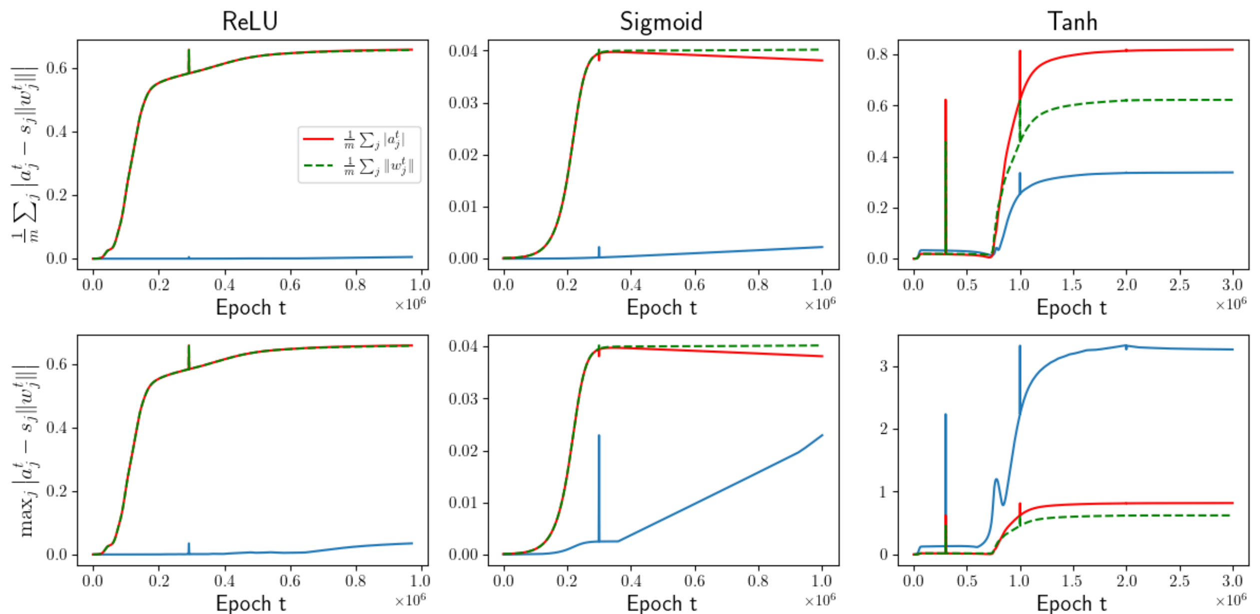



FIGURE C.28: Balancedness of $\theta^t$ throughout the training process for ReLU, Sigmoid and Tanh for non-orthogonal dataset 1.3.2 with 60 neurons, $\lambda = 10^{-4}$ and $\alpha = 10^{-3}$. Upper windows : averaged $\ell_1$-norm of the balancedness error $\frac{1}{m}\sum_{j=1}^{m}\left|a_j^t - s_j\|w_j^t\|\right|$ (blue line). Lower windows : $\ell_\infty$-norm of the balancedness error $\max_{j\in\{1,\cdots,m\}}\left|a_j^t - s_j\|w_j^t\|\right|$ (blue line). In each window, $\frac{1}{m}\left|a_j^t\right|$ (red line) and $\frac{1}{m}\left\|w_j^t\right\|$ (green line) are plotted to provide the order of magnitude of the weights and to facilitate comparison with the computed errors.

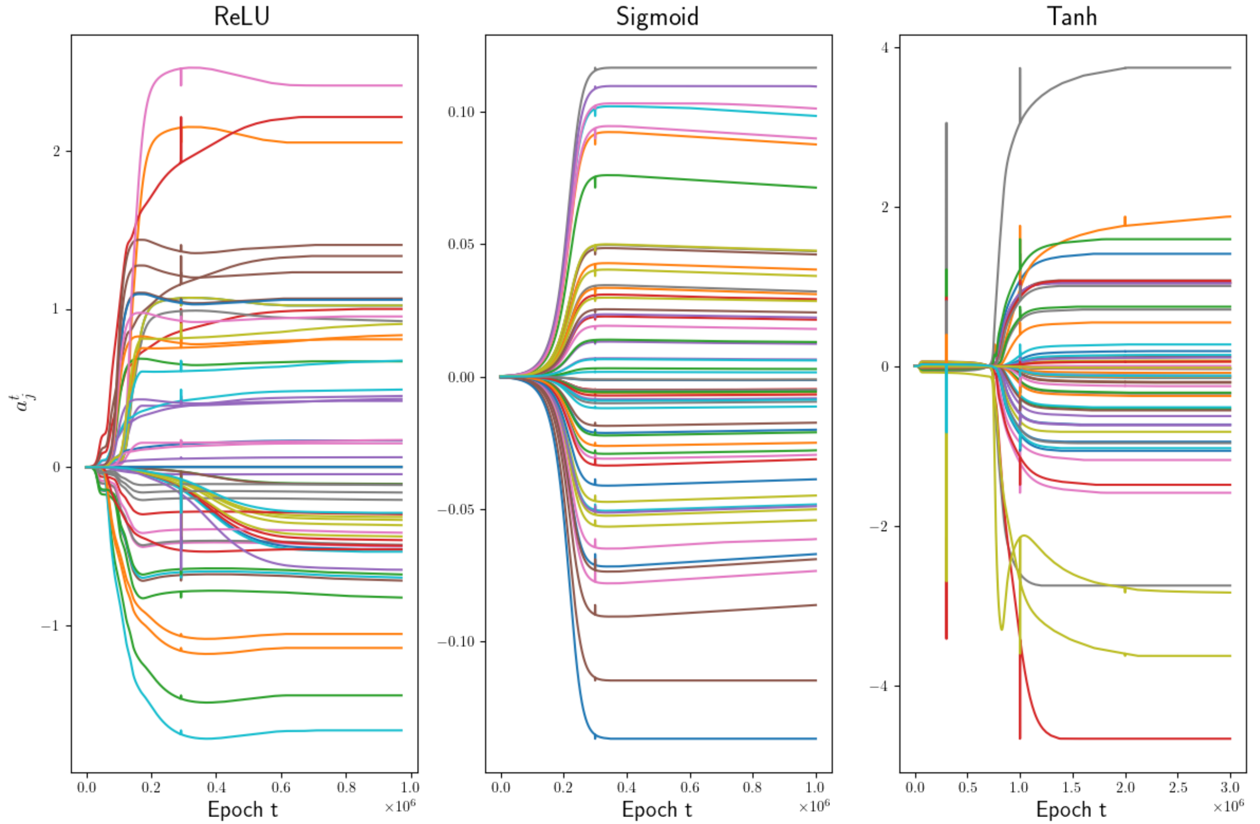



FIGURE C.29: Evolution of each $a_j^t$ over epochs $t$ training one-hidden layer ANNs for the three activation functions. We use Dataset 1.3.2, 60 neurons, $\lambda = 10^{-4}$ and $\alpha = 10^{-3}$.

### C.3.3 Dataset 1.3.3

In this section, we illustrate the transition from the lazy regime to the rich regime when using non-orthogonal input data. The following figures correspond to Section 3.3.1, where we examine model complexity. For all three activation functions, we observe that the loss progressively exhibits phases of slow convergence as $\lambda$. decreases. Moreover, the final interpolator becomes simpler with a decreasing $\lambda$.

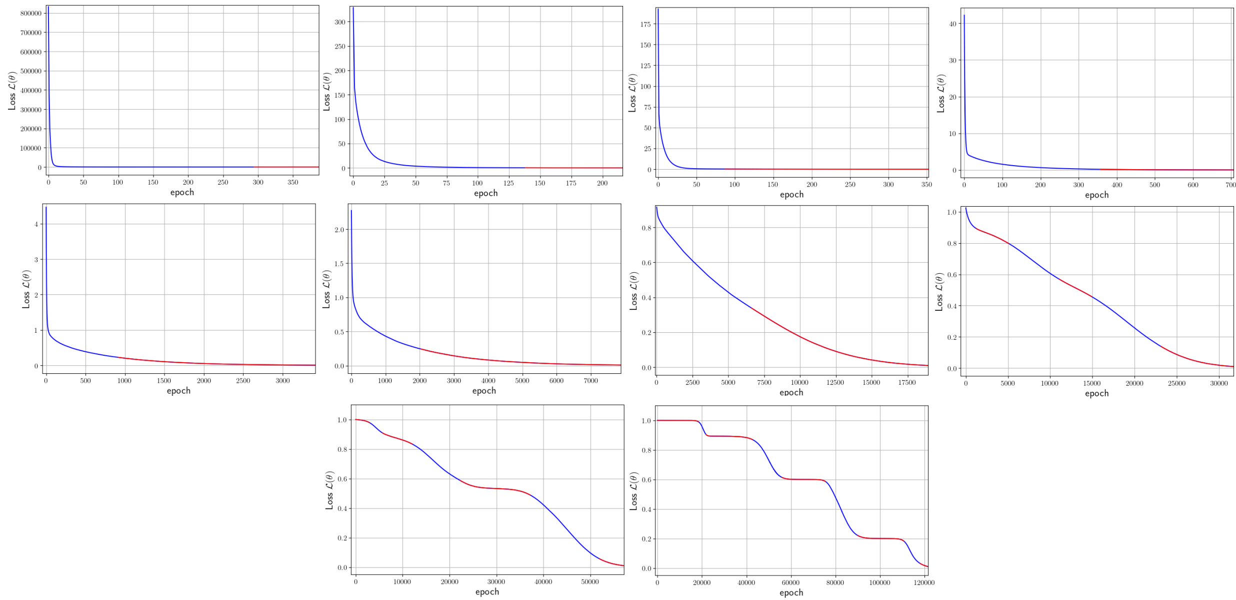



FIGURE C.30: (Dataset 1.3.3) Illustration of the evolution of the training loss curve for a ReLU network as the initialization scale $\lambda$ decreases. Each window corresponds to a different $\lambda$ value from the set $\{10, 2.5, 2, 1, 7.5 \times 10^{-1}, 5 \times 10^{-1}, 2.5 \times 10^{-1}, 10^{-1}, 10^{-2}, 10^{-6}\}$.

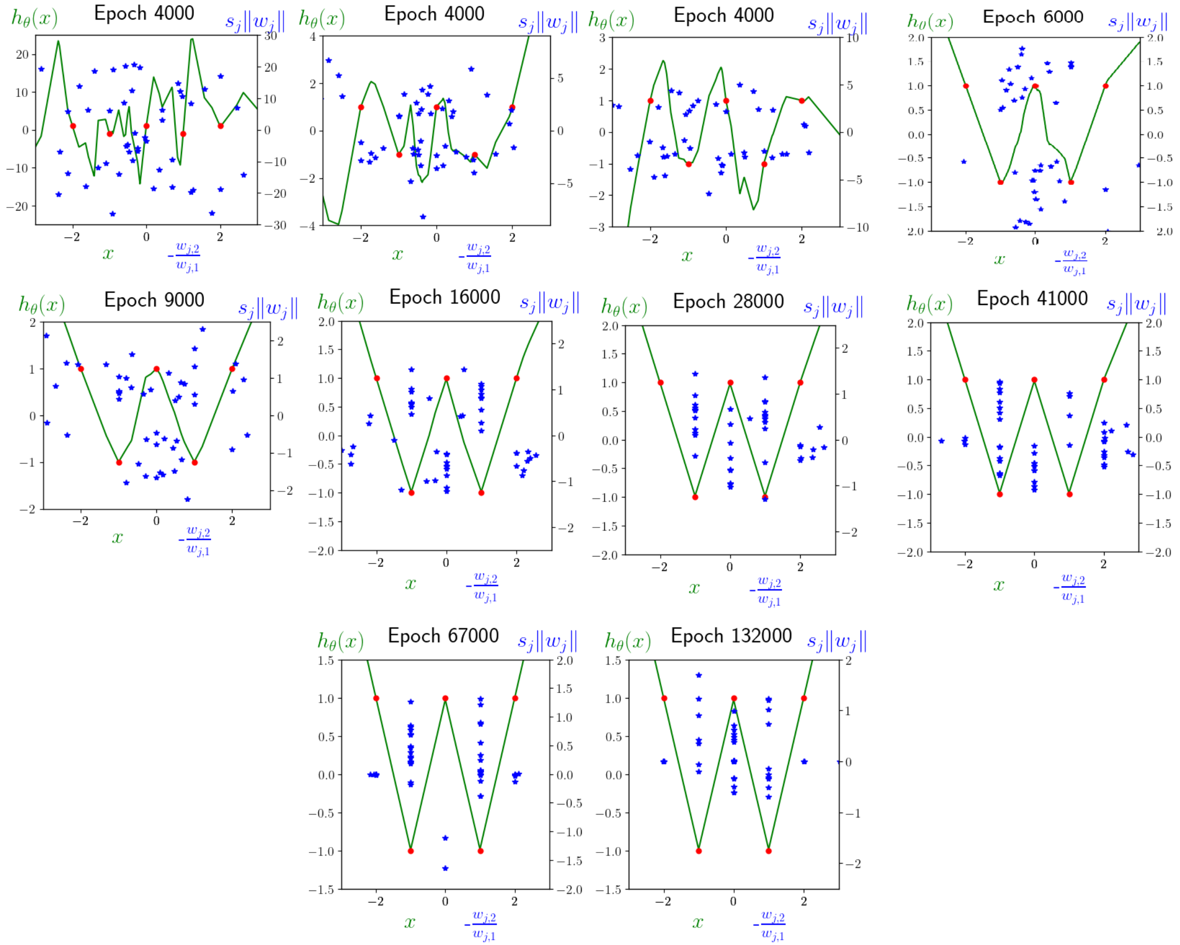



FIGURE C.31: (Dataset 1.3.3) Illustration of the evolution of the final interpolator for a ReLU network as the initialization scale $\lambda$ decreases. Each window corresponds to a different $\lambda$ value from the set $\{10, 2.5, 2, 1, 7.5 \times 10^{-1}, 5 \times 10^{-1}, 2.5 \times 10^{-1}, 10^{-1}, 10^{-2}, 10^{-6}\}$.

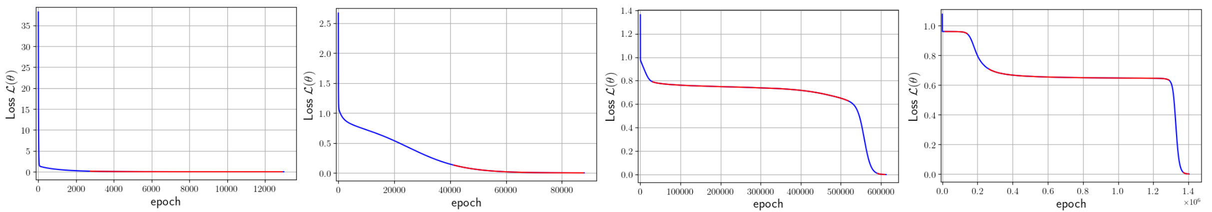



FIGURE C.32: (Dataset 1.3.3) Illustration of the evolution of the training loss curve for a Sigmoid network as the initialization scale $\lambda$ decreases. Each window corresponds to a different $\lambda$ value from the set $\{2, 1, 5 \times 10^{-1}, 10^{-6}\}$.

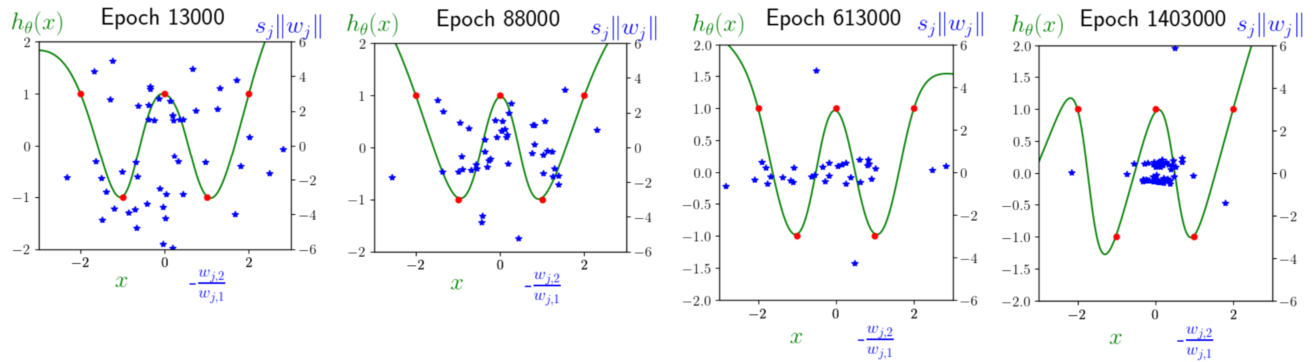



FIGURE C.33: (Dataset 1.3.3) Illustration of the evolution of the final interpolator for a Sigmoid network as the initialization scale $\lambda$ decreases. Each window corresponds to a different $\lambda$ value from the set $\{2, 1, 5 \times 10^{-1}, 10^{-6}\}$.

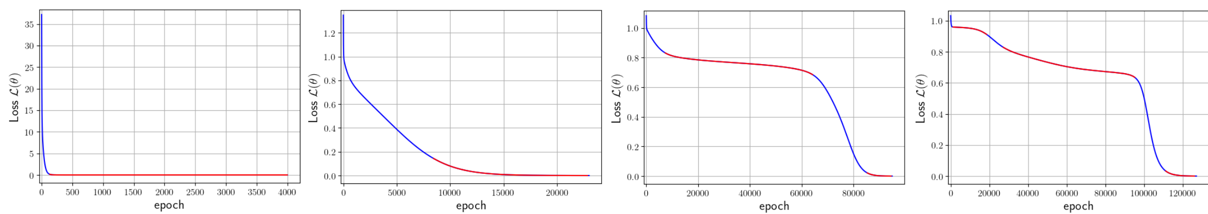



FIGURE C.34: (Dataset 1.3.3) Illustration of the evolution of the training loss curve for a Tanh network as the initialization scale $\lambda$ decreases. Each window corresponds to a different $\lambda$ value from the set $\{2.5, 5 \times 10^{-1}, 2.5 \times 10^{-1}, 10^{-1}\}$.

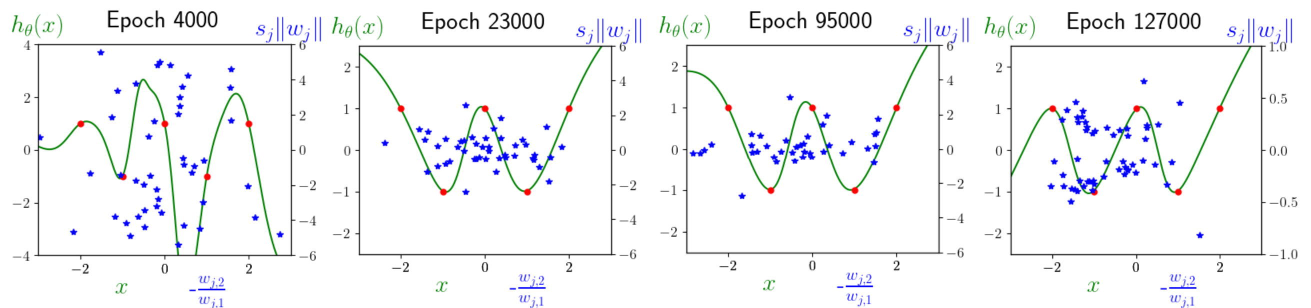



FIGURE C.35: (Dataset 1.3.3) Illustration of the evolution of the final interpolator for a Tanh network as the initialization scale $\lambda$ decreases. Each window corresponds to a different $\lambda$ value from the set $\{2.5, 5 \times 10^{-1}, 2.5 \times 10^{-1}, 10^{-1}\}$.

# Appendix D

# Numerical check of gradient and Hessian formulas

It is important to have a confirmation of our calculation of the formulas of gradient and Hessian in order to avoid calculation errors. To allow this, we try to confirm numerically the calculation of formulas made in Chapter 4. Let us define the numerical lost, gradient and hessian obtained with different methods :

- the loss, its gradient and its hessian computed with PyTorch : $\mathcal{L}_{PT}(\theta)$, $\nabla\mathcal{L}_{PT}(\theta)$, and $\nabla^2\mathcal{L}_{PT}(\theta)$;
- the loss, its gradient and its hessian computed with the formulas of section 4.1 : $\mathcal{L}_F(\theta)$, $\nabla\mathcal{L}_F(\theta)$, and $\nabla^2\mathcal{L}_F(\theta)$;
- and an approximation of the gradient and the hessian of the loss computed by finite difference scheme : $\nabla\mathcal{L}_{App}(\theta)$, and $\nabla^2\mathcal{L}_{App}(\theta)$.

The results are made by training an one-hidden layer ANN with 10 neurons. The coefficient values and order of precision of finite methods we use in this Appendix come from [9] and [8]. We also mention that all the formulas in Chapter 4 have been derived in 3 different ways (direct derivation of $\mathcal{L}$, the formulas for fully-connected neural network in the preliminaries, and via the matrix formulation as presented in Chapter 4), obtaining the same results. This already confirms their validity.

## D.1 Numerical check of the gradient formulas

The finite scheme used to approximate the gradient is the central difference, i.e.

$$\nabla\mathcal{L}_{App}(\theta) = \begin{bmatrix} \frac{L_F(\theta+\frac{1}{2}he_1)-L_F(\theta-\frac{1}{2}he_1)}{h} \\ \vdots \\ \frac{L_F(\theta+\frac{1}{2}he_i)-L_F(\theta-\frac{1}{2}he_i)}{h} \\ \vdots \\ \frac{L_F(\theta+\frac{1}{2}he_{m(d+1)})-L_F(\theta-\frac{1}{2}he_{m(d+1)})}{h} \end{bmatrix}$$

with $h$ the step size, and $e_i = 1$ for the entry $i$, and 0 otherwise. The order of precision of the schema is in $\mathcal{O}(h^2)$.

Firstly, we generate one thousand times a random set of parameters $\theta$ with $\theta_i \sim \lambda \times \mathcal{N}(0,1)$, and we compute the gradient via the formulas and via the finite differences. Then, we compute the $l_2$-error $E$ between the two, and we make the average over the thousand computations. For $h = 10^{-4}$, we obtain this kind of results :

| | $\lVert \nabla \mathcal{L}_F(\theta) - \nabla \mathcal{L}_{App}(\theta) \rVert_2$ | | |
|---|---|---|---|
| $\lambda$ | ReLU | Sigmoid | Tanh |
| $10^{-7}$ | $1.91194 \times 10^{-7}$ | $4.22131 \times 10^{-8}$ | $3.02942 \times 10^{-12}$ |
| $10^{-2}$ | $6.95204 \times 10^{-5}$ | $5.08386 \times 10^{-8}$ | $2.43728 \times 10^{-9}$ |
| 5 | $7.60822 \times 10^{-5}$ | $8.85910 \times 10^{-7}$ | $2.21812 \times 10^{-6}$ |

We expect an error of order $\mathcal{O}(\lVert E \rVert) = \mathcal{O}(\sqrt{\sum_{i=1}^{10}(E_i)^2}) = \mathcal{O}(10^{-7.5})$ for smooth activation functions. This is exactly what we see in the table just above, except for $\lambda = 5$ where the step-size $h$ chosen creates dominant round-off errors. Across all of our verification, we encountered this phenomenon when increasing $\lambda$. However, ReLU is not smooth since it is not differentiable in 0. So, we do not have theoretical accuracy as for smooth function. We can see however that the error remains relatively small.

In a second time, we compute the $l_2$-error and $\infty$-error on the training path between the gradient computed via the formulas, and the gradient returned by PyTorch or the approximation via finite difference. We use the Dataset 1.1.0, $\lambda = 10^{-6}$ and $\alpha = 10^{-3}$. We obtain the following figures :

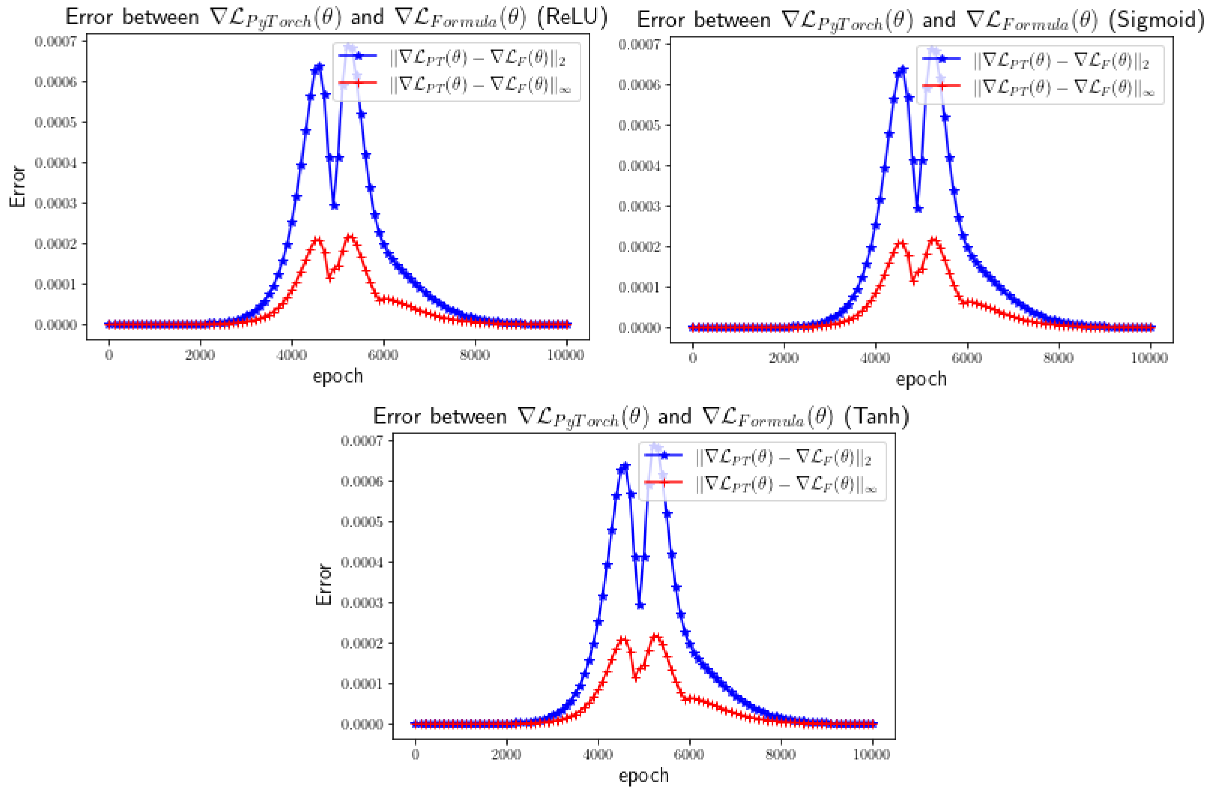



FIGURE D.1: Error between $\nabla\mathcal{L}_{PT}(\theta^t)$ and $\nabla\mathcal{L}_F(\theta^t)$ over the training process for one-hidden layer ANN with 10 neurons, an initialization scale $\lambda = 10^{-6}$, and a GD step-size $\alpha = 10^{-3}$ for ReLU, Sigmoid and Tanh activation functions. The red curve represents the $\infty$-error, and the blue curve represents the $l_2$-error.

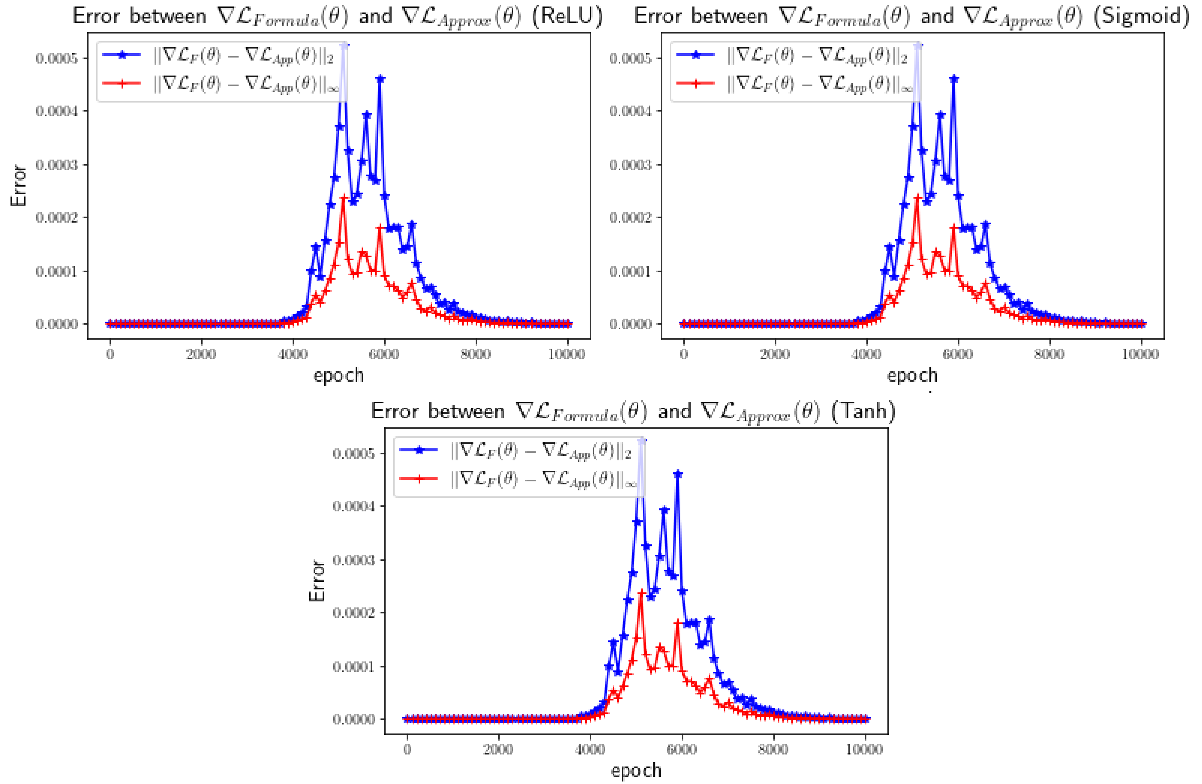



FIGURE D.2: Error between $\nabla\mathcal{L}_{App}(\theta^t)$ and $\nabla\mathcal{L}_F(\theta^t)$ over the training process for one-hidden layer ANN with 10 neurons, an initialization scale $\lambda = 10^{-6}$, and a GD step-size $\alpha = 10^{-3}$ for ReLU, Sigmoid and Tanh activation functions. The red curve represents the $\infty$-error, and the blue curve represents the $l_2$-error.

We can notice that the $l_2$-norm error is always under $7 \times 10^{-4}$, and the error grows when the loss varies more along the training path. We can connect this growth with the implementation of the formulas, which probably causes numerical errors. If we check the error between the loss given by PyTorch $\mathcal{L}_{PT}$ and the loss computed with the formula $\mathcal{L}_F$, we obtain the same kind of curves.

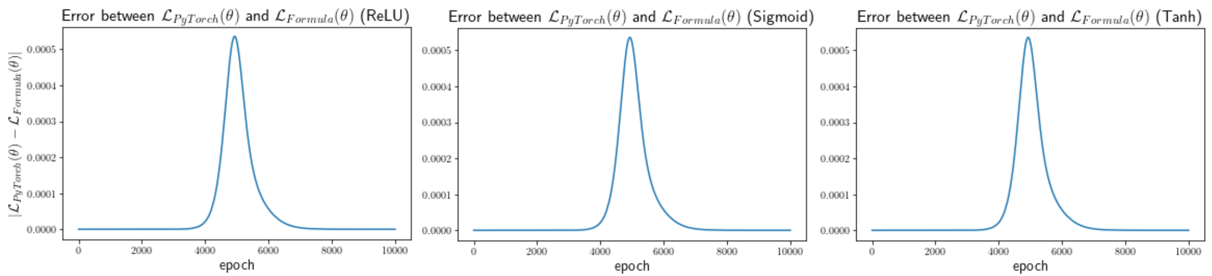



FIGURE D.3: Difference in absolute value between $\mathcal{L}_{PT}(\theta^t)$ and $\mathcal{L}_F(\theta^t)$ over the training process for one-hidden layer ANN with 10 neurons, an initialization scale $\lambda = 10^{-6}$, and a GD step-size $\alpha = 10^{-3}$ for ReLU, Sigmoid and Tanh activation functions.

All these results comfort us in the formulas of the gradient of the loss.

## D.2 Numerical check of the hessian formulas

In order to check numerically the formula for the hessian of the square loss $\nabla^2\mathcal{L}$, we use a difference schema to approximate each partial derivative for each element of the gradient $\nabla\mathcal{L}$. Because of some numerical difficulties, we use a central finite difference schema of order 8. More precisely, the schema is the following

$$\begin{aligned} f'(x) =& \frac{1}{280}f(x-4h) - \frac{4}{105}f(x-3h) + \frac{1}{5}f(x-2h) - \frac{4}{5}f(x-h) \\ &+ \frac{4}{5}f(x+h) - \frac{1}{5}f(x+2h) + \frac{4}{105}f(x+3h) - \frac{1}{280}f(x+4h). \end{aligned}$$

Setting $f = \frac{\partial\mathcal{L}}{\partial\theta_i}$, we obtain

$$\nabla^2\mathcal{L}_{App}(\theta) = \begin{bmatrix} \frac{1}{280}\frac{\partial\mathcal{L}}{\partial\theta_i}(\theta-4h_ie_j) - \frac{4}{105}\frac{\partial\mathcal{L}}{\partial\theta_i}(\theta-3h_ie_j) + \frac{1}{5}\frac{\partial\mathcal{L}}{\partial\theta_i}(\theta-2h_ie_j) - \frac{4}{5}\frac{\partial\mathcal{L}}{\partial\theta_i}(\theta-h_ie_j) \\ +\frac{4}{5}\frac{\partial\mathcal{L}}{\partial\theta_i}(\theta+h_ie_j) - \frac{1}{5}\frac{\partial\mathcal{L}}{\partial\theta_i}(\theta+2h_ie_j) + \frac{4}{105}\frac{\partial\mathcal{L}}{\partial\theta_i}(\theta+3h_ie_j) - \frac{1}{280}\frac{\partial\mathcal{L}}{\partial\theta_i}(\theta+4h_ie_j) \end{bmatrix}_{i,j=1,\cdots,m(d+1)}$$

with $h_i$ the step size, and $e_i = 1$ for the entry $i$, and 0 otherwise. Because the sensitivity to the step size is very different for the weights of the hidden layer $w_{i,j}$ and the weights of the output layer $a_i$, we set $h_i = h_w$ and $h_i = h_a$ for the two kinds of weights.

For example, for $\theta = \begin{bmatrix}\log(4) & \log(1) & \log(9) & \log\left(\frac{1}{3}\right) 1 & -1\end{bmatrix}^T$, the basic orthogonal dataset, and a neural network with $m = 2$ and Sigmoid activation function, we get with our formulas of the Hessian :

$$\nabla^2\mathcal{L}_F(\theta) = \begin{bmatrix} 0.49846213 & 0.47044818 & -0.09157311 & 0.17180604 & -0.02491297 & -0.03120648 \\ 0.47044818 & 0.46992347 & 0.0478305 & 0.03780414 & -0.09394345 & 0.01677828 \\ -0.09157311 & 0.0478305 & 0.12368054 & 0.01785879 & -0.00631329 & 0.00309335 \\ 0.17180604 & 0.03780414 & 0.01785879 & 0.09689236 & 0.00309335 & -0.01095332 \\ -0.02491297 & -0.09394345 & -0.00631329 & 0.00309335 & -0.07315118 & -0.01135684 \\ -0.03120648 & 0.01677828 & 0.00309335 & -0.01095332 & -0.01135684 & -0.05611592 \end{bmatrix}$$

With $h_a = 1$ and $h_w = 10^{-4}$, the approximation with the finite difference method gives

$$\nabla^2\mathcal{L}_{App}(\theta) = \begin{bmatrix} 0.49846209 & 0.47044818 & -0.0915731 & 0.17180604 & -0.02491296 & -0.03120648 \\ 0.47044817 & 0.46992343 & 0.04783051 & 0.03780414 & -0.09394344 & 0.01677827 \\ -0.09189404 & 0.04740521 & 0.12368531 & 0.01790037 & -0.00632838 & 0.00317856 \\ 0.17169331 & 0.03767333 & 0.01787625 & 0.09683803 & 0.0031547 & -0.01098216 \\ -0.02459827 & -0.09342212 & -0.00620864 & 0.0031516 & -0.0731773 & -0.01135956 \\ -0.03090288 & 0.01705509 & 0.00309199 & -0.01095235 & -0.01136098 & -0.05610247 \end{bmatrix}$$

The Frobenius norm of the error is equal to $9.410 \times 10^{-4}$. We expect an error of order $\mathcal{O}(\sqrt{36 \times 10^{-16}}) = \mathcal{O}(6 \times 10^{-8})$.

In a second time, we generate random theta as the same way that in section D.1, and we compute the average of the Frobenius norm of the error.

| | $\|\nabla^2 \mathcal{L}_F(\theta) - \nabla^2 \mathcal{L}_{App}(\theta)\|_F$ | | |
|---|---|---|---|
| $\lambda$ | ReLU | Sigmoid | Tanh |
| $10^{-7}$ | 0.882706 | $2.616767 \times 10^{-4}$ | $1.201516 \times 10^{-7}$ |
| $10^{-2}$ | 2.008855 | $4.529790 \times 10^{-4}$ | $2.384548 \times 10^{-5}$ |
| 5 | 0.331454 | $3.316418 \times 10^{-3}$ | $6.629788 \times 10^{-3}$ |

We can see that the resulting errors are not satisfying to confirm the correctness of our formulas for the Hessian. This is due to the fact that the sensitivity of the error to $h_a$ and $h_w$ is very high, making difficult to find good step-size $h_a$ and $h_w$ to avoid round-off errors and minimizing the truncation error. This shows more that it is difficult to use finite difference scheme to confirm the validity of formulas we derived for the Hessian of the loss.

# Bibliography

[1] Sanjeev Arora et al. "A Convergence Analysis of Gradient Descent for Deep Linear Neural Networks". In: *arXiv* (2018). URL: https://arxiv.org/abs/1810.02281.

[2] Žiga Avsec et al. "Effective gene expression prediction from sequence by integrating long-range interactions". In: *Nature methods* 18.10 (2021), pp. 1196–1203. URL: https://www.nature.com/articles/s41592-021-01252-x.

[3] Anas Barakat and Pascal Bianchi. "Convergence Rates of a Momentum Algorithm with Bounded Adaptive Step Size for Nonconvex Optimization". In: *Asian Conference on Machine Learning* (2020), pp. 225–240. URL: https://arxiv.org/pdf/1911.07596.

[4] Dimitri P Bertsekas. *Nonlinear programming*. https://mcube.lab.nycu.edu.tw/~cfung/docs/books/bertsekas1999nonlinear_programming.pdf(visited 2024-07-01). Athena Scientific, 1999.

[5] Etienne Boursier, Loucas Pillaud-Vivien, and Nicolas Flammarion. "Gradient flow dynamics of shallow ReLU networks for square loss and orthogonal inputs". In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 20105–20118. URL: https://arxiv.org/abs/2206.00939.

[6] Lenaic Chizat, Edouard Oyallon, and Francis Bach. "On Lazy Training in Differentiable Programming". In: *Advances in neural information processing systems* 32 (2019). URL: https://proceedings.neurips.cc/paper/2019/hash/ae614c557843b1df326cb29c57225459-Abstract.html.

[7] Wikipedia contributors. *BERT (language model)*. 2024. URL: https://en.wikipedia.org/w/index.php?title=BERT_(language_model)&oldid=1239944093 (visited on 08/14/2024).

[8] Wikipedia contributors. *Finite difference coefficient*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Finite_difference_coefficient&oldid=1239301196 (visited on 08/15/2024).

[9] Wikipedia contributors. *Finite difference method*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Finite_difference_method&oldid=1211001443 (visited on 08/15/2024).

[10] Wikipedia contributors. *Machine learning*. 2024. URL: https://en.wikipedia.org/w/index.php?title=Machine_learning&oldid=1231414429 (visited on 06/29/2024).

[11] Wikipedia contributors. *Neurone*. 2024. URL: https://fr.wikipedia.org/w/index.php?title=Neurone&oldid=216294500 (visited on 06/29/2024).

[12] Yann N Dauphin and Samuel Schoenholz. "MetaInit: Initializing learning by learning to initialize". In: *Advances in Neural Information Processing Systems* 32 (2019). URL: https://papers.nips.cc/paper_files/paper/2019/file/876e8108f87eb61877c6263228b67256-Paper.pdf.

[13] Yann N Dauphin et al. "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization". In: *Advances in neural information processing systems* 27 (2014). URL: https://arxiv.org/pdf/1406.2572.

[14] Tian Ding, Dawei Li, and Ruoyu Sun. "Sub-Optimal Local Minima Exist for Neural Networks with Almost All Non-Linear Activations". In: *arXiv* (2019). URL: https://arxiv.org/pdf/1911.01413.

[15] Felix Draxler et al. "Essentially No Barriers in Neural Network Energy Landscape". In: *International conference on machine learning* (2018), pp. 1309–1318. URL: https://arxiv.org/pdf/1803.00885.

[16] Haoxing Du. *Inside the mind of a superhuman Go model: How does Leela Zero read ladders?* 2023. URL: https://www.lesswrong.com/posts/FF8i6SLfKb4g7C4EL/inside-the-mind-of-a-superhuman-go-model-how-does-leela-zero-2 (visited on 08/14/2024).

[17] Matěj Fanta. "Rules extraction from deep neural networks Master thesis". In: *ResearchGate* (2019). URL: https://www.researchgate.net/publication/335146775_Rules_extraction_from_deep_neural_networks_Master_thesis.

[18] Michael S. Floater. *Lecture 13: Non-linear least squares and the Gauss-Newton method*. 2018. URL: https://www.uio.no/studier/emner/matnat/math/MAT3110/h19/undervisningsmateriale/lecture13.pdf (visited on 08/14/2024).

[19] C Daniel Freeman and Joan Bruna. "Topology and Geometry of Half-Rectified Network Optimization". In: *arXiv* (2016). URL: https://arxiv.org/pdf/1611.01540.

[20] Guillaume Garrigos and Robert M Gower. "Handbook of Convergence Theorems for (Stochastic) Gradient Methods". In: *arXiv* (2023). URL: https://arxiv.org/pdf/2301.11235.

[21] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. "Qualitatively Characterizing Neural Network Optimization Problems". In: *arXiv* (2014). URL: https://arxiv.org/pdf/1412.6544.

[22] Sepp Hochreiter and Jürgen Schmidhuber. "Flat Minima". In: *Neural computation* 9.1 (1997), pp. 1–42. URL: https://www.researchgate.net/publication/14100213_Flat_Minima.

[23] *Inception v4*. 2021. URL: https://paperswithcode.com/model/inception-v4?variant=inception-v4-1 (visited on 08/14/2024).

[24] Ziwei Ji and Matus Telgarsky. "Gradient descent aligns the layers of deep linear networks". In: *arXiv* (2018). URL: https://arxiv.org/abs/1810.02032.

[25] Kenji Kawaguchi. "Deep Learning without Poor Local Minima". In: *Advances in neural information processing systems* 29 (2016). URL: https://arxiv.org/pdf/1605.07110.

[26] Nitish Shirish Keskar et al. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima". In: *arXiv* (2016). URL: http://arxiv.org/abs/1609.04836.

[27] Thomas Laurent and James Brecht. "Deep Linear Networks with Arbitrary Loss: All Local Minima Are Global". In: *International conference on machine learning* (2018), pp. 2902–2907. URL: https://arxiv.org/pdf/1712.01473.

[28] Thomas Laurent and James Brecht. "The Multilinear Structure of ReLU Networks". In: *International conference on machine learning* (2018), pp. 2908–2916. URL: https://arxiv.org/pdf/1712.10132.

[29] *Leela Chess Zero Wiki*. 2020. URL: https://lczero.org/dev/wiki/leela-chess-zero-wiki/ (visited on 08/14/2024).

[30] Dawei Li, Tian Ding, and Ruoyu Sun. "On the Benefit of Width for Neural Networks: Disappearance of Basins". In: *SIAM Journal on Optimization* 32.3 (2022), pp. 1728–1758. URL: https://arxiv.org/pdf/1812.11039.

[31] Hao Li et al. "Visualizing the Loss Landscape of Neural Nets". In: *Advances in neural information processing systems* 31 (2018). URL: https://arxiv.org/pdf/1712.09913.

[32] Chaoyue Liu, Libin Zhu, and Mikhail Belkin. "Loss landscapes and optimization in over-parameterized non-linear systems and neural networks". In: *Applied and Computational Harmonic Analysis* 59 (2022), pp. 85–116. URL: https://arxiv.org/abs/2003.00307.

[33] Haihao Lu and Kenji Kawaguchi. "Depth Creates No Bad Local Minima". In: *arXiv* (2017). URL: https://arxiv.org/pdf/1702.08580.

[34] Hartmut Maennel, Olivier Bousquet, and Sylvain Gelly. "Gradient Descent Quantizes ReLU Network Features". In: *arXiv* (2018). URL: https://arxiv.org/abs/1803.08367.

[35] Yurii Nesterov et al. *Lectures on convex optimization*. Vol. 137. Springer, 2018.

[36] Maher Nouiehed and Meisam Razaviyayn. "Learning Deep Models: Critical Points and Local Openness". In: *INFORMS Journal on Optimization* 4.2 (2022), pp. 148–173. URL: https://arxiv.org/pdf/1803.02968.

[37] Samet Oymak and Mahdi Soltanolkotabi. "Overparameterized Nonlinear Learning: Gradient Descent Takes the Shortest Path?" In: *International Conference on Machine Learning* (2019), pp. 4951–4960. URL: https://proceedings.mlr.press/v97/oymak19a/oymak19a.pdf.

[38] Ruo-Yu Sun. "Optimization for Deep Learning: An Overview". In: *Journal of the Operations Research Society of China* 8.2 (2020), pp. 249–294. URL: https://link.springer.com/article/10.1007/s40305-020-00309-6.

[39] *tesla.com*. URL: https://www.tesla.com/AI (visited on 08/14/2024).

[40] Santosh Vempala and John Wilmes. "Gradient Descent for One-Hidden-Layer Neural Networks: Polynomial Convergence and SQ Lower Bounds". In: *Conference on Learning Theory* (2019), pp. 3115–3117. URL: https://arxiv.org/pdf/1805.02677.

[41] Luca Venturi, Afonso S Bandeira, and Joan Bruna. "Spurious Valleys in One-hidden-layer Neural Network Optimization Landscapes". In: *Journal of Machine Learning Research* 20.133 (2019), pp. 1–34. URL: https://arxiv.org/pdf/1802.06384.

[42] Aladin Virmaux and Kevin Scaman. "Lipschitz regularity of deep neural networks: analysis and efficient estimation". In: *Advances in Neural Information Processing Systems* 31 (2018). URL: https://arxiv.org/abs/1805.10965.

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**