Deep Learning as Dynamical Systems

Universidad Autónoma de Madrid

Sergi Andreu

Departament of Mathematics

Faculty of Sciences - UAM

Master Thesis

Master in Mathematics and Applications

2019-2020

Supervised by

Enrique Zuazua

Acknowledgments

Agradecer a Enrique Zuazua la oportunidad de desarrollar este trabajo, su guía y consejos, tanto en lo referente a la memoria como a la investigación.
A los profesores del máster, por ayudarme a entender mi relación con las Matemáticas, por su dedicación y por enseñarme que hay mejores maneras de ser docente, lejos de ciertas experiencias de antaño.
Y, sobre todo, a mis amigos y familiares, por estar ahí en esta época difícil y plagada de teletrabajo; por respetar mi mal humor, irascibilidad y pocas ganas de hacer ocio.

This work has been done under a fellowship with the ERC DyCon Project, at the Chair of Computational Mathematics at University of Deusto.

Abstract

Several attemps have been tried to provide a rigorous and fundamental understanding of deep learning. A unified framework that explains the reason why deep learning has been so succesful in practice remains limited. In this work, the dynamical systems approach of deep learning is presented, and under this approach some results are introduced and aligned that help understand the effectivity of deep learning under from a mathematical perspective.

Neural networks are seen as discrete-time dynamical systems, where the weights or the networks are recasted as controllers. ResNets are introduced and seen as discretization of ODEs, providing a robust framework from which it is possible to see Deep Learning as a mean field optimal control problem.

This approach have been based in recent work, and some of the key aspects remain unkown. The idea is to bring the reader a wide idea of the problem, the formulation and the mathematics needed to be developed so as to find a theoretical framework in which to study deep learning formally, and to propose new architectures and algorithms.

Contents

List of Figures

Part I

Introduction

Chapter 1

Motivation

Deep Learning is aimed at solving the problem of supervised learning.

That is, given a dataset $\{x^i, y^i\}_{i=1}^N$, with $x \in \mathcal{X}$, $y \in \mathcal{Y}$; and an oracle function $\mathcal{F} : \mathcal{X} \to \mathcal{Y}$, $y^i = \mathcal{F}(x^i)$, the idea is to learn or approximate the function $\mathcal{F}$, by using the information from the dataset.
The function $\mathcal{F}$ is approximated with functions $\hat{\mathcal{F}}$ from an hypothesis space $\mathcal{H}$.

In classical supervised learning, the candidate functions are of the type $\sum_j a_j \varphi_j(x) \ : \ a_j \in \mathbb{R}$, where the functions φ_j are chosen specifically for each problem. For example, in one-dimensional polynomial regression, they are given by $\varphi_j(x) = x^j$.

That is, the hypothesis space is given by

$$\mathcal{H}^{reg} = \left\{ \sum_{j=1}^{J} a_j \varphi_j(x) \ : \ a_j \in \mathbb{R} \right\} \tag{1.1}$$

The problem with these hypothesis spaces is that they are not dense in the space of continuous functions unless $J \to \infty$, the functions $\{\varphi_j\}$ usually need to be chosen case-by-case, the computational complexity needed so as to find the weights $\{a_j\}$ depends on the functions $\{\varphi_j\}$, and this problem does not scale well when the dimension of x is high.[1] This is called the *curse of dimensionality.*

[1]A more detailed description of these problems can be found in the appendix "supervised learning".

Therefore, in high-dimensional problems or in settings in which the function $\mathcal{F}$ is unkown and no properties can be deduced a priori, $\mathcal{H}^{reg}$ is not a good hypothesis space.

Shallow neural networks are then introduced, such that

$$\mathcal{H}^{\text{shallow}} = \left\{ \sum_j a_j \sigma(\langle w_j, x\rangle + b_j) \,:\, w_j \in \mathcal{X}^* \,,\, b_j \in \mathbb{R} \right\}$$

where $\mathcal{X}^*$ is the dual space of $\mathcal{X}$, that needs to be introduced such that $\langle w_j, x\rangle \in \mathbb{R}$, where the standard matrix product $\langle , \rangle$ is considered; and σ is the activation function (usually ReLu or tanh).
This new hypothesis space has a very similar formalism as the previous ones, but in this case the basis functions $\varphi_j = \sigma(\langle w_j, x\rangle + b_j)$ are not explicit.

A shallow network essentially consist on an affine transformation (given by $(\langle w_j, x\rangle + b_j)$) and a nonlinearity (given by σ).

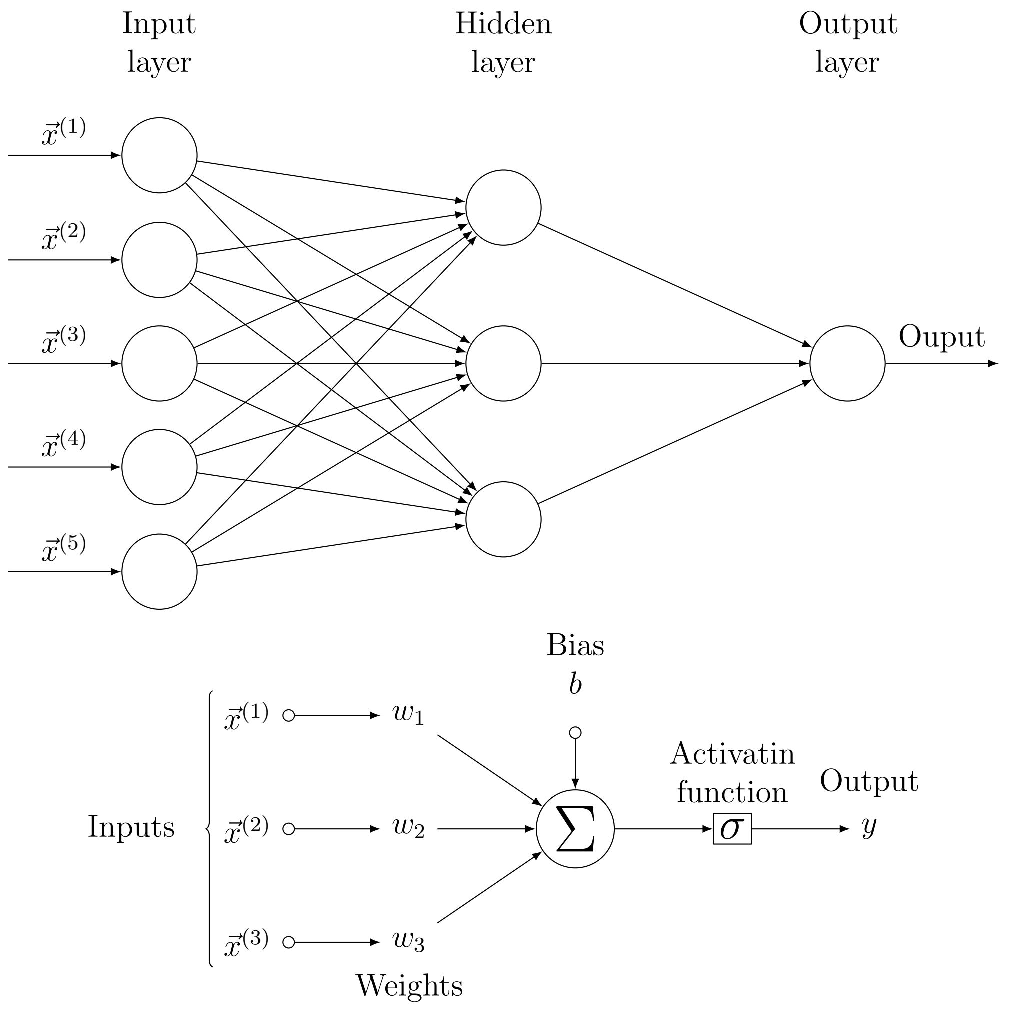


Figure 1.1: Scheme of a shallow neural network with one hidden layer and input dimension 3

So as to get more robust or rich hypothesis spaces, the idea has been to stack shallow neural networks, getting an hypothesis space

$$\mathcal{H}^{\text{deep}} = \{F_L \circ F_{L-1} \circ ... \circ F_0(x) \ : \ \ F_l \text{ is a shallow network}\} \tag{1.2}$$

Once the hypothesis space is fixed, the idea is to find the weights (in the case of deep learning, $W_j^{(l)}, b_j^{(l)}$ for each function F_l) that define $\hat{F} \in \mathcal{H}$, so as to minimize the empirical risk

$$\hat{F} = \underset{F \in \mathcal{H}}{\arg\min} \frac{1}{N} \sum_{i=1}^{N} \|F(x^i) - y^i\|_{\mathcal{L}} \tag{1.3}$$

where $y^i = \mathcal{F}(x^i)$, and $\|\cdot\|_{\mathcal{L}}$ is a distance defined by a loss function $\mathcal{L}$.

The problem of empirical risk minimization is an approximation of the population risk minimization problem,

$$\tilde{F} = \underset{F \in \mathcal{H}}{\arg\min}\, \mathbb{E}_{(x,y)\sim\mu^*} \|F(x) - y\|_{\mathcal{L}} \tag{1.4}$$

.

It is expected that, given enough training samples, the solution to the problem of empirical minimization would be really close to the solution of the population risk minimization.

There are three main paradigms associated with supervised learning.

- Approximation: Is the hypothesis space dense in the space of functions that we want to approximate?
- Optimization: Can we find the weights of $\hat{F}$ such that we solve the empirical risk minimization problem? How?
- Generalisation: How does the empirical risk minimization problem relates to the population risk minimization problem? Do we have enough samples?

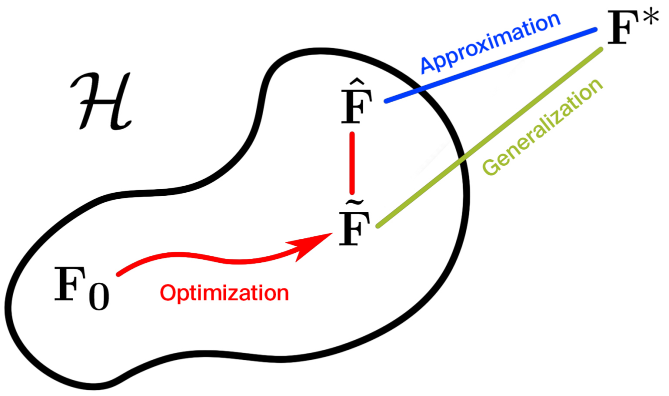


Figure 1.2: Diagram of the main paradigms of supervised learning.

In deep learning theory, the problem of approximation is related to the Universal Approximation Theorems, that state that even shallow neural networks are rich in the space

of bounded continuous functions, given a sufficient number of neurons.
The problem of optimization is related to the algorithms used so as to update the weights, that are widely based in gradient descent and backpropagation. In practice, the weights are initialized at random, and how exactly to initialize those weights is known as the problem of initialization, that remains unsettled.
The problem of generalisation is perhaps the most heuristical one. There is not yet a mathematical framework in which to treat this problem easily and in a general case.

By introducing deep neural networks ("stacked" shallow neural networks) we do not have an explicit expression for the functions $\hat{F}$, since the role of composition is complex.

That is, given a point $x \in \mathcal{X}$, it is not direct to find $\hat{F}(x)$, where

$$\hat{F}(x) = F_L \circ F_{L-1} \circ ... \circ F_0(x) \tag{1.5}$$

That is, there is not a compact analytical expression for $\hat{F}(x)$, but rather it has to be computed iteratively.

This problem relates to the problem of finding the position in space of a point as propagated by a dicrete map in dynamical systems.
Indeed, if the functions $F_l = F \;\; \forall l = \{0, ..., L\}$ was chosen to be

$$F(x) = rx(1-x)$$

Then the problem would be related to solving the logistic map, which is a well-studied discrete dynamical system that is known to generate a very complex dynamics.
The problem of deep learning is more difficult to solve, since in practice the functions F_l depend on some parameters (weights), and so the formula of the discrete map changes at each timestep l.

The dynamical systems perspective of deep learning allows us to treat the problem of composition easily, and use previous results in analysis.
The problem of approximation would be related to how complex is the dynamics of the system, the problem of control could be reformulated as an optimal control problem and

a mean-field optimal control approach can give some insight into the problem of generalisation.

A recent architecture used in deep learning is called ResNet.
In this case, instead of composing simple functions F_l (given by a shallow neural network, or by a deep neural network) the idea is to sum them; the represented function by a ResNet is given by z_L, with

$$z_{l+1} = z_l + F_l \text{ for } 0 \leq l \leq L-1 \quad z_0 = x_0 \tag{1.6}$$

In this class of architectures, the discrete maps of the hypothesis space can be recasted as a discretiztion of a flow-map from dynamical systems, i.e.

$$\mathcal{H}^{\text{ResNet}} \approx \mathcal{H}^{\text{ODE}} = \{z \mapsto g(x_L) \,:\, \dot{x}_t = f(x_l, \theta_l),\, x_0 = z,\, \theta_l \in \Theta,\, l \in [0, L]\} \tag{1.7}$$

where $g : \mathcal{X} \to Y$ is a terminal activation function, that matches the output space with the input space. For example, in the case of binary classification, the function g would split the propagated space of $\mathcal{X}$ in two.

The dynamical system perspective is now more appropiate for finding useful results when dealing with ResNets.

Chapter 2

Overview

The scope of this work is not to provide a solid and closed theory of Deep Learning as Dynamical Systems, but to explain the essentials of this approach.

The notation and exposition would be particularly interesting for people with background in Deep Learning. In practice, Deep Learning is considered to be blackbox, in the sense that both the input and outputs are known, but the model (the neural network) but there is not knowledge about its internal working.

The approach presented in here is radically opposed. The idea is to understand the intermediate activations of the network as propagation of the inputs through a dynamical system. This is very interesting in terms of explainability of the models, so as to understand better the convergence of algorithms and to show results regarding architecture selection.

However, this interpretation and recent work seems not to be mostly focused on explaining why Deep Learning has succeded. Instead, new algorithms, architectures and modifications are done to as that the neural networks are easily explained in terms of Optimal Control, but the assumptions made such as the theory works are usually not satisfied in practice.
The idea is to build a general theoretical frame such that no strong assumptions are needed, and such that the ideas apply to a wide variety of networks and training algorithms.

The theory presented is very general, and is not focused on finding a very specific kind of results. The idea is to present to the reader a solid background of this interpretation, and also to show the weaknesses.

Many questions would remain open in most of the directions. For simplicity reasons (and also because I don't know the answers) they are not presented and are out of the scope of this work.

In chapter $3-5$ the problem of Deep Learning is introduced directly as a specific problem of optimal control.
This is perhaps misleading, since it makes the problem of relating Deep Learning with mathematics trivial.

Indeed, the backpropagation algorithms were historically deduced more heuristically.
The lagrangian formalism is a particularly interesting way of deducing and pose the problem, due to its intrinsic relation with optimal control.

In chapter 6, results of approximation are shown. The main part of the results are focused on shallow or multilayer perceptrons, since the ResNets are based on this architectures, and so proving results on the approximation with perceptrons implies those results with ResNets.

In chapter 7, each layer of the network is seen as a coordinate transformation (this is an abuse of notation, since the functions may not be homeomorphisms).

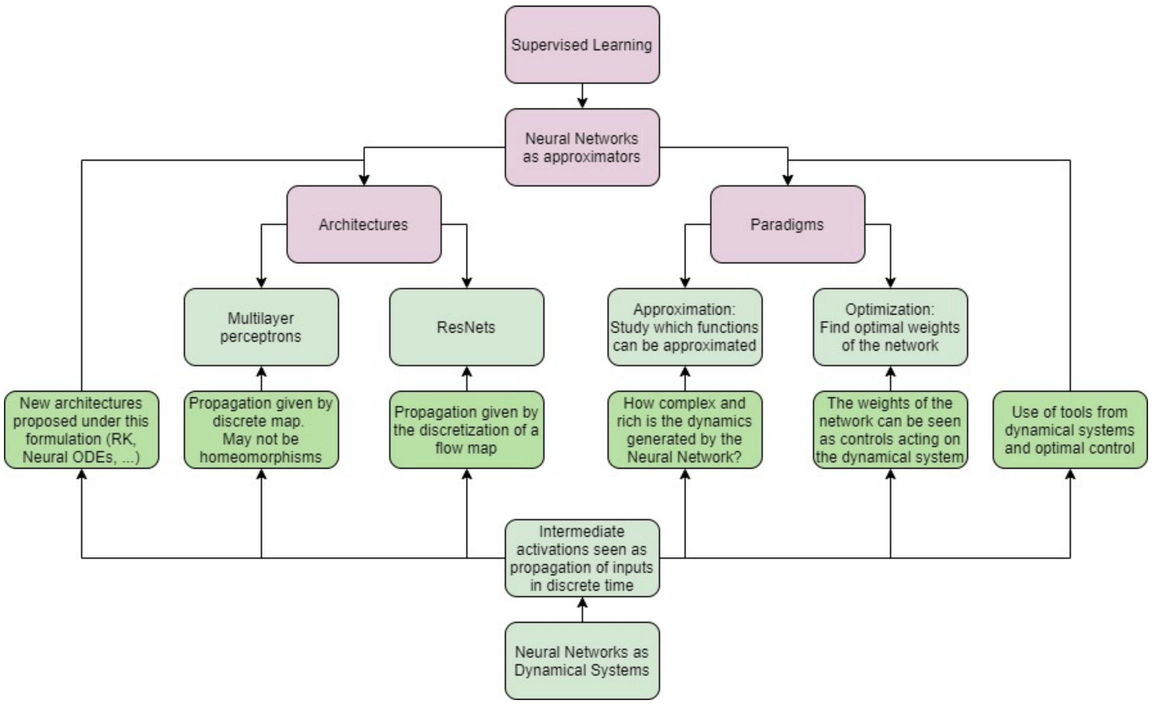
In related work, some ideas are introduced that help understand the theory of Deep Learning as Dynamical Systems.

In chapter 6, the relation between multilayer perceptrons and ResNets is presented.
This aspects are usually omitted, but play a very important role in understanding the theory and ResNets are presented as the only valid architecture. The choice of ResNets over multilayer perceptrons is explained in the deep learning limit.

In chapter 7, the idea of Deep Learning as Optimal Control is fostered with the use of mean-field games.
The technical difficulties of this are avoided. However, the introduction of mean-field

games allows for obtaining a priori generalisation estimates.
In practice, those estimates are very difficult to handle, and the assumptions made are not met in many applications.

In chapter 7, a specific problem is presented, such that the previous theory can be used, specifically in the architecture selection. Some of the problems and weaknesses of the theory are presented, as well as some of the open problems that need to be tackled.


Supervised Learning
Neural Networks as approximators
Architectures
Paradigms
Multilayer perceptrons
ResNets
Approximation: Study which functions can be approximated
Optimization: Find optimal weights of the network
New architectures proposed under this formulation (RK, Neural ODEs, ...)
Propagation given by discrete map. May not be homeomorphisms
Propagation given by the discretization of a flow map
How complex and rich is the dynamics generated by the Neural Network?
The weights of the network can be seen as controls acting on the dynamical system
Use of tools from dynamical systems and optimal control
Intermediate activations seen as propagation of inputs in discrete time
Neural Networks as Dynamical Systems
Figure 2.1: Diagram between the interrelation of Deep Learning as Dynamical Systems

Part II

Main results

Chapter 3

Neural Networks as discrete maps

Deep learning is a type of supervised learning, We want to infer a function

$$F : \mathcal{X} \to \mathcal{Y} \tag{3.1}$$

by using training data

$$\{(x_i, y_i)\}_{i=1}^{n} \tag{3.2}$$

Such that $x_i \in \mathcal{X}$, $y_i \in \mathcal{Y}$ for all $i = \{1, ..., n\}$, and the mapping F is expected to connect each x_i with y_i such that $F(x_i) = y_i$, and generalise to unseen data.

Deep learning assumes that the function F can be approximated by the giant composition of simple functions $\{f_i\}_{i=1}^{L}$ such that each f_i is given by an affine transformation and a nonlinearity, depending each f_i on some parameters θ_i.
The collection of the θ_i would be denoted by θ.

Using a multilayer perceptron, the function F would be then approximated by

$$\hat{F} : \mathcal{X} \to \mathcal{Y} \tag{3.3}$$

$$\hat{F}(\cdot\,;\theta) = f_L \circ f_{L-1} \circ ... \circ f_1(\cdot) \tag{3.4}$$

in the sense that $\hat{F}$ is close to F with respect to a distance defined by a loss function.

The problem can be stated as finding θ such that

$$\min_{\theta_1,\dots,\theta_L} \sum_{i=1}^{N} \text{loss}\left(\hat{F}(x_i\,;\theta), y_i\right) \tag{3.5}$$

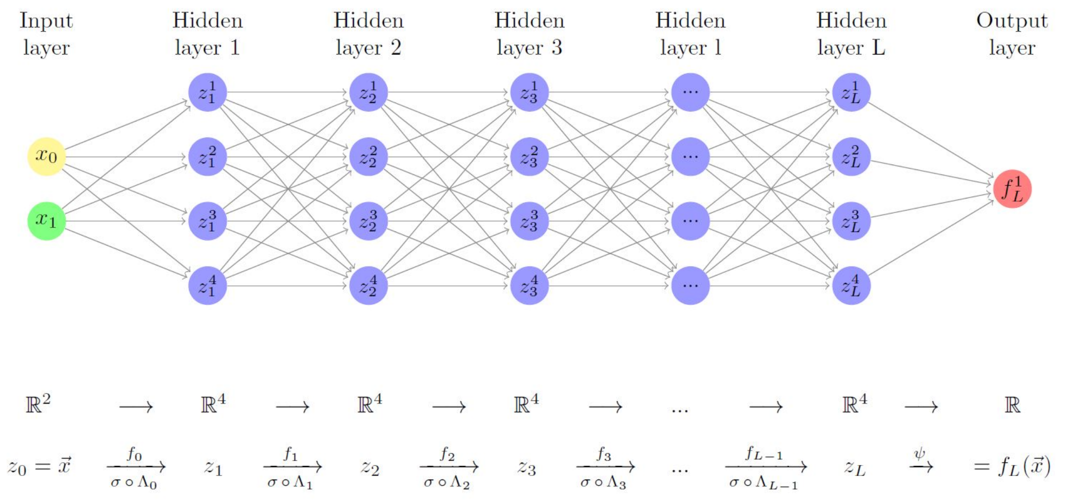
constrained by

$$\hat{F}(x_k) = z_k^{(L)} \tag{3.6}$$

where

$$\begin{cases} z_k^{(L)} = f_L(z_k^{(L-1)}, \theta_L) \\ z_k^{(L-1)} = f_{L-1}(z_k^{(L-2)}, \theta_{L-1}) \\ \dots \\ z_k^{(1)} = f_1(x_k, \theta_1) \\ z_0 = x_k \end{cases} \tag{3.7}$$

where $f_l = \sigma \circ \Lambda_l$, σ a nonlinearity (tanh, ReLU) and Λ_l an affine transformation.



Input layer
Hidden layer 1
Hidden layer 2
Hidden layer 3
Hidden layer l
Hidden layer L
Output layer
x_0
x_1
z_1^1 z_1^2 z_1^3 z_1^4
z_2^1 z_2^2 z_2^3 z_2^4
z_3^1 z_3^2 z_3^3 z_3^4
...
z_L^1 z_L^2 z_L^3 z_L^4
f_L^1
$\mathbb{R}^2 \longrightarrow \mathbb{R}^4 \longrightarrow \mathbb{R}^4 \longrightarrow \mathbb{R}^4 \longrightarrow \dots \longrightarrow \mathbb{R}^4 \longrightarrow \mathbb{R}$
$z_0 = \vec{x} \xrightarrow[\sigma\circ\Lambda_0]{f_0} z_1 \xrightarrow[\sigma\circ\Lambda_1]{f_1} z_2 \xrightarrow[\sigma\circ\Lambda_2]{f_2} z_3 \xrightarrow[\sigma\circ\Lambda_3]{f_3} \dots \xrightarrow[\sigma\circ\Lambda_{L-1}]{f_{L-1}} z_L \xrightarrow{\psi} = f_L(\vec{x})$

Figure 3.1: Scheme of a multilayer perceptron with dimension of the input 2, output dimension 1, a constant number of components 4 and a feedforward function given by f a shallow neural network.

The key point is to see the intermediate activations $z_k^{(l)} = f^{(l)}(f^{(l-1)}(\dots(x_k)))$ as the propagation of x_k by a discrete map $\Phi(x_k, l)$, such that

$$\begin{aligned} \Phi : \mathcal{X} \times \{1, \dots, L\} &\to \mathcal{Z} \\ x_k \mapsto \Phi(x_k, l) &= z_k^{(l)} \end{aligned} \tag{3.8}$$

This is particularly interesting when the network is already trained. That is, when the weights have been iterativaly updated by a backpropagation algorithm. The weights θ of the neural network are initialized following an arbitrary distribution.

Indeed, a more complex dynamics could be studied using the map $\tilde{\Phi}$ such that

$$\begin{aligned} \tilde{\Phi} : \mathcal{X} \times \{1, ..., L\} \times \{0, ..., T\} &\to \mathcal{Z} \\ x_k &\mapsto \tilde{\Phi}(x_k, l, \tau) = z_k^{(l)}(\theta_\tau) \end{aligned} \tag{3.9}$$

where in this case the notation expresses explicitly the fact that $z_k^{(l)}$ depends on θ, and θ_τ is an element of the succession $\{\theta_0, ..., \theta_T$ where θ_0 is initialized at random and the succesive terms θ_τ are obtained using a backpropagation gradient descent algorithm.

Indeed, the map Φ is the relevant think to study in this setting, since the interesting part of the dynamical systems approach is to study deep learning as opposed to learning with shallow networks by understanding the index of the layer as a discrete timestep.

It can be understood that, if the algorithm finds some optimal weights, then $\Phi(x_k, l) \approx \tilde{\Phi}(x_k, l, \tau \to \infty)$. Indeed this is not needed, since the results presented work for any given training time τ, although some interesting results may only fully apply for optimal weights, i.e. $\tau \to \infty$ in the ideal case.

Although the backpropagation algorithm is deduced heuristically as an extension of the classical gradient descent, it is interesting to deduce it from a Lagrangian formalism as inspired by optimal control theory [36].

Essentially, the backpropagation algorithm uses a simple gradient descent and combines is with the chain rule so as to derive how the weights of each neuron should be updated.

In optimal control, the question addressed are related to how the state of a system at time T would be modified by a change in the control of the variables at $T - t$. In a discretized version, the problem of finding the weights θ at an intermediate layer can be replaced by studying how the output of the network will be modified by changind a weight in a layer $K - k$, and therefore seeing each layer as a discretized time index.

Under this approach, a lagrangian can be defined as

$$\mathcal{L}(x,p,\theta) = \text{loss}(z^{(L)}, y) - \sum_{l=1}^{L} p_l^T(z^{(l)} - f_l(z^{(l-1)}); \theta_l)) \tag{3.10}$$

where $z^{(L)}$ is given by equation 6.2, p are lagrange multipliers. $n = 1$ is considered so as to drop the dependance on k and simplify the notation.

The first term is just the cost function associated to the error wanted to minimize, and the second term is given by the constraints.

From variational calculus, it is known that $\nabla\mathcal{L}(x,p,\theta) = 0$ is a necessary condition which defines the local minimum of the performance function while meeting the constraints.

This can be split into three subconditions

$$\frac{\partial\mathcal{L}(x,p,\theta)}{\partial x} = \nabla_x\mathcal{L}(x,p,\theta) = 0 \tag{3.11}$$

$$\frac{\partial\mathcal{L}(x,p,\theta)}{\partial p} = \nabla_p\mathcal{L}(x,p,\theta) = 0 \tag{3.12}$$

$$\frac{\partial\mathcal{L}(x,p,\theta)}{\partial \theta} = \nabla_\theta\mathcal{L}(x,p,\theta) = 0 \tag{3.13}$$

The associated gradients of the lagrangian would be given as

$$\nabla_{p_l}\mathcal{L} = z^{(l)} - f_l(z^{(l-1)}; \theta_l) \tag{3.14}$$

Setting $\nabla_{p_l}\mathcal{L} = 0$ gives the so called *forward pass*, that is the pass defining the intermediate activations $z^{(l)}$. This would give the expression of a discrete map $\Phi : \mathcal{X} \times \{0, ..., L\} \to \mathcal{Z}$, with $\Phi(\cdot, l) = z^{(l)}(\cdot)$, being l the index layer, interpreted as a discrete timestep.

$$\nabla_{z^{(L)}}\mathcal{L} = -p_L + \nabla_{z^{(L)}}\text{loss}(z^{(L)}, y) \tag{3.15}$$

$\nabla_{z^{(L)}}\mathcal{L} = 0$ gets the terminal condition of p, $p_L = \nabla_{z^{(L)}}\text{loss}(z^{(L)}, y)$

$$\nabla_{z^{(l)}}\mathcal{L} = -p_l + \nabla_{z^{(l)}} f_{l+1}(z^{(l)}; \theta_{l+1}) \text{ for } l = 1, ..., L-1 \tag{3.16}$$

Using this expression and the terminal condition of p, we obtain the *backward pass.*

$$\nabla_{\theta_l}\mathcal{L} = \nabla_{\theta_l} f_l(z^{(l-1)}, \theta_l)^T p_l \tag{3.17}$$

using the above expressions, the gradient of the loss with respect to θ is computed, and therefore θ can be updated using gradient descent.
This is done using automatic differenciation, using a technique called reverse mode accumulation [4].
Essentially, the computation of the gradients for each neuron is stored in the memory, and so the memory cost of training a neural network scales with the number of neurons in the network.
The implementation of automatic differenciation in ML is not trivial from a mathematical viewpoint, but it is easy and cheap to implement it computationally. It can be understood as if partial derivatives were actually computed if no numerical errors were considered.

θ is initialized using some protocol. In general it is done by using an i.i.d. variable, but it depends on the specific implementation. The problem of initialization is not yet fully understood, and it has indeed a huge impact on performance [27].
This is reasonable, since if some optimal weights θ^* existed, then initializing $\theta_0 = \theta^*$ would solve the problem.

This is not trivial, and since it is not a compact analytical expression for the function represented by neural networks, it is not always feasible to find θ_0 to be θ^* or even close, but rather θ_0 must follow a distribution given by a random vector, whose variance and mean can be modulated for each architecture, and is usually chosen to be given by a uniform random variable between -1 and 1, or using more sophisticated initialization algorithms [32] [34] [19].

In order to reduce the memory cost, the backpropagation algorithms take into account only a few samples of the training data, rather than the whole distribution; this is usually referred as stochastic gradient descent.
A batch of the training data is chosen at each training step

$$\{(x_i, y_i)\}_{i=1}^m \tag{3.18}$$

with $m \leq n$, and where the training data has been shuffled. Due to memory constraints, it is usual that $m << n$.

The *forward pass* it then computed, using the previous expression, and obtaining an expression of the intermediate and final activations. Then, using the *backward pass*, the gradient $\nabla_\theta \mathcal{L}^m$ can be computed.

Then, θ is updated to θ' such that

$$\theta' = \theta - \nu \nabla_\theta \mathcal{L}^m(\theta) \tag{3.19}$$

and this process is repeated iteratively. Each time step is called *epoch*. The intention is that, after a lot of *epochs*, θ is such that $\hat{F}(\cdot; \theta)$ is close to the desired function F.

In this general case, it is not possible to infer properties or characterize the discrete map Φ, as the problem as stated before is super-general.
There are many different neural architectures that have been shown effective in different settings. An architecture means the specification of $\{f_i\}_{l=1}^L$, for general weights; i.e. in multilayer perceptrons the architecture would be fully characterized by knowing the input and output dimensions, the activation functions σ, the number of components C (neurons per layer) and the number of layers L.

The most simple architectures are the ones already shown, the multilayer perceptrons, in which $f_l = \sigma_l \circ \Lambda_l$, being σ_l an activation function (tanh, sigmoid, ReLU, ...) and Λ_l an affine transformation, and considering that the dimension of f_l is the same for each l.

This architectures are useful, but they have some associated problems when going to the limit $L >> 1$ (the deep learning regime). Moreover, the discrete map Φ cannot be seen as the discretization of a continuous map, and Φ may not correspond to an homeomorphism.

Regardless of that, to study the discrete map Φ is an interesting exercise, in the sense that characterizing the complexity of Φ is useful for understanding the space of functions

represented by multilayer perceptrons (and thus which functions can multilayer perceptrons approximate) as inspired by, for example, the logistic map.
This allow for a more comprehensive interpretation of fundamental results of deep learning.

Although some improvements in algorithms and initialization can be done to multilayer perceptrons, a new family of architectures, called residual networks, is introduced, that is particularly aimed to give good results at the deep learning limit.

Residual networks, also called ResNets, represent functions $\hat{R}(\cdot)$, such that

$$\hat{R}(\cdot) = h_T(\cdot) \tag{3.20}$$

with

$$\begin{cases} h_T = h_{T-1} + f_t(h_{T-1};\theta_{T-1}) \\ ... \\ h_{t+1} = h_t + f_t(h_t;\theta_t) \\ ... \\ h_0 = x \end{cases} \tag{3.21}$$

where the notation is such that $h_t \equiv h_t(x)$.

The associated discrete map $\Phi(x,t) = h_t(x)$, defined for $t = 0, ..., T$ can be in this case seen as a discretization of a continuous map.

This can be seen by rewriting the fundamental equation of ResNets as

$$h_{t+1} = h_t + \epsilon f_t(h_t;\theta_t)$$

where ϵ can be easily introduced by absorbing it into $f_t()$. Experimentally, ϵ is observed to be small [16] [60]; if not, it is easy to introduce ϵ in the implementation of ResNets.

Supposing a variable $h(t)$

$$\begin{cases} \dot{h}(t) = f(h(t); \theta(t)) \\ h(0) = x \end{cases} \tag{3.22}$$

then the fundental equation of ResNets can be seen as a discretization of this variable h using an euler method.

Then, the discrete map associated to the forward pass in resnets, Φ, can be seen as the discretization of a continuous map.
This is beneficial in terms of reinterpreting Deep Learning in a more sophisticated mathematical framework.

The viewpoint of studying Deep Learning as Dynamical Systems makes more sense in ResNets, since it is related to the dynamics and control of an ODE.

Some approximation and optimization results of Deep Learning can be easily infered from classical ODE theory.
However, the understanding of multilayer perceptrons is limited and unsettled, and some results from ResNets cannot be applied in multilayer perceptrons.

The main theory of Deep Learning as Dynamical Systems presented in here refers mostly to ResNets, but some effort has been done so as to include also results with multilayer perceptrons, and to discern and analyse the relation between ResNets and multilayer perceptrons.
Indeed, the block functions f of a ResNet can be given by multilayer perceptrons.

Chapter 4

Control theory of Neural Networks

Consider the system defined by

$$\dot{x}_t^i = f(t, x_t^i, \theta_t), \quad x_0^i = x, \quad 0 \leq t \leq T \tag{4.1}$$

where the index i represents the index of the datapoint, and the index t the index of the layer in the continuous-time idealization. This system is equivalent to that defined by a ResNet, under the assumption that the residual block $x_{l+1} - x_l$ in ResNets tends in norm to 0 as $L \to \infty$.

Then, θ_t can be seen as a control variable.

The space of admissible controls is

$$\mathcal{U} = \{\theta[0, T] \to \Theta \mid \theta \text{ is Lebesgue measurable}\} \tag{4.2}$$

where Θ is usually $\mathbb{R}$, but this interpretation can be done in a more general case.

A final function $g(\cdot)$ is applied in top of h_T so as to proceed with the learning. For example, in the case of binary classification, the aim is to make that the ResNets make the two classes linearly separable, which constitutes a weak problem of optimal transport[1]. This "last function" maps a part of the space to one class, and the other to the other class, i.e. it may be defined by an hyperplane.

[1]The problem is weak in the sense that there is not a final optimal distribution, but rather a family of them. For example, if two distributions are linearly separable in some space, any rotation or affine transformation would make them also linearly separable

Formally, and in general, we want to compare our desired solution $F(\cdot)$ to $g(\cdot)$. The function g is chosen such that

$$g : \mathcal{Z} \to \mathcal{Y} \tag{4.3}$$

and it is sometimes referred to as the terminal activation function.
If the terminal activation function is not specified, it is implicit that the identity function is considered.

The aim is to find the solution to

$$\min_{\theta \in \mathcal{U}} \sum_{i=1}^{n} \text{loss}\,(g(x'_T), y') + \int_0^T L(\theta(t))dt$$

subject to

$$\dot{x}_t^i = f(t, x_t^i, \theta_t), \quad x_0^i = x, \quad 0 \le t \le T \tag{4.4}$$

where $L(\theta)$ is a regularization term. In the settings previously introduced, $L = 0$, but this more general case is considered.
This regularization term is added such that the Pontryagin Maximum Principle give us valuable information about the control functional, and can be seen as the running cost.

A hamiltonian can be defined

$$H[0, T] \times \mathbb{R}^d \times \mathbb{R}^d \times \mathcal{U} \to \mathbb{R} \tag{4.5}$$

such that

$$H(t, x, p, \theta) = p \cdot f(t, x; \theta) - L(\theta) \tag{4.6}$$

At optimiality, it is known that

$$\dot{x_t}^* = \nabla_p H(t, x_T^*, p_T^*, \theta_T^*) \tag{4.7}$$

this can be easily seen as a continuous time idealization of the forward pass.

$$\dot{p_t} = -\nabla_x H(t, x_t^*, p_t^*, \theta_t^*),\ p_T^* = -\nabla \text{loss}\,(g(x_T^*), y) \tag{4.8}$$

which corresponds to the continuous time idealization of the backward pass.

The Pontryagin's Maximum Principle (PMP) states that

$$H(t, x_T^*, p_T^*, \theta_T^*) \geq H(t, x_T^*, p_T^*, \theta)\ \ \forall\theta, \forall t \tag{4.9}$$

These equations (4.7, 4.8, 4.9) gives us first order optimality necessary conditions for our problem.
This allows to use better methods for finding θ, that are more robust, sophisticated and understood than gradient descent methods.

The PMP therefore suggest a new training algorithm:

PMP - based algorithm for finding θ in ResNets:

- Initialise $\theta^k \in \mathcal{U}$ for $0 \leq k \leq K$, with K an arbitrary number of iterations.
- Run the forward pass so as to obtain the x^k trajectory.
- Run the backward pass to obtain the p^k trajectory.
- Update θ such that

$$\theta^{k+1} = arg\max_{\theta} H(t, x^k, p^k, \theta) \tag{4.10}$$

This method is known as the *method of succesive approximations* (MSA) [28].

The question that now arises is whether it is possible to recover gradient descent from equation 4.10.

4.1 PMP and gradient descent

In the MSA, the last step can be replaced by a discrete form such that

$$\theta_n^{k+1} = \theta_n^k + \nu \nabla_\theta H_n(x_n^{\theta^k}, p_{n+1}^{\theta^k}, \theta_n^k) \tag{4.11}$$

where n is the time index, $0 \leq n < N$, imposed by the discretization of the variables.

the equation 4.11 can be recasted as

$$\theta_n^{k+1} = \theta_n^k - \nu \nabla_\theta J(\theta) \tag{4.12}$$

where

$$J(\theta) = \operatorname{loss}(g(x_n), y) + \delta \sum_{n=0}^{N-1} L(\theta_n) \tag{4.13}$$

and $\delta \sum_{n=0}^{N-1} L(\theta_n)$ is the discretization of the previous loss function.

This is equivalent to the gradient descent method previously seen [38].

From the equations, it can be deduced that

$$p_n = -\nabla_{x_n} \operatorname{loss}(g(x_N), y) \tag{4.14}$$

and

$$\nabla_{x_n} x_{n+1} = \nabla_x (x_n + \delta f_n(x, \theta)) \tag{4.15}$$

Then,

$$\begin{aligned} \nabla_{\theta_n} J(\theta) = \nabla_{x_{n+1}} \operatorname{loss}(g(x_n), y)) \cdot \nabla_{\theta_n} x_{n+1} + \delta \nabla_{\theta_n} L(\theta_n) = \quad (4.16) \\ = -p_{n+1} \cdot \nabla_{\theta_n}(x_{n+1} + \delta f_n(x_n; \theta_n)) + \delta \nabla_{\theta_n} L(\theta_n) = \\ = -\nabla_{\theta_n} H_n \quad (4.17) \end{aligned}$$

Then we can see this algorithm as a generalisation of the gradient descent algorithm, but based on optimal control theory.

Previous knowledge from control theory of ODEs could now be used so as to understand why Deep Learning with ResNets architectures behaves well in practice.

Chapter 5

Deep Neural networks and ODEs

Consider the families of architectures of Neural Networks given by the following fundamental equations

$$\begin{cases} Y_{j+1} = \sigma(A_j Y_j + b_j) \\ Y_{j+1} = Y_j + \sigma(A_j Y_j + b_j) \\ Y_{j+1} = Y_j + \sigma(A_{j,2}\sigma(A_{j,1} Y_j + b_{j,1}) + b_{j,2}) \\ \dots \end{cases} \tag{5.1}$$

where the first one corresponds to a multilayer perceptron, the second one to a ResNet, ...

The variables Y_j are the features, the variables A are matrices and b vectors, and constitute the weights θ of the network; the dimensions of A and b are here unspecified, but they are arranged such that $dim(A_j Y_j) = dim(b_j)\ \forall j$. This dimension may depend on j, and it corresponds to the number of components of each layer.
We may define the number of components $C_j = dim(b_j)$.

The variables σ are functions, called activation functions, such that

$$\sigma_j : \mathbb{R}^{C_j} \to \Sigma^{C_j} \subseteq \mathbb{R}^{C_j} \tag{5.2}$$

For each vector $b_j = (b_j^1, ..., b_j^{C_j})$, a function σ is applied such that

$$\sigma b_j = \Big(\sigma(b_j^1), ..., \sigma(b_j^{C_j})\Big) \tag{5.3}$$

where there is abuse of notation for σ.

This function σ operates from $\mathbb{R}$ to Σ. The first architectures used as σ the functions sigmoid or tanh, such that $\Sigma = [0, 1]$.
More recent architectures use as activation functions ReLU, meaning Rectified Linear Units; they computed $ReLU(x) = max(x, 0)$, such that $\Sigma = \mathbb{R}_+$.

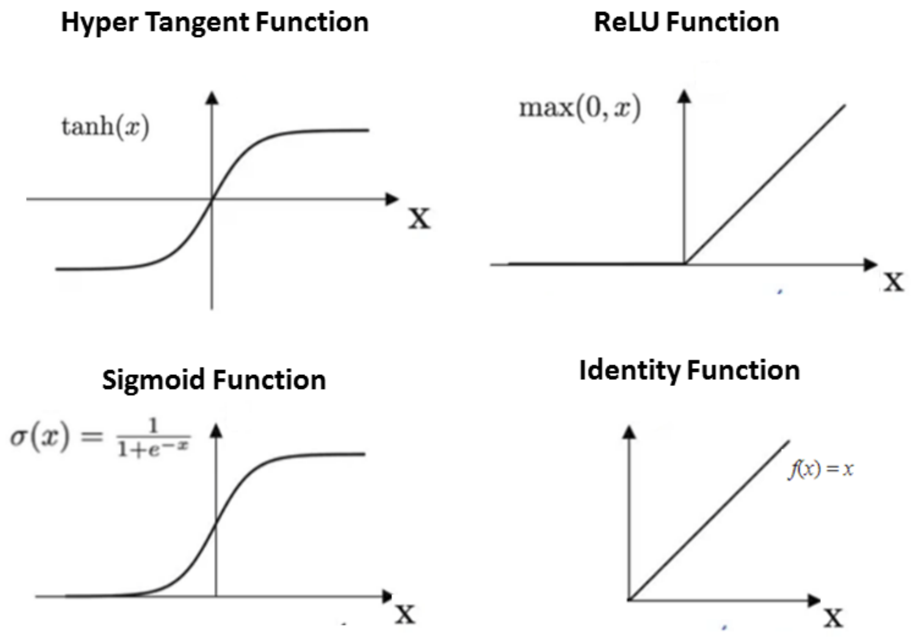


Figure 5.1: Different activation functions.

The functions σ can be chosen arbitrarily, yet it is important that they are easy to evaluate, and in order to create rich representation spaces or with good properties they are chosen to be continuous and monotonic. A reason for this is due to the Lebesgue's Theorem for the Differentiability of Monotone Functions.

It is interesting that ReLU functions are widely usted, since the derivative is not defined at 0, and it is not regular. ReLU can be seen as the limit of some smooth functions, for example the so called softplus functions.
It is important to take into account that Deep Learning is always done using a computer, and so the numerical deffects are intrinsic to this problem.

The problem of supervised learning using Deep Learning can be rewritten using this idealization.

Definition 5.0.1 (Supervised Deep Learning problem). *Given x_0 inputs, with correspondent labels y, the problem of supervised learning with deep learning architectures aims at finding the network weights (K, b) and classification weights (W, μ), by solving*

$$\begin{aligned} &minimize_{K,b,W,\mu}\ loss[g(Wx_N + \mu), y] + regularizer[K, b, W, \mu] \\ &subject\ to\ \ Y_{j+1} = activation(K_j Y_j + b_j), \quad \forall j = 0, ..., N-1 \end{aligned} \tag{5.4}$$

where the *activation* is given by the specific fundamental equation of the given architecture.

The ResNets architectures outperform more classical architectures (multilayer perceptrons) in specific settings such as in computer vision, they are better at the deep learning regime $L >> 1$, and are easier to characterize as discretization of ODEs [25].

A family of ResNets can be given by the following fundamental equation

$$Y_{j+1} = Y_j + A_{j,2}\sigma(A_{j,1}Y_j + b_j), \quad \forall j = 0, 1, ..., N-1 \tag{5.5}$$

where a new affine transformation $A_{j,2}$ has been added.

This architecture can be seen as a forward euler discretization of

$$\partial_t y(t) = A_2(t)\sigma(A_1(t)y(t) + b(t)), \quad y(0) = y_0 \tag{5.6}$$

A simplified notation is

$\theta(t) = (A_1(t), A_2(t), b(t))$
Then,

$$\partial_t y(t) = f(y, \theta(t)), \quad y(0) = y_0 \tag{5.7}$$

where $f(y, \theta) = A_2(t)\sigma(A_1(t)y(t) + b(t))$.

Instead of ResNets, it is easy to implement architectures based in higher-order difference equations as in equation 5.1.

Chapter 6

Approximation with neural networks

In order to see whether a function can be approximated by the composition of simpler functions as in by a Neural Network, the space of functions represented by Neural Networks needs to be introduced.
This space of functions is usually called *hypothesis space* in regression.

6.1 Universal Approximation Theorem(s)

Definition 6.1.1 (Representation space of a Neural Network). *Consider a neural network architecture defined by the fundamental equation $z^l = \mathcal{F}(z^{(l-1)}; \theta^{(l-1)})$, with $z^{(0)} = x$ the input data and $\hat{f}(x, \theta) = z^L$, being L the number of layers. The representation space of that neural network with L layers is, $\mathcal{N}^L$, is*

$$\mathcal{N}^L = \{\hat{f}(\cdot, \theta) \,|\, \theta \in \Theta\} \tag{6.1}$$

where Θ is the space of the weights.

Usually, $\Theta = \mathbb{R}^{\#\text{weights}}$, but a more general case is considered.

Considering fully connected neural networks, i.e. neural networks as defined by the fundamental equation

$$\begin{cases} z_k^{(L)} = f_L(z_k^{(L-1)}, \theta_L) \\ z_k^{(L-1)} = f_{L-1}(z_k^{(L-2)}, \theta_{L-1}) \\ \dots \\ z_k^{(1)} = f_1(x_k, \theta_1) \\ z_0 = x_k \end{cases} \tag{6.2}$$

where the functions f are given by shallow neural networks, and $\hat{f}(x_k;\theta) = z_k^{(L)}$.

Consider the dimension of $z_k^{(l)}$, also called the number of components C, constant throught l, $C = \dim(z_k^l)$, and L the number of layers.

Definition 6.1.2 (Hypothesis space of rectified networks)**.** *Consider a fully connected neural network, with a fundamental equation defined by a ReLU activation function, C components and L layers. The hypothesis space of a rectified feedforward neural network is*

$$\mathcal{H}[C,L] = \{\hat{f}(\cdot;\theta)\,|\,\hat{f}(\cdot,\theta) = \Lambda_{L+1}\circ\sigma\circ ...\circ\sigma\circ\Lambda_1)(\cdot)\,\theta\in\Theta\} \tag{6.3}$$

where Λ_l are affine transformations and σ is the ReLU activation function.

Lemma 6.1.1 (The more components, the better)**.**

$$\mathcal{H}[C,L] \subseteq \mathcal{H}[C+1,L]$$

Proof. Given a *neural network* with L layers, and C components per layer, and an arbitrary function represented by that neural network, $\hat{f}(x;\theta) = \psi\left(A^L z^L + b^L\right)$, where

$$\begin{cases} z^{k+1} = \sigma(A^k z^k + b^k) & \text{for } k = 0, ..., L-1 \\ z^0 = \vec{x}_i \in \mathbb{R}^d \end{cases}$$

and $A^k \in \mathbb{R}^{C\times C}$, $b^k \in \mathbb{R}^C$, $\forall k$

We can consider a function $\hat{g}(\cdot;\theta')$, with $(A')^k \in \mathbb{R}^{(C+1)\times(C+1)}$, $(b')^k \in \mathbb{R}^{C+1}$ defined as $\hat{f}_\theta$. If we consider

$$[(A')^k]_{ij} = [A^k]_{ij}\ \forall i,j \leq C,\ \forall k \quad [(A')^k] = 0 \text{ otherwise}$$
$$[(b')^k]_i = [b^k]_i\ \forall i \leq C\ \forall k \quad [(b')^k] = 0 \text{ otherwise}$$

then, $\hat{g}_\theta(\cdot) = \hat{f}_\theta(\cdot)$ ■

Lemma 6.1.2 (The more layers, the better)**.**

$$\mathcal{H}[C,L] \subseteq \mathcal{H}[C,L+1]$$

Proof. We consider $\hat{f}(\cdot,\theta)$ defined as $\hat{f}(x;\theta) = \psi\left(A^L z^L + b^L\right)$,

$$\begin{cases} z^{k+1} = \sigma(A^k z^k + b^k) & \text{for } k = 0, ..., L-1 \\ z^0 = \vec{x}_i \in \mathbb{R}^d \end{cases}$$

We can consider $\hat{g}(\cdot;\theta') = \sigma(A^{L+1}z'^{L+1} + b^{L+1})$ such that $z'^k = z^k \; \forall k < q$ $(1 < q < L-1)$, $z'^{k+1} = z^k \; \forall k > q$ and

$$z^q = \sigma(\mathbb{I}z^{q-1} + 0) = \sigma(z^{q-1}) = z^{q-1}$$

$$\blacksquare$$

Corollary 6.1.1.

$$\mathcal{H}[C, L] \subseteq \mathcal{H}[C', L'] \text{ for } C' \geq C, L' \geq L$$

This results could be analogously proven for different activations functions such as sigmoid and tanh. Indeed, the only requirement is that each layer can reproduce the identity function.

Theorem 6.1.1 (Cybenko '89, MCSS [12])**.** *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded and continuous function. Let $d \geq 1$ and $L = 1$.*
For any $\epsilon > 0$, and any $f \in \mathcal{C}^0([0,1]^d)$ there exists $N_1 \in \mathbb{N}$, $A^0 \in \mathbb{R}^{N_1 \times d}$, $A^1 \in \mathbb{R}^{1 \times N_1}$ and $b^0 \in \mathbb{R}^{N_1}$ such that

$$f_L(\vec{x}) = A^1 \sigma(A^0 \vec{x} + b^0)$$

satisfies

$$\sup_{\vec{x} \in [0,1]^d} |f(\vec{x}) - f_L(\vec{x})|$$

Theorem 6.1.2 (Poggio et al. '17 [45])**.** *Let $\sigma \in C^\infty(\mathbb{R})$ be bounded and not a polynomial. Let $d \geq 1$ and $L = 1$.*
For any $\epsilon > 0$ and any $f \in \mathcal{C}^k([0,1]^d)$ with $k \geq 1$ there exist coefficients $A^0 \in \mathbb{R}^{N_1 \times d}$, $A^1 \in \mathbb{R}^{1 \times N_1}$, and $b^0 \in \mathbb{R}^{N_1}$ with

$$N_1 = \mathcal{O}\left(\epsilon^{-d/k}\right)$$

This means that a sufficient number of neurons with one hidden layer can approximate C^k functions. However, the number of neurons grow exponentially in d, and so does the storage needed. In this sense, this approach suffers from the curse of dimensionality.

In the case of ReLU, $\sigma(x) = \max(x, 0)$, we have a theorem by Hanin:

Theorem 6.1.3 (Hanin '17 [54]). *Let $d \geq 1$ and let $f : [0,1]^d \to \mathbb{R}$ be a positive and continuous functions with $\|f\|_\infty = 1$. Then for any $\epsilon > 0$, there exists a neural network f_L with ReLU activation of depth*

$$L = \frac{2d!}{\omega_f(\epsilon)^d}$$

and width $\max_k N_k \leq d+3$ such that

$$\|f - f_L\|_\infty \leq \epsilon$$

Where $\omega_f : \delta \mapsto \sup\{|f(x) - f(y)| \,:\, |x-y| \leq \delta\}$ denotes the modulus of continuity of f.

Using the previous notation, these results can be stated as

Proposition 6.1.1 (Universal Approximation Theorem(s)). *Under some conditions to the activation function σ (that tanh, sigmoid and ReLU fulfill), there exists L^0, C^0 such that*

$$\mathcal{H}[L,C] \text{ is dense in } \mathcal{C}^0 \text{ for } L \geq L_0,\ C \geq C_0$$

6.2 Approximation with ResNets

The approximation results from shallow networks also apply to ResNets given that the residual parts are generated by shallow network.

It is also interesting to study the idealized version of ResNets in terms of ODE so as to study the approximation.

6.2.1 $\mathcal{H}_\infty$ hypothesis space

Recall the hypothesis space generated by flow maps

$$\mathcal{H}_T = \{z \mapsto g(x_T) \,:\, \dot{x}_t = f(x_t,\ \theta_t) = x_0 = z,\ \theta_t \in \Theta\} \tag{6.4}$$

If any depth is considered, the $\mathcal{H}_\infty$ hypothesis space can be defined such that

$$\mathcal{H}_\infty \equiv \cup_{T \geq 0} \mathcal{H}_T \tag{6.5}$$

The question that arises is what functions is it possible to approximate using the hypothesis space $\mathcal{H}_\infty$.

Lemma 6.2.1. *Let $\mathcal{X}$ be a compact subset of $\mathbb{R}$, and $\mathcal{Y} = \mathbb{R}$. Let the terminal loss function $g(x) = x$.*
Then, $F \in \mathcal{H}_\infty$ must be continuous and increasing.

Proof. From basic ODE theory, the solutions are known to exist, and must be continuous, as can expressed in the integral form; the fact that they are increasing is proven due to uniqueness.

■

The reverse, in the case of ReLU networks, is also true.

Proposition 6.2.1 (Sufficient conditions for approximation [39]). *Consider $f(x, \theta) = a\sigma(wx + b)$, $\sigma(z) = max(0, z)$, $\theta = (a, w, b) = \mathbb{R}^3$.*
Let $p \in [1, \infty)$.
Given $F^* : \mathcal{X} \to \mathbb{R}$ a continuous and increasing function. For any $\epsilon > 0$ there exists a function F in the hypothesis space $\mathcal{H}_\infty$ such that

$$\|F^* - F\|_{L^p(\mathcal{X})} \leq \epsilon$$

One proof of this can be based on the Universal Approximation Theorems seen before.

Using this ODE perspective, it is possible to ask which are the sufficient conditions for the residual blocks $f(x_t, \theta_t)$ and activation function such that the ResNets can approximate any function.

The whole sufficient conditions for approximation are given by a theorem by Li et al.

Theorem 6.2.1 (Sufficient conditions for approximation with $n \geq 2$ [39])**.** *Let $n \geq 2$, $p \in [1, \infty)$ and $\mathcal{F}$ be some control family. Let the target function $F : \mathbb{R}^n \to \mathbb{R}^m$ be continuous and $K \subset \mathbb{R}^n$ be any compact set. Suppose that $g : \mathbb{R}^n \to \mathbb{R}^m$ is Lipschitz and $F(K) \subset g(\mathbb{R}^n)$. Consider the hypothesis space*

$$\mathcal{H}_{ode} = \cup_{T>0}\{x \mapsto g(z(T)) \mid \dot{z}(t) = f_t(z(t)), f_t \in \mathcal{F}, z_0 = x, t \in [0, T]\}$$

Assume $\mathcal{F}$ satisfies the following conditions:

1. *$\mathcal{F}$ is restriced affine invariant.*
2. *$\overline{CH}(\mathcal{F})$ contains a well function.*

Then, for any $\epsilon > 0$ there exists $\hat{F} \in \mathcal{H}_{ode}$ such that

$$\|F - \hat{F}\|_{L^p(K)} \leq \epsilon$$

Chapter 7

Neural Networks as coordinate transformations

By seeing the intermediate activations as discrete map $\Phi(\cdot;t)$, the forward pass can be understood as an application that transforms an initial manifold $\mathbf{x}$ to a final manifold as transformed by the map $\Phi(\mathbf{x},T)$.

The attempt is to construct a general theory in which Neural Networks can be seen as geometric transformation of the data manifold.
Some attempts have been done in order to do this, but no successful theory has been found yet that explains the phenomenology when using Neural Networks as approximators. This section is aimed at stating the starting point of this approach, as inspired by previous work [23] and by the characterization of the forward pass as a dynamical system.

To study this perspective from a topological point of view is interesting. In particular, some results of this section would be focused in the case of binary classification, but can be easily generalised to a more general classification problem, or even to a general supervised learning problem.

The main issue regarding this approach is that the topology of the initial manifold, in terms of the different classes, is needed so as to give strong and general results. It is not expected that this approach yields useful results without specifying the topological stucture of the training data.
Therefore, this study needs to be done combining results of *topological data analysis* [**?**] [63], that is not included in the scope of this work.

7.1 Neural Networks and manifolds

The difficult part when addressing Deep Learning from a mathematical perspective is that it is difficult to characterize what are neural networks doing.

Again using the Dynamical System perspective, the number of neurons per layer correspond to the dimension of the dynamical system.

If instead of considering that the initial datum covers all the input space, but rather a manifold, it is more interesting to see the neural networks as if they deform the input manifold, to a final one in which data can be easily classified.

In order to stablish a formal description, we need that the transformations at each layer are homeomorphisms.

Theorem 7.1.1 (Layers as homeomorphisms)**.** *Let a neural layer have N inputs and N outputs, and suppose that uses an activation function which is continuous and with continuous inverse. Then, this layer represents an homeomorphism given that the weight matrix W is non-singular.*

Proof. It is known that a continuous bijection from a topological space to a Hausdorff topological space is a homeomorphism.
Since the matrix W is non-singular, it has a linear inverse, which is continuous. The composition of continuous functions is continuous, and so we only need that the activation function is continuous and bijective, which it by hypothesis and if the appropiate range is considered.

■

Corollary 7.1.1. *A Neural Network represents a homeomorphism if it is given by the composition of layers that represents homeomorphisms.*

Although *tanh*, *sigmoid*, ... are bijective, ReLU is not. Moreover, it is not guaranteed that the matrix W is non-singular.
This approach of Neural Network "deforming" manifolds still useful for visualization purposes, and this representation still holds for non-homeomorphic transformations.

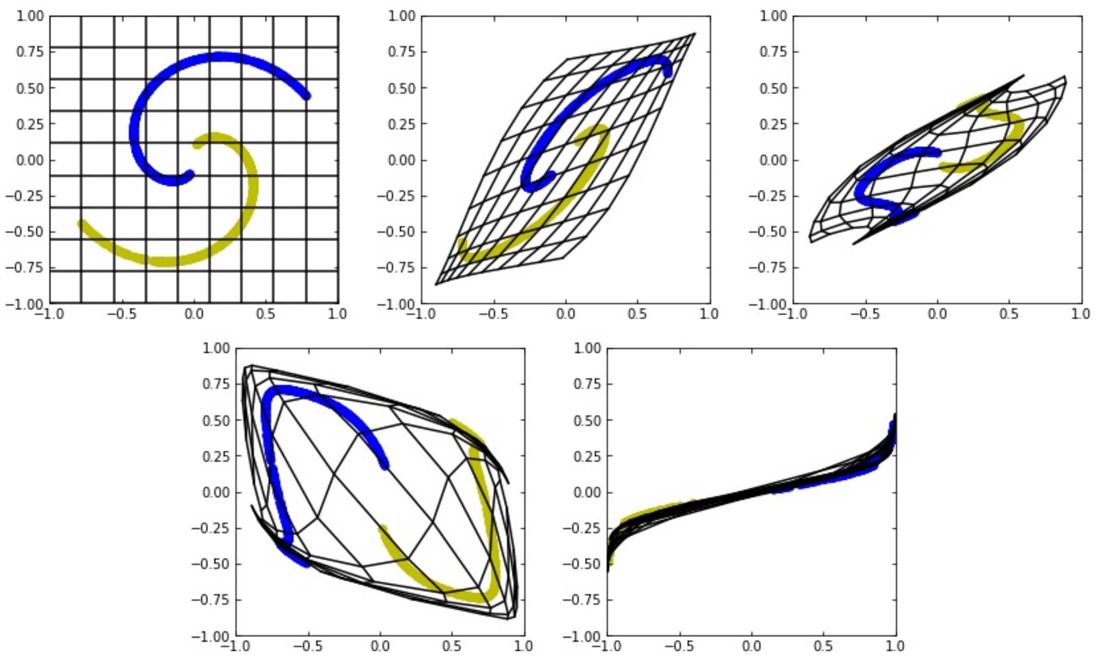


Figure 7.1: Scheme of a multilayer perceptron with dimension of the input 2, dimension of the blocks 2 and a feedforward function given by f.

7.2 Riemannian geometry of neural networks

A more technical and formal approach can be given.

$$\begin{tikzcd}
 & & M \arrow[dll, bend right, "x^{(0)}"'] \arrow[dl, "x^{(1)}"'] \arrow[d, "x^{(2)}"] \arrow[dr, "x^{(3)}"] \arrow[drr, bend left, "x^{(L)}"] & & \\
x^{(0)}(M) \arrow[r, "\varphi^{(0)}"] & x^{(1)}(M) \arrow[r, "\varphi^{(1)}"] & x^{(2)}(M) \arrow[r, "\varphi^{(2)}"] & x^{(3)}(M) \arrow[r, dashed, "\varphi^{(L)}"] & x^{(L)}(M)
\end{tikzcd}$$

Figure 7.2: From [23] . Coordinate systems $x^{(l+1)} = \phi^{(l)} \circ ... \circ \phi^{(1)} \circ \phi^{(0)} \circ x^{(0)}$ induced by the coordinate transformations $\phi^{(l)} : x^{(l)}(M) \to \left(\phi^{(l)} \circ x^{(l)}\right)(M)$ learned by the neural network. The pullback metric $g_{x^{(l)}(M)}(X,Y) = g_{(\phi^{(l)} \circ x^{(l)})(M)}\left(\phi_*^{(l)} X, \phi_*^{(l)} Y\right)$ backpropagates the coordinate representation of the metric tensor from layer $l+1$ to layer l, via the pushforward map $\phi_*^{(l)} : Tx^{(l)}(M) \to T\left(\phi^{(l)} \circ x^{(l)}\right)$

Notation and background

The notation presented in here is as in [23].
$x^{(l)}$ is the lth coordinate system, and $\phi^{(l)}$ the lth coordinate transformation.
If the index is not in parenthesis, it is referred to the component of a vector, and a supscropt free index means it is referred to the component of a covector.
Einstein notation is used.

Definition 7.2.1 (Topological manifold [62])**.** *A topological manifold M of dimension* $\dim M$ *is a Hausdorff, paracompact topological space that is locally homeomorphic to* $\mathbb{R}^{\dim}$

Definition 7.2.2 (Coordinate system)**.** *The homeomorphism $x : U \to x(U) \subseteq \mathbb{R}^{\dim M}$ with M a topological manifold is called a coordinate system on $U \subseteq M$*

A feedforward network learns coordinate transformations $\phi^{(l)} : x^{(l)}(M) \to \left(\phi^{(l)} \circ x^{(l)}\right)(M)$, where the new coordinates $x^{(l+1)} \equiv \phi^{(l)}\left(x^{(l)}\right); M \to x^{(l+1)}(M)$, and it is initialized in Cartesian coordinates $x^{(0)} : M \to x^{(0)}(M)$ [23].

Every data point $q \in M$ has corresponding coordinates to some coordinate system. In the intermediate layer $l+1$, q is represented as

$$x^{(l+1)}(q) \equiv \left(\phi^{(l)} \circ ... \circ \phi^{(1)} \circ \phi^{(0)} \circ x^{(0)}\right)(q). \tag{7.1}$$

And the output reprpesentation is $x^{(L)}(M) \subseteq \mathbb{R}^d$.

Proposition 7.2.1 (Feedforward networks transformations)**.** *Given an activation function f (ReLU, tanh or sigmoid), a standard feedforward network transforms coordinates as*

$$x^{(l+1)} \equiv \phi^{(l)}\left(x^{(l)}\right) \equiv f(x^{(l)}; l). \tag{7.2}$$

The activation function $ReLU(\cdot) = max(.,0)$ is not bijective, and thus it is not a proper coordinate transformation. The notation is abused in this sense.

Proposition 7.2.2 (ResNets transformations)**.** *A residual network (ResNet) transforms coordinates as*

$$x^{(l+1)} \equiv \phi^{(l)}\left(x^{(l)}\right) \equiv x^{(l)} + f(x^{(l)}; l) \tag{7.3}$$

Lemma 7.2.1. *The coordinates as transformed by a ResNet are global coordinates over the entire manifold.*

Idea of the proof. A residual network is bijective, even with ReLU activations. ■

A ResNet with ReLU activations is piecewise linear, with kinks of infinite curvature (since it is not differentiable at the intersections of different linear regions).

Definition 7.2.3 (Softmax transformation). *A Softmax coordinate transformation is defined by*

$$softmax\left(W^{(L)} \cdot x^{(L)}\right)^j \equiv \frac{e^{W^{(L)j}x^{(L)}}}{\sum_{k=1}^{K} e^{W^{(L)k}x^{(L)}}} \tag{7.4}$$

being the probability of $q \in M$ being from class j. i.e. $softmax\left(W^{(L)} \cdot x^{(L)}(q)\right)^j = \mathbb{P}(Y = j \mid X = q)$

7.3 ResNets as differentiable coordinate transformations

Consider a feedforward newtork, described by the fundamental equation

$$x^{(l+1)} \equiv f(x^{(l)}; l) \tag{7.5}$$

if f is of class $\mathcal{C}^0$. Then if the feedforward network is a C^0 coordinate transformation.

A ResNet architecture is given by the fundamental equation

$$x^{(l+1)} = x^{(l)} + f(x^{(l)}; l) \tag{7.6}$$

since it is well-behaved at the limit $L \to \infty$, l can be redefined such that

$$x^{(l+1)} \approx x^{(l)} + f(x^{(l)}; l)\Delta l \tag{7.7}$$

where $\Delta l = 1/L$ and a uniform partition of $[0, 1]$.

Then the ODE approximation for ResNets corresponds to a class of coordinates transformations, in which k^{th} order differentiable smoothness can be imposed such that

$$\delta x^{(l)} \equiv x^{(l+1)} - x^{(l)} \approx f(x^{(l)}; l)\Delta l \tag{7.8}$$

$$\delta^2 x^{(l)} \equiv x^{(l+1)} - 2x^{(l)} + x^{(l-1)} \approx f(x^{(l);l}\Delta l^2 \tag{7.9}$$

$$\dots$$

ResNet architectures as defined by 7.8 correspond to $\mathcal{C}^1$ transformations with $\mathcal{O}(\Delta l)$ error, and higher order architectures can be constructed by doing central differencing approximations of $\mathcal{C}^1$ coordinate transformations, which would give an error of $\mathcal{O}(\Delta l^2)$, and iteratively to $k-$order smoothness architectures.

The corresponding differential architectures can be easily obtained by taking the limit $\Delta l \to 0$, i.e.

$$\frac{dx^{(l)}}{dl} \equiv \lim_{\Delta l \to 0} \frac{x^{(l+\Delta l)} - x^{(l)}}{\Delta l} = f(x^{(l)}; l) \tag{7.10}$$

In order for this to make sense in practice, it is needed that the limit $L \to \infty$ can be implemented, and thus that ResNets architectures are really stable. This agrees with experiments, for example for $L \approx 100$. Since we are considering the discrete case, the approximation is considered to be good.

In practice, Δl does not need to be a constant discretization. The more general case in which $\Delta l = \Delta l(n)$ with n the index of the layer can be trivially implemented, and the problem would be well-defined if $\max \Delta l(n) \to 0$.

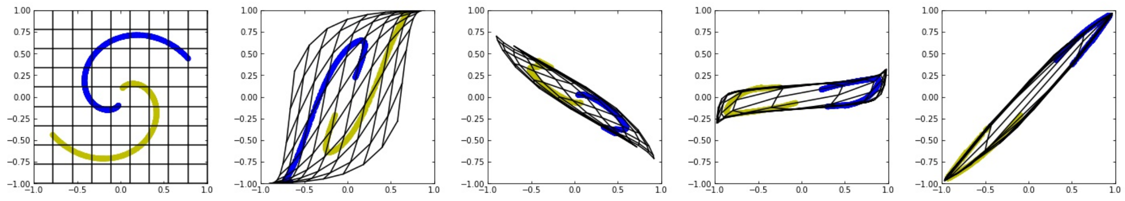


Figure 7.3: Scheme of a multilayer perceptron with dimension of the input 2, dimension of the blocks 2 and a feedforward function given by f.

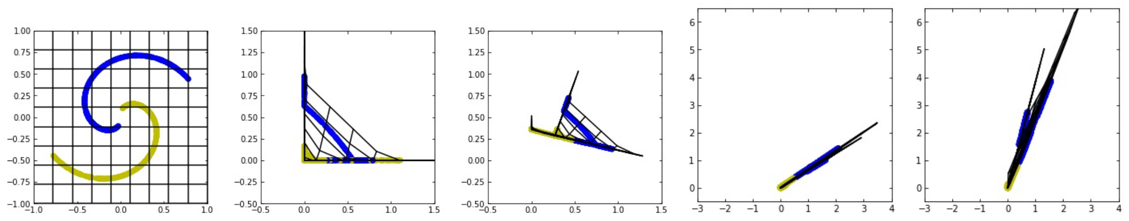


Figure 7.4: Scheme of a multilayer perceptron with dimension of the input 2, dimension of the blocks 2 and a feedforward function given by f.

Part III

Related work

Chapter 8

From multilayer perceptrons to ResNets

There are three widely used activation functions: tanh, ReLU and sigmoid.

8.1 Vanishing gradients

The vanishing gradient problems observed in training Neural Networks is a series of phenomena related to the fact that the gradient at some point can be arbitrarily close to 0 with respect to some neurons, but that point is not close to a minima.

An example of this is a constant function represented by a network. The backpropagation training would not be able to update that function, and it may not be a minima of the problem specified.

8.1.1 Tanh and sigmoid activations

The hyperbolic tangent is given by

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{8.1}$$

whereas the sigmoid is given by

$$\tilde{\sigma}(x) = \frac{1}{1 + e^{-x}} \tag{8.2}$$

It is easy to see that $tanh(x) = 2\tilde{\sigma}(2x) - 1$.

So as for the backpropagation algorithm to be efficient, convergence is usually proven to be faster when the mean of the weights is close to 0 [37].
For that reason, it is prefered that *tanh* activation is used, since the mean of *tanh* under a symmetric distribution of x is 0.

The problem with the *tanh* activation is that the gradients are saturated easily, i.e. they are arbitrarily close to 0, and so make training more difficult.

So as to see this,

$$\frac{d}{dx} tanh(x) = sech^2(x) = \frac{4}{(e^{-x} + e^x)^2}$$

New activation functions are proposed such that the derivative of the activation function does not vanish easily.

8.1.2 Dying ReLU

The most used activation function in practice is the *rectified unit activation*, $ReLU(x) = max(x, 0)$. [59][1] [68]

The idea is that the derivative of *ReLU* is 1 for $x > 0$ and 0 for $x \leq 0$, and since so the gradient does not vanish for $x > 1$.

When using *rectified unit activations* the problem of *vanishing gradients* is partially solved, but due to the fact that a *ReLU* function is not smooth, and that for $x < 0$ the gradients are 0, some problems arise.

It is possible that a *ReLU neuron* gets into a state that is not able to communicate any information through the network; in this case, we will call this neuron a *dead neuron.*

Definition 8.1.1 (Dead neuron)**.** *A rectified neuron, which represents a function* $h(\cdot) = \max(A(\cdot) + B), 0)$ *is said to be dead when* $h(x) = 0$ *for all* x *the input space of that layer.*

Lemma 8.1.1. *When using a gradient descent method, and if all the weights of other neurons are fixed, a dead neuron would not be updated. The state of that neuron given by the weights $\{A, B\}$ is absorbent.*

Proof. The update on the weights (in this case $\{A, B\}$ is given by the partial derivatives w.r.t. that weights. That is, the $(\Delta A, \Delta B)$ update to (A, B) is proportional to $(\frac{\partial h}{\partial A}, \frac{\partial h}{\partial B})$. Since $h(x) = 0 \; \forall x$ in $\mathcal{X}$, $\frac{\partial h}{\partial A} = 0 = \frac{\partial h}{\partial B} \; \forall x \in \mathcal{X}$ the input space of that neuron, the weights would not be updated. ■

Although the probability that a *dead neuron* recovers is not high [44], still it is in theory possible.

Lemma 8.1.2 (A dead neuron can recover)**.** *Let a dead neuron be in an intermediate layer, such that its represented function is $h(\cdot) = \max(A(\cdot) + B, 0)$ and $h(x) = 0 \; \forall x \in \mathcal{X}$. Then it is in general possible that the neuron, after the neural network is trained with a gradient descent method, is such that $h(x) \neq 0$ for some $x \in \mathcal{X}$*

Proof. Although the weights of that neuron would not be inmediately updated (see previous lemma), it is possible that the input space $\mathcal{X}$, which is the *output space* of the previous layer, updates such that we can find a value of $x \in \mathcal{X}$ such that $h(x) \neq 0$. ■

Proposition 8.1.1. *A dead neuron has the effect of reducing the effective dimensions of the input space of posterior neurons.*

Proof. Suppose a neuron in the layer l is dead. Then, the neurons in layer $l+1$ would have as an input $\{h_1(\cdot), ..., h_C(\cdot)\}$ with $h_d(\cdot)$ a dead neuron $1 < d < C$. Since this component would always be 0, it would not have an effect to the affine transformation done by the neuron. ■

The state $\{A, B\}$ such that a neuron is dead could be reached due to different reasons. It is trivial to see that initialization plays a very important role (i.e. the weights A, B could be initialized such that the neurons is born dead), but it could also be that $A, B, \mathcal{X}$ are updated such that the neuron is dead.

A systematic approach of this problem, and possible solutions, could be done in specific scenarios [41] [2] [14].

However, *ReLU* still the most used in Deep Learning, and the problem of dying ReLU is usually solved partially by tuning the network architecture.

Although there are many empirical and heuristical evidences that ReLU multilayer perceptrons "die", little is known about its theoretical analysis.

In order to alleviate the dying ReLU problem, the easier way is to change the initialization procedure. However, the initialization must be done at random. The map $\theta \to \hat{f}(\cdot;\theta)$ is not easy to characterize, nor it can be done in general. Therefore, there is no *a prioiri* weights that are better than others, and studies must be done case-by-case.

There is empirical and heuristical work for choosing the distribution of the weights, and they are mainly initialized using a uniform distribution or a normal one, and tuning the standar deviation.

The theoretical background of initialization is not completely understood, and few works have been done in the general case.
However, it is known that the limit $L \to \infty$ in feed-forward neural networks, as opposed to ResNets, is not well-defined, in the sense that the functions represented are constant and then the network is not able to learn anything.

This can be seen as an especific case of being stuck in a minima that is not a global minima. However, in practical applications, the cost landscape is too complex, and the weights are not in a global minima but rather in a "good" local minima, which is close to the optimal case.

Some general results are presented in here for the case of rectified networks. These results imply that the number of components N, or "dimensions" of the dynamical system, must be large so as to avoid this problems.

Lemma 8.1.3. *Let $\mathcal{N}^L(x)$ be a $L-$layer ReLU neural network with N_l neurons at the $l-$th layer. Suppose that all weights are randomy independently generated from probability distributions $\mathbb{P}(W_l^l z = 0) = 0$ for any nonzero vector $z \in \mathbb{R}^{N_{l-1}}$ and any $j-$th row of W^l. Then,*

$$\mathbb{P}(\mathcal{N}^L(x) \text{ is born dead in } \mathcal{D}) = \mathbb{P}(\exists l \in \{1, ..., L-1\} \text{ such that } \phi(\mathcal{N}^l(x))) = 0 \, \forall x \in \mathcal{D}$$

where $\mathcal{D} = B_r(0)$ is the training domain, with r any positive real number.

Idea of the proof. It is clear that $\phi(\mathcal{N}^L(x)) = \phi(\mathcal{N}^L(0))$. Starting at $l = L$, recursively, it is clear that it might exist such an l.
Conversely, if such an l did not exist, then $\mathcal{N}^L$ could not be 0.

■

Then, we know that the more layers we have (if we mantain N constant), the easier it would be for a feed-forward neural network to born dead.

Indeed, the probability that a feed-forward neural network, as initialized by a zero-mean probability distribution, is born dead becomes 1 as $L \to \infty$.

Theorem 8.1.1 (Taki '17 [58]). *Let $\mathcal{N}^L(x)$ be a ReLU neural network with L layers, each of them with N neurons.*
Suppose that the weights and biases are randomly initialized from probability distributions which satisfy

$$\mathbb{P}\left(\langle W_j^l, z_+ \rangle + b_j^l < 0, \forall z_+ \in \mathbb{R}_+^N\right) \geq p > 0 \quad \forall j \in \{1, ..., N\}$$

for some constant $p > 0$.

Then,

$$\lim_{L \to \infty} \mathbb{P}\left(\mathcal{N}^L(x; \theta_L) \text{is born dead}\right) = 1$$

Theorem 8.1.2 ([58]). *Let $\mathcal{N}^L(x)$ be a ReLU neural network with L layers, each of the having N_l neurons. Suppose that the weights are initialiez from symmetric probabilities distributions around* 0 *and that all biases are either drawn from a symmetric distribution or set to zero.*

Then,

$$\mathbb{P}\left(\mathcal{N}^L(x, \theta_L) \text{ is born dead }\right) \leq 1 - \prod_{l=1}^{L-1} (1 - (1/2)^{N_l})$$

Idea of the proof. L ■

Corollary 8.1.1. *Let $N_l = N$ for all l. Then,*

$$\lim_{L\to\infty} \mathbb{P}(\mathcal{N}^L(x;\theta_L) \text{ is born dead }) = 1$$
$$\lim_{N\to\infty} \mathbb{P}(\mathcal{N}^L(x;\theta_L) \text{ is born dead }) = 0$$

Theorem 8.1.3 (Taki '17 [58]). *Suppose that the feed-forward ReLU neural network is born dead. Then, for any loss function $\mathcal{L}$, and for any gradient base method, the ReLU network is optimized to be a constant function which minimizes the loss.*

8.1.2.1 Numerical example

Suppose a *neural network* with $C = 2$ components per layer, and a number of layers L.

The input space of the network is $X = [-1, 1] \in \mathbb{R}$ and the pair $\{x_j, y_j\}_{j=1}^{S}$ is given such that

$$y_j = 0 + \mathbf{1}_{x_j > 0}$$

It is clear that with $L = 1$ this classification problem could be solved easily if using as a terminal loss function a *sigmoid*. Indeed, a single neuron $h(\cdot) = max(A(\cdot) + B)$ could solve this problem by using $A = 1$, $B = 0$.

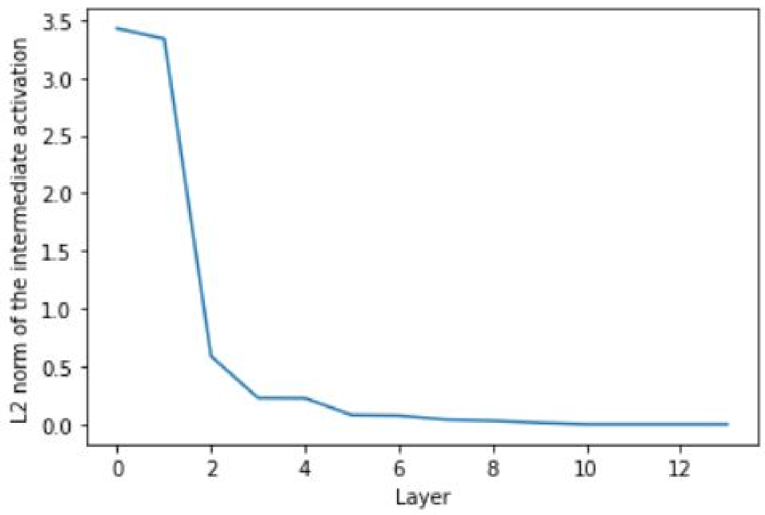


Figure 8.1: Dissipative effect of a *neural network* with $C = 2$ components and $L = 15$ layers

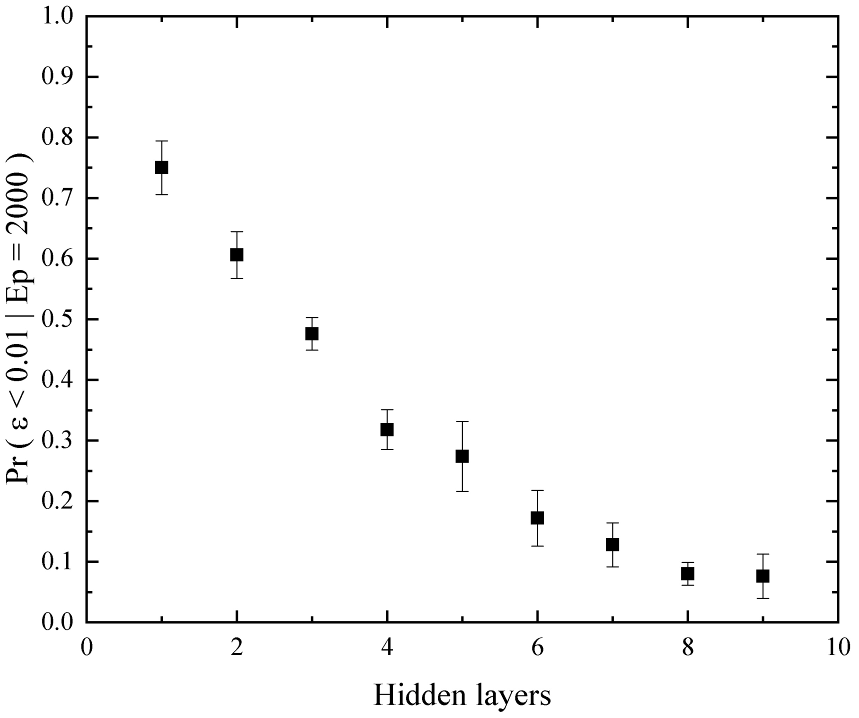


Figure 8.2: Effect of the number of layers with $C = 2$ components and $L = 15$ layers

8.1.3 Initialization

Some experiments are presented related to the problem of initialization.

The weights of a multilayer perceptron with C components and L layers are usually initialized following a uniform random distribution from $[-1, 1]$.

Different initializations are proposed, all of them following a uniform random distribution $[-\gamma, +\gamma]$ for different architectures.

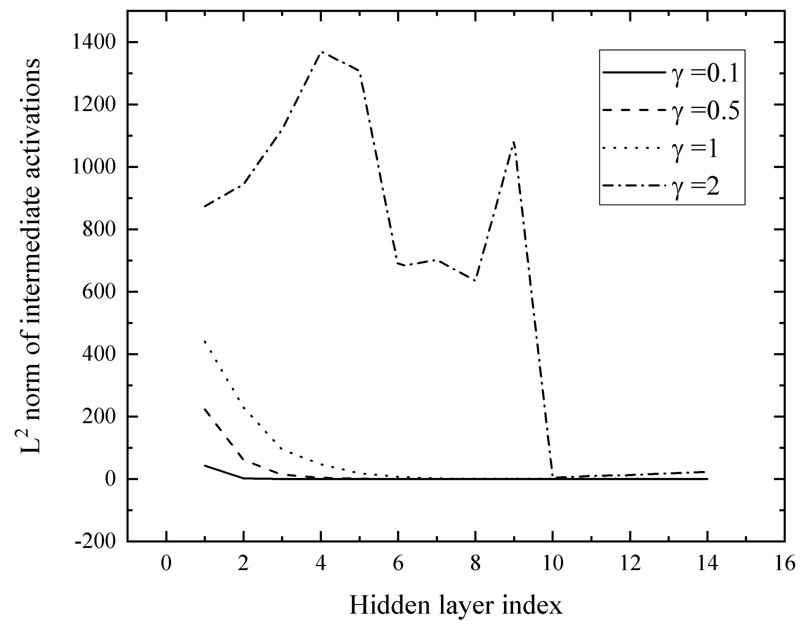


Figure 8.3: L^2 norm of the intermediate activations in multilayer perceptrons with 15 hidden layers and 2 neurons per layer, with an initialization following a uniform random distribution $[-\gamma, +\gamma]$, for an input $x \in [-1, 1]$ and over 100 experiments.

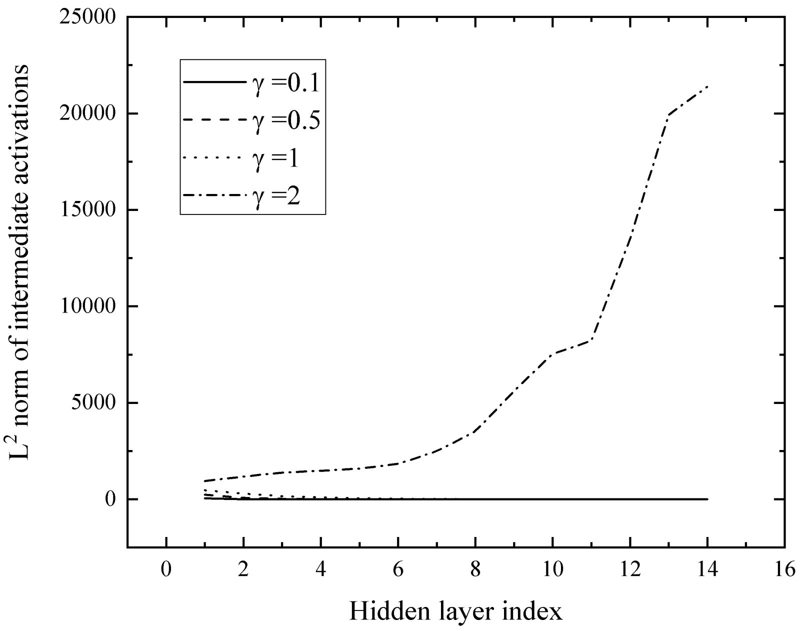


Figure 8.4: L^2 norm of the intermediate activations in multilayer perceptrons with 15 hidden layers and 3 neurons per layer, with an initialization following a uniform random distribution $[-\gamma, +\gamma]$, for an input $x \in [-1, 1]$ and over 100 experiments.

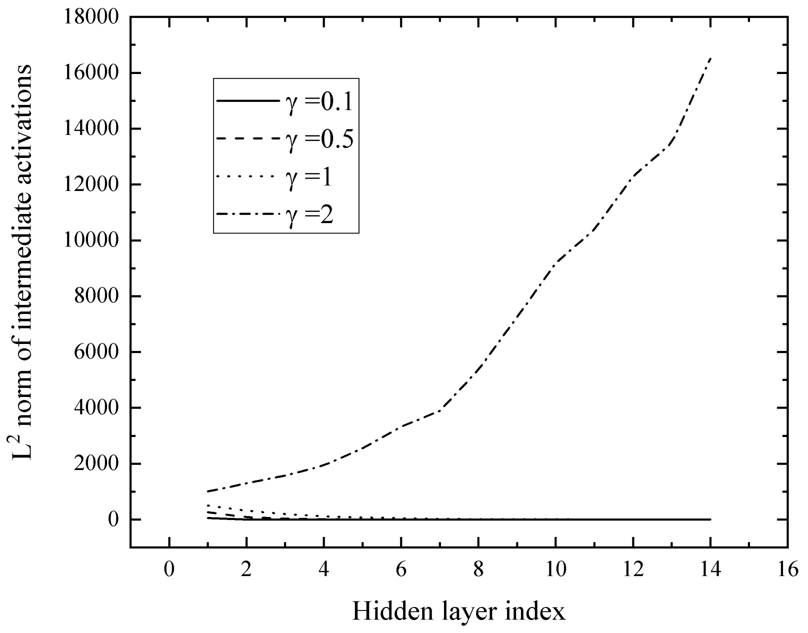


Figure 8.5: L^2 norm of the intermediate activations in multilayer perceptrons with 15 hidden layers and 4 neurons per layer, with an initialization following a uniform random distribution $[-\gamma, +\gamma]$, for an input $x \in [-1, 1]$ and over 100 experiments.

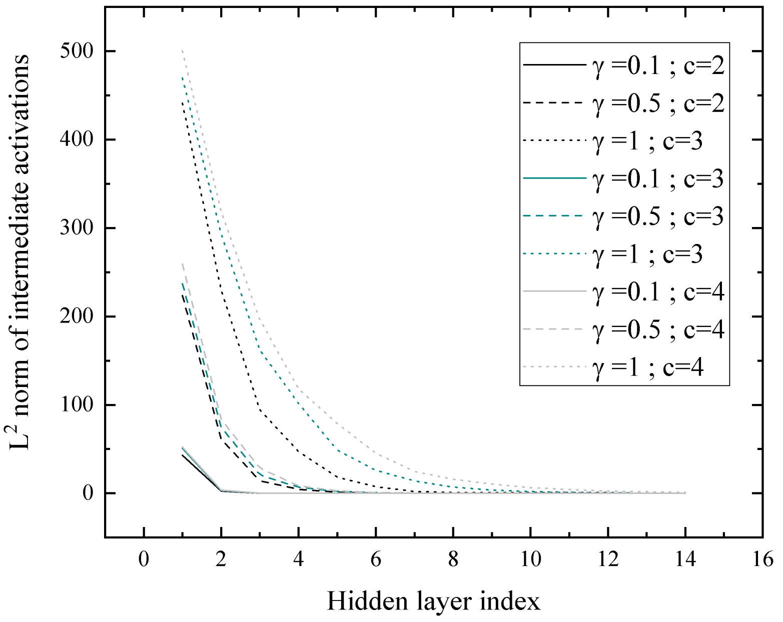


Figure 8.6: Effect of initialization for a different multilayer perceptron architectures, and initialized following a uniform random distribution $[-\gamma, +\gamma]$.

8.2 Residual networks

In order to avoid the *vanishing gradient problem* (see **??**), skip connections provide an alternative path for the gradient in the backpropagation algorithm. The results of implementing skip connections have been proven to have a positive impact in the learning process [15] [65] [48] and allow for having more layers than using perceptrons. [67]

Skip connections skip some layer in the neural network and feeds the output of one layer as the input of the next layers in the architecture, instead of only feed it as an input for the next layer. Formally,

Definition 8.2.1 (Skip connection)**.** *Let a neural network with L layers be represented as $\{z_0, ..., z_L\}$, with z_i the intermediate activations, and $z_0 = x_0$ given by the data. Then, there is a skip connection when*

$$z_{k+q} = \mathcal{F}_{k+q}(z_k, z_{k+1}, ..., z_{k+q-1})$$

is such that $\mathcal{F}(z_k, z_{k+1} = \vec{0}, ..., z_{q-1} = \vec{0}) = z_k$, with $k, q \in \mathbb{Z}$, $k \geq 1$ and $q > 1$.

If a neural network has a skip connection, then if some intermediate layers ($\{z_{k+1}, ...z_{k+q-1}\}$) have activations arbitrarily close to 0, the final layer may not be close to 0; note that this does not happen with multilayer perceptrons.

An example of skip connection is addition. That is, given

$$z_{k+q} = \mathcal{L}_{k+q}(z_k, z_{k+1}, ..., z_{q-1})$$

the function

$$\mathcal{L}_{k+q} = z_k + \mathcal{G}_{k+q}$$

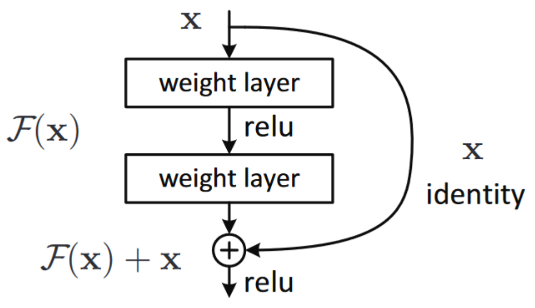
with $\mathcal{G}_{k+q}$ defined by a multilayer perceptron, is such that, if $\mathcal{G}_{k+q} = \vec{0}$, then $\mathcal{L}_{k+q} = z_k$. In order for this to make sense, the dimension of $\mathcal{G}_{k+q}$ must be the same as the dimension of z_k.

Skip connections can be introduced in a neural network architecture inductively, such that

$$z_{k+q} = \mathcal{F}_{k+q}(z_k, z_{k+1}, ..., z_{k+q-1})$$

$$z_k = \mathcal{F}_{k+q}(z_k, z_{k+1} = \vec{0}, ..., z_{k+q-1} = \vec{0})$$
$$\text{for every } \ k = q \cdot j, \quad j = \{1, 2, 3, ...\}$$

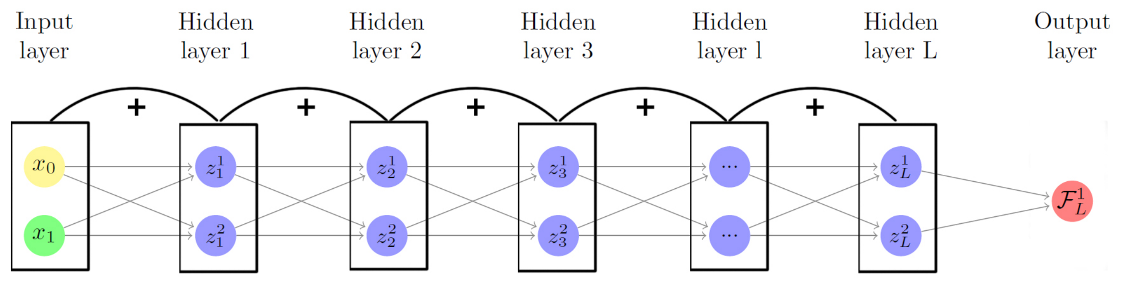
Then, if the skip connection is implemented as an addition, that neural network would be called a *ResNet*.



x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 8.7: Diagram of the skip connections in ResNets [25]

In this case, the *vanishing gradient problem* is partially solved, because the gradient at z_{k+q}, ∇z_{k+q} would be given by $\nabla z_k + \nabla \mathcal{G}_{k+q}$, where $\nabla \mathcal{G}_{k+q}$ represents the gradient of a multilayer perceptron.
Then, the gradient ∇z_{k+q} would not necessarily vanish whenever $||z_{k+i}|| < \epsilon << 1$ with $1 < i < q$, which happened in multilayer perceptrons.



$$\mathbb{R}^2 \longrightarrow \mathbb{R}^2 \longrightarrow \mathbb{R}^2 \longrightarrow \mathbb{R}^2 \longrightarrow \ ... \ \longrightarrow \mathbb{R}^2 \longrightarrow \mathbb{R}$$
$$z_0 = \vec{x} \xrightarrow{\vec{x}+f_0} z_1 \xrightarrow{z_1+f_1} z_2 \xrightarrow{z_2+f_2} z_3 \xrightarrow{z_3+f_3} \ ... \ \xrightarrow{z_{L-1}+f_{L-1}} z_L \xrightarrow{\psi} = \mathcal{F}_L(\vec{x})$$

Figure 8.8: Scheme of a ResNet with dimension of the input 2, dimension of the blocks 2 and a feedforward function given by f.

Chapter 9

Mean field optimal control formulation

Consider a ResNet given by

$$x_{l+1} = x_l + f(x_l; \theta_l) \tag{9.1}$$

and replaced by

$$x_{t+1} = x_t + f(x_t; \theta_t)\Delta t \tag{9.2}$$

where the requirement is that Δt is small, and goes to 0 as $T \to \infty$. [1]

Considering a continuous time-independent of time resolution, the equation driven the continuous limit forward dynamics of ResNets is

$$\lim_{\Delta t \to 0} \frac{x_{t+1} - x_t}{\Delta t} = \dot{x}_t = f(x_t; \theta_t) \tag{9.3}$$

Then the problem of supervised learning using que continuous limit of ResNet can be seen as a population risk minimization problem, where the data-label joint distribution $(x_0, y_0) \sim \mu^*$ on $\mathbb{R}^d \times \mathbb{R}^l$ is given:

$$\begin{aligned} \inf_{\theta \in L^\infty([0,T],\Theta)} J(\theta) \equiv \mathbb{E}_{\mu^*} \left[\Psi(x_T, y_0) + \int_0^T R(x_t, \theta_t) dt \right] \\ \dot{x}_t = f(x_t, \theta_t) \quad 0 \le t \le T \quad (x_0, y) \sim \mu^* \end{aligned} \tag{9.4}$$

[1]This may already happen without changing the structure of a ResNet, or should be implemented by-hand by multiplying each block by a suitable Δt, that can be absorbed by the function f.

where $R : \mathbb{R}^d \times \Theta \to \mathbb{R}$ is a regularizer, that can be introduced easily in the neural network, $\Psi : \mathbb{R}^d \times \mathbb{R}^l \to \mathbb{R}$ is the terminal loss function, and $f : \mathbb{R}^d \times \Theta \to \mathbb{R}^d$ the function describing the feed-forward dynamics.

Then we need to find θ that controls an entire distribution of inputs, and transports that distribution to their correspondent labels.
This is a mean-field problem.

This is related to the problem of minimizing a curve function in optimal control and calculus of variations.
However, in this case the control θ is over the entire population. The questions that arise is whether, under this formulation, it is possible to find necessary or sufficient conditions for optimality, by using the formulism of mean-field optimal control.

From this viewpoint, two independent results can be obtained. One related to the Pontryagin Maximum Principle, which is a necessary condition given by a local characterization of the optimal solution in terms of ODEs [3].

The other results is related to the Dynamic Programming Principle as inspired by Bellman, which is a necessary and sufficient condition and is given by the global characterization of the value function in terms of PDEs [5].

The Dynamic Programming Principle was later developed and completed by Crandall and Lions [11], and its relation with the PMP was pointed out through the method of characteristics [69].

9.1 Pontryagin Maximum Principle

(A) Suppose f is bounded; f, R are continuous in θ; f, R, Ψ continuously differentiable w.r.t. x, and that the distribution μ has bounded support in $\mathbb{R}^d \times \mathbb{R}^l$.
The solution θ^* such that the population risk minimization problem is solved would be given by

$$J(\theta^*) = \min_{\theta} J(\theta) \tag{9.5}$$

Under these assumptions, the mean-field Pontryagin Maximum Principle states that:

Theorem 9.1.1 (Mean-field PMP [64])**.** *Let (A) be satisfied, and $\theta^* \in L^\infty([0,T],\Theta)$ be a solution of the mean-field population risk minimization problem.*
Let $H : \mathbb{R}^d \times \mathbb{R}^d \times \Theta \to \mathbb{R}$ the Hamiltonian function, given by $H(x,p,\theta) = p \cdot f(x,\theta) - R(x,\theta)$

Then, there exists μ absolutely continuous stochastic processes $\mathbf{x}^*$, $\mathbf{p}^*$ such that θ^* maximizes the Hamiltonian function point-wise in time, i.e.

$$\theta_t^* \in \arg\max_{\theta\in\Theta} \mathbb{E}_{(x_0^*,y)\sim\mu^*} H(x_t^*, p_t^*, \theta) \qquad \textit{for each } t \in [0,T]$$

where $\{x_t^*, p_t^*\}$ satisfy the Hamiltonian dynamics described by

$$\dot{x}_t^* = \nabla_p H(x_t^*, p_t^*, \theta_t^*) = f(x_t^*, \theta_t^*) \qquad x_0^* = x_0 \tag{9.6}$$

$$\dot{p}_t^* = -\nabla_x H(x_t^*, p_t^*, \theta_t^*) \qquad p_T^* = -\nabla_x \Psi(x_T^*, y_0) \tag{9.7}$$

with $(x_0^*, y) \sim \mu^*$ and $p_T^* = -\nabla_x \Psi(X_T, y)$, and

$$\mathbb{E}_\mu H(x_t^*, p_t^*, \theta_t^*) = \max_{\theta\in\Theta} \mathbb{E}_\mu H(x_t^*, p_t^*, \theta), \quad a.e. t \in [0,T] \tag{9.8}$$

In this generalised Pontryagin Maximum Principle, no derivatives with respect to θ are needed, the parameter space Θ is general and the maximization condition is point-wise in time.
This is good for finding new training algorithms in a general setting.

The equations regarding the variables $\dot{x}_t^*$ and $\dot{p}_t^*$ are not new and have been obtained in a classical control approach, yet in this case the formalism is more general as it is dealing with probabilities distributions.
However, the third equation arises from this mean-field optimal control formulation. This equation is referring to the whole distribution instead of pointwisely.

9.1.1 From mean-field to empirical PMP

A specific case of population risk minimization would be the *sampled optimal control problem*, given by

$$\min_{\theta \in L^\infty([0,T],\Theta)} J_N(\theta) \equiv \frac{1}{N} \sum_{i=1}^{N} \left[\Psi(x_T^i, y_0^i) + \int_0^T R(x_t^i, \theta_t) dt \right] \tag{9.9}$$

Indeed, the problem of deep learning in practice is a problem of sampled optimal control, as the number of data is finite.

The gap between the sampled optimal control problem and the population risk minimization problem is closely related to generalisation. The problem deals with how a sampled problem related to the entire distribution, and how big must be the number of samples N such that the solution to the sampled optimal control problem is close to the solution of the analogue population risk minimization problem.

In order to mantain the previous results, the empirical distribution can be defined as

$$\mu_N \equiv \frac{1}{N} \sum_{i=1}^{N} \delta_{(x_0^i, y_0^i)} \tag{9.10}$$

doing this replacement, the mean-field PMP becomes the classical PMP [64] [**?**].

In order to further study this gap between the population risk and the empirical risk minimization problems, E, Han and Li propose a strategy which compares the mean-field PMP to the sampled PMP.

Suppose a solution of the mean-field PMP such that

$$F(\theta^*)_t \equiv \mathbb{E} \nabla_\theta H(x_t^{\theta^*}, p_t^{\theta^*}, \theta_t^*) = 0 \tag{9.11}$$

while the solution of the sampled problem is

$$F_N(\theta^N)_t \equiv \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta H(x_t^{\theta^N,i}, p_t^{\theta^N,i}, \theta_i^N) = 0 \tag{9.12}$$

The relation with $F_N(\theta^*)_t$ and $F(\theta^*)_t$ can be obtained using a contraction mapping

$$G_N(\theta) \equiv \theta - DF_N(\theta^*)^{-1}F_N(\theta) \tag{9.13}$$

where D is a Frechet derivative.

Definition 9.1.1. *Let $\rho > 0$ and $x \in U$. $S_\rho(x) \equiv \{y \in U \,:\, \|x-y\| \leq \rho\}$.*

A mapping F is stable on $S_\rho(x)$ if there exists a constant $K_\rho > 0$ such that, for all $y, z \in S_\rho(x)$,

$$\|y - z\| \leq K_\rho \|F(y) - F(z)\| \tag{9.14}$$

The requirement of a map to be stable is very common, and can be recasted as the supposition of the optimal solution x^* is non-singular, and that the inverse map is Lipschitz.

Theorem 9.1.2 (Neighbouring solution for sampled PMP [64])**.** *Let θ^* be a solution $F = 0$, whih is stable on $S_\rho(\theta^*)$ for some $\rho > 0$.*

There exists $s_0, C, K_1, K_2 \in \mathbb{R}_+$, $\rho_1 < \rho$ and a random variable $\theta^N \in S_{\rho_1}(\theta^*) \subset L^\infty(]0,T],\Theta)$ such that

$$\mu\left[\|\theta - \theta^N\|_{L^\infty} \geq C_S\right] \leq 4\exp\left\{-\frac{Ns^2}{K_1 + K_2 s}\right\}, \quad s \in (0, s_0], \tag{9.15}$$

$$\mu[F_N(\theta^N) \leq 0] \leq 4\exp\left\{-\frac{Ns_0^2}{K_1 + K_2 s_0}\right\} \tag{9.16}$$

Where $\theta^N \to \theta^*$ and $F_N(\theta^N) \to 0$ in probability is considered in our case.

Indeed, if second-order regularity conditions are supposed for the hamiltonian, then the sequence θ^N previously seen minimizes the problem with high probability [64]. That is, there exists K_1, K_2 constants such that

$$\mathbb{P}[|J(\theta^N) - J(\theta^*)| \geq s] \leq 4\exp\left\{-\frac{Ns^2}{K_1 + K_2 s}\right\}, \quad s \in (0, s_0] \tag{9.17}$$

This can be seen as an a priori error estimate for the sampled risk minimization problem.

9.1.2 PMP as sufficient condition

If uniqueness and existence is guaranteed, then the necessary condition given by the mean-field PMP becomes sufficient.

Indeed, imposing more conditions on f, R and Ψ, there is small-time uniqueness.

(A') suppose f bounded, f, R, Ψ twice continuously differentiable with respect to x and θ. f, R, Ψ have bounded and Lipschist partial derivatives.

Theorem 9.1.3 (Small-time uniqueness [64]). *Let (A') hold. Suppose $H(x, p, \theta)$ strongly concave in θ, uniformly in $x, p \in \mathbb{R}^d$. i.e. $H(x, p, \theta) + \lambda_0 I \preceq 0$ for some $\lambda_0 > 0$. Then, for sufficiently small T, the solution of the PMP is unique.*

These strong concavity requirements is imposed on the hamiltonian, and not on the loss function J, that can even be non-convex and the condition still satisfied. It is in general difficult to derive whether the conditions hold in a practical setting.

Stronger results are needed, such that the characterization of the minimum is applicable to larger times. In order to do that, the Dynamic Programming Principle needs to be introduced.

9.2 Dynamic Programming Principle

Another perspetive introduced by the mean-field optimal control approach is to use a mean-field Dynamic Programming Principle.

The idea is to understand the joint distribution of (x_t, y_0) as a state variable in the Wasserstein space [61]. It is also needed to define a value function $v(t, \mu)$ such that it is the optimal objective value of $\mathcal{P}(t, \mu)$ [64].

Let ω be the concatenation of (x, y), $(\Omega, \mathcal{F}, \mathbb{P})$ be the probability space, $L^2(\mathcal{F}; \mathbb{R}^{d+l})$ the space of square-integrable random variables with L^2 metric, and $\mathcal{P}_2(\mathbb{R}^{d+l})$ the space of square integrable measures with $2-$Wasserstein metric.

Indeed,

$$W \in L^2(\mathcal{F};\mathbb{R}^{d+l}) \iff \mathbb{P}_W \in \mathcal{P}_2(\mathbb{R}^{d+l}) \tag{9.18}$$

Theorem 9.2.1 (Mean-field HJB equation [64]). *Let v be the unique viscosity solution of que Hamilton-Jacobi-Bellman equation*

$$\frac{\partial}{\partial t}v(t,\mu) + \inf_{\theta\in\Theta}\langle \partial_\mu v(t,\mu)(\cdot)\cdot f(\cdot,\theta) + R(\cdot,\theta),\mu\rangle = 0$$
$$v(T,\mu) = \langle \Phi,\mu\rangle \quad \textit{where } \langle U,\mu\rangle = \int U(x,y)d_\mu(x,y))$$

If the minimum is attained,

$$\theta_t^* = \underset{\theta\in Theta}{\arg\min}\langle \partial_\mu v(t,\mu_t^*)(\cdot)\cdot f(\cdot,\theta) + R(\cdot,\theta),\mu_t^*\rangle \quad t\in[0,T]$$

is a set of optimal trainable parameters, where μ_t^* denotes the law of (x_t^*,y).

The development and notation of the mean-field Dynamic Programming Principle is technical and tedious, and out of the scope of this work. In [64], the existence and uniqueness of solutions using these approach is proven, and the results are stronger than the ones of PMP.

Part IV

Applications

Chapter 10

Classification with Neural Networks

Definition 10.0.1 (Classification problem)**.** *A classification problem is a subproblem of supervised learning in which the target variable y is discrete.*

Given a classification problem, with a training set $\{(x_i, y_i)\}_{i=1}^{k}$, such that $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$.
Suppose $\mathcal{X} \subseteq \mathbb{R}^n$. In practice, the input space is normalized such that $\mathcal{X} = [-1, 1]^n$ or $[0, 1]^n$.
Let $\mathcal{Y} = \{0, 1, ..., L-1\}$ with L the number of classes.
The classification problem consists on finding an approximation of $F : \mathcal{X} \to \mathcal{Y}$ such that, if no noise is considered, $F(x) = y$, by using neural networks with represented functions $\hat{F}(;\theta)$.

Proposition 10.0.1. *Consider noisless data of more than one class Then it possible to find $\tilde{\mathcal{X}}$ such that $\hat{F}(x) \neq y = F(y)$ $\forall x \in \tilde{\mathcal{X}}$.*

Proof. Since F maps from a a compact set $\mathcal{X}$ to $\{0, 1, ..., L-1\}$, except for the case that y_i is constant, the function F is not continuous.
However, the function $\hat{F}(;\theta)$ must be continuous, since it is given by the composition of continuous functions.
Then there must be a subset of the intput space, $\tilde{\mathcal{X}} \subset \mathcal{X}$, such that $\hat{F}(x) \neq F(x)$ $\forall x \in \tilde{\mathcal{X}}$. ■

Indeed, the problem of classification is not well posed. Such a function F is not expected to exist, since one expects to find data x_i that cannot be classified.

Imagine a problem of binary classification, in which we want to classify images of dogs and cats. Suppose that the input space is the space of 64×64 pixel images, and the corresponding label to each image x_i is either 0 (a dog) or 1 (a cat).
Then, it is easy to see that an input image x_i corresponding to an all-black pixels image, cannot be classified, and thus $F(x)$, for x an all-black pixels image, would not be well-defined.

This problem is easily solved by introducing a new label ξ such that $\xi \in (0,1)$. Then, $\xi_i = F(x_i)$ represents the probability of x_i of being of class 1.
Indeed, the function F is

$$F : \mathcal{X} \to [0,1]^L \tag{10.1}$$
$$x \mapsto (\xi^0(x), ..., \xi^{L-1}(x)) \tag{10.2}$$

where $\sum_{j=0}^{L-1} \xi^j(x) = 1$, and such that $\xi^l(x)$ represents the probability that x is of class l.

The training data $\{(x_i, y_i\}$ is such that $y_i = ((\xi^0(x_i), ..., \xi^{L-1}(x_i)))$ with $\xi^l(x_i) = 1$ for a given l, and $\xi^j(x_i) = 0$ for $j \neq l$.In compact form, $\xi^j = \delta_{jl}$ the kroenecker delta.
Some training data could be used such that this does not happen, but this is not implemented in practice.

This reformulation of the classification problem allow for introducing noisy data, and also makes the problem well-posed, in the sense that a function $\hat{F}$ could in principle be the same as F.

Proposition 10.0.2. *A binary classification problem is equivalent to a transportation problem plus a margin classifier problem.*

10.1 Dimensionality

The dimension of y can be arbitrarily chosen by specifying the number of components (neurons) per layer.
The question of which dimension would give the best model arises.

In practice, it is important to be able to change the dimensions of the neural network for different layers.
Neural Networks usually have access to a limited number of types of nonlinear coordinate transformaions (tanh, sigmoid, ReLU), which limits the ability of the network to separate the wide variety of manifolds that exist.
Locally Linear Embeddings [51] solve this problem, as k-nearest neighbours is an extremely flexible type of nonlinearity. [23]

However, some results can be given when using Neural Networks as approximators for, in this case, classification.

Suppose the problem of binary classification. The input space is $[-1,1]^2$,¡. The data labelled as 1 is $\mathcal{X}_1$ and the labelled as 0 is $\mathcal{X}_2$, such that

$$\mathcal{X}_0 = \{x \,|\, d(x,0) < a\}$$
$$\mathcal{X}_1 = \{x \,|\, b < d(x,0) < 1\}$$

with $a < b$.

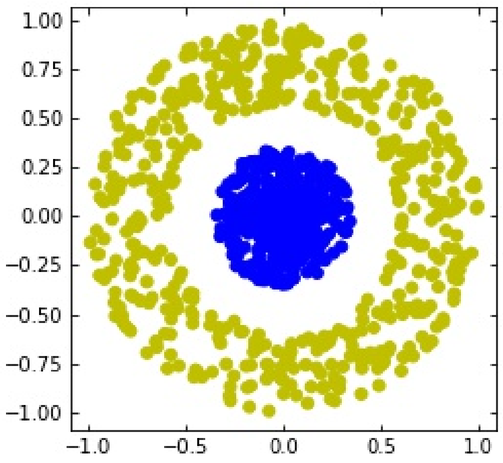


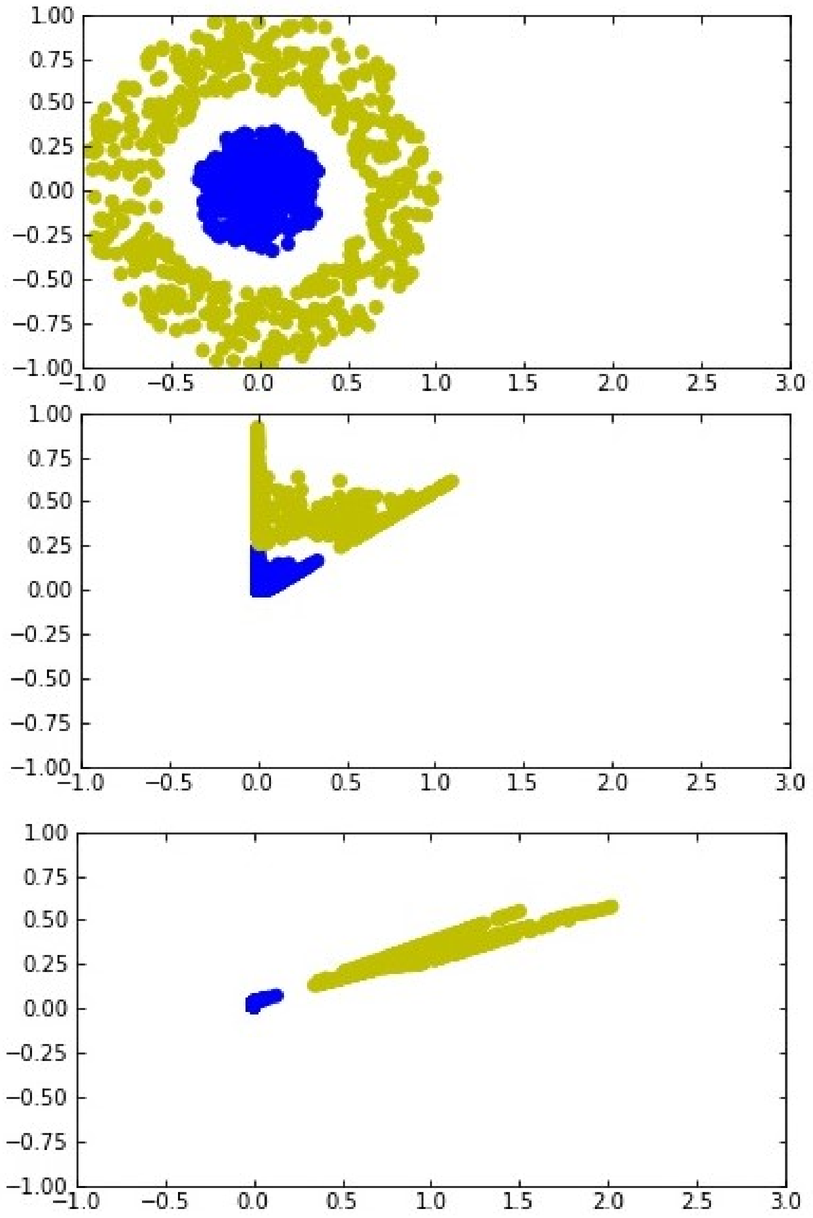
Figure 10.1: Example of data distribution for binary classification, with 1000 samples.

Then, if a *tanh* activation is used, this problem can be solved using a neural network with 3 neurons per layer.
The classification problem is equivalent to separating the datasets $\mathcal{X}_0$ and $\mathcal{X}_1$ as transformed by transformations given by the neural networks, that can be homeomorphisms.

If the transformations were not homeomorphisms, then it would be possible that some points in $\mathcal{X}_1$ got mixed with points in $\mathcal{X}_2$. If using homeomorphic transformations, then in order to linearly separate $\mathcal{X}_0$ and $\mathcal{X}_1$ three dimensions would be needed.

It is clear that if 2-dimensional homeomorphic transformations were used, using notions from knot theory, the two sets $\mathcal{X}_0, \mathcal{X}_1$ would not be separable using homeomorphisms.

In a three-dimensional space, however, it is possible to make $\mathcal{X}_0$ and $\mathcal{X}_1$ linearly separable. In order to see this, it is enough considering a homeomorphism that pushes $\mathcal{X}_1$ away in this new dimension.



Formally,

Proposition 10.1.1. *Consider two manifolds $\mathcal{X}_0$, $\mathcal{X}_1$, such that every point in $\mathcal{X}_0$ is classified as* 0*, and every point in $\mathcal{X}_1$ as* 1*.*
Let $\mathcal{X}_0, \mathcal{X}_1 \subseteq \mathbb{R}^N$ compact subsets, and $d(\mathcal{X}_0, \mathcal{X}_1) > \epsilon > 0$, where the Haussdorff distance is used.
Then, it is possible to make $\mathcal{X}_0$ and $\mathcal{X}_1$ linearly separable by using an homeomorphism of dimension at least $N+1$.

Proof. Consider a continuous map $\Phi : \mathbb{R}^{N+1} \to \mathbb{R}^{N+1}$, that defines a homeomorphism. The $N-$dimensional input can be padded to $N+1$ dimensions such that $(\vec{x}) \mapsto (\vec{x}, 0)$. There exist Φ such that

$$\Phi((\vec{x}, 0)) \mapsto (\vec{x}, \omega) \ \forall \vec{x} \in \mathcal{X}_0$$

$$\text{and } \Phi((\vec{x}, 0)) \mapsto (\vec{x}, 0) \ \forall \vec{x} \in \mathcal{X}_1$$

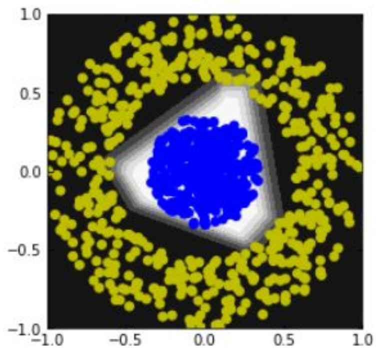
where this is possible due to the fact that $d(\mathcal{X}_0, \mathcal{X}_1) > \epsilon > 0$, and so continuity can be guaranteed.
Let $\omega > 0$. Then the manifolds, as transformed by the neural network, would be linearly separable by using an hyperplane perpendicular to the vector $(\cdot, 1)$.

■

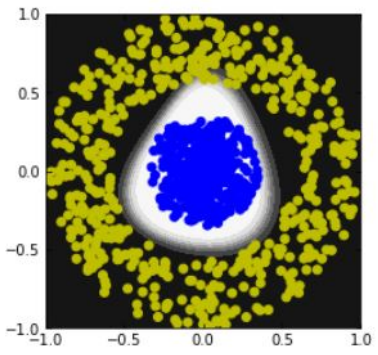
The problem with this approach is that even if an homeomorphism exists such that the two manifolds are linearly separable, it is not guaranteed that this homeomorphism can be represented or approximated by the neural network in the case of multilayer perceptrons, when fixing the number of neurons per layer.

Moreover, the hypothesis that data of different classes is in different manifolds is too strong. In practice, since there is noise, it would be possible that yellow and blue points are arbitrarily close.

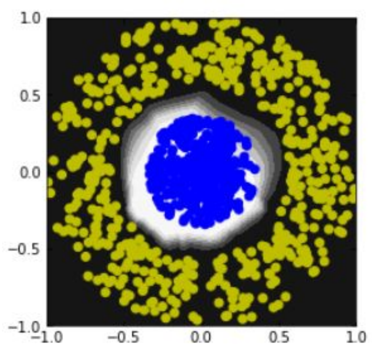
And, given the fact that Deep Learning deals with datapoints, and not manifolds, it is always possible to find a finite-measure cut such that, if the input is $N-$ *dimensional*, the points can be separated using an homeomorphism in N dimensions.



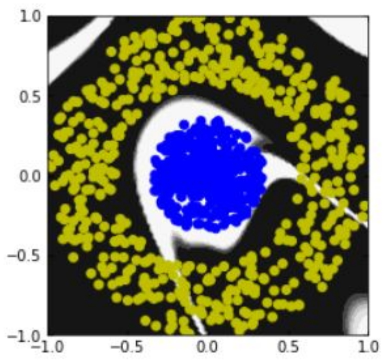
(a) 4-hidden layer multilayer ReLU perceptron



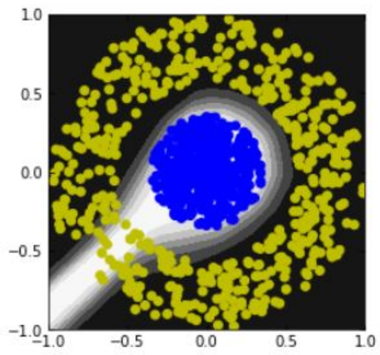
(b) 4-hidden layer multilayer tanh perceptron



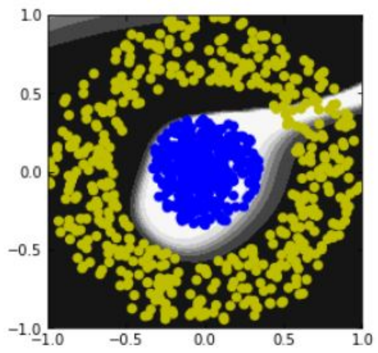
(c) 20-hidden blocks ReLU ResNet



(d) 20-hidden blocks tanh ResNet

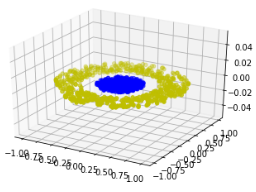


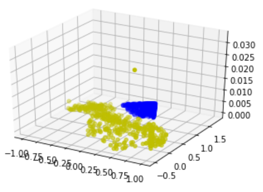
(e) 20-hidden blocks modified ReLU ResNet

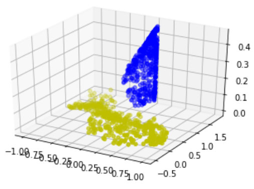


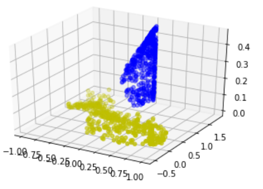
(f) 20-hidden bloks modified tanh ResNet

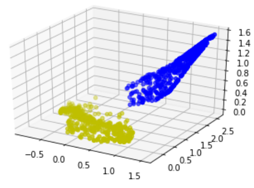
Figure 10.2: Level sets of binary classification with different architectures after training.


(a) $T = 0$, $\tau = 0$


(b) $T = 0$, $\tau = 1$


(c) $T = 0$, $\tau = 2$


(d) $T = 0$, $\tau = 3$


(e) $T = 1$, $\tau = 0$

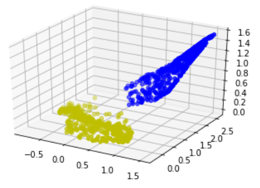
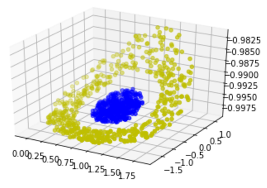
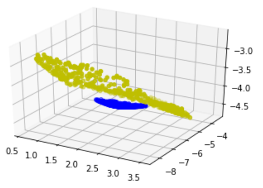
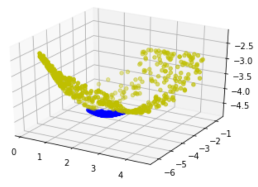
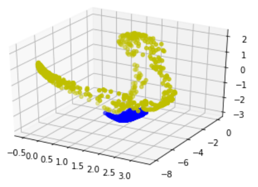

(f) $T = 1$, $\tau = 1$

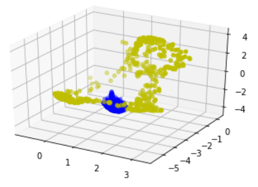
Figure 10.3: Level sets with a ring distribution using a multilayer perceptron with tanh activation, 2 neurons per layer and 4 hidden layers.


(a) $T = 1$, $\tau = 2$


(b) $T = 1$, $\tau = 3$


(c) $T = 2$, $\tau = 0$


(d) $T = 2$, $\tau = 1$


(e) $T = 2$, $\tau = 2$

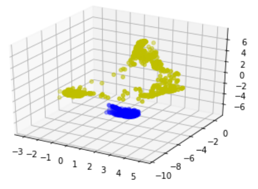

(f) $T = 2$, $\tau = 3$

Figure 10.4: Level sets with a ring distribution using a multilayer perceptron with tanh activation, 2 neurons per layer and 4 hidden layers.

Part V

Discussion

Chapter 11

Conclusions and further work

The theory of Deep Learning as Dynamical Systems is mostly effective when dealing with the idealization of ResNets, although it provides more intuition when dealing with multilayer perceptrons.
This idealization, although easily implementable in practice, does not explain the effectivity of ResNets when the residual part is not small enough.

The visualization of multilayer perceptrons as a discrete dynamics interesting by means of visualization, but it is difficult to obtain strong results in a general framework. Some interesting directions for obtaining results for the approximation using multilayer perceptrons are proposed [8] [29] but they are in general weak, and do not explain the fact that Deep Learning with multilayer perceptrons is efficient in practice. The results in terms of expressivity or capacity are necessary for approximation, but not sufficient. And even if approximation is guaranteed, optimization may not be possible in general.

Some authors propose that multilayer perceptrons first focus on lower frequencies and then on higher ones, and so a finite-time training would have some kind of instrinsic low-pass filter, helping with generalization [66].

Regarding the idealization of ResNets as discretizations of ODEs, new architectures are proposed that can be understood as the continuous limit of ResNet, and are trained using the adjoint sensitivity method of Pontryagin instead of backpropagation [18] [31] [43]. Their implementation is not yet understood, nor its applications, but they promise to be memory efficient, and easier to study from an analytical perspective.

One of the main unkowns of Deep Learning is why are some functions more difficult to approximate than others.

The Dynamical Systems perspective provides a solid background to study this, but results have been obtained on a the more general direction that (almost) any function can be approximated, without specifying the amount of training time needed nor the initialization, that play a huge role in the efficiency of this methods.

There is a need to study the neural networks from a complex systems perspective, and to obtain results in non-convex loss landscapes, which can be easily found in practice.

An interesting tool to study the complexity of the datasets is topological data analysis [9] [63] [21]. And, since classification can be recasted as a problem of transport, also transport theory [57] [56].

The theory presented is a fundamental tool for understanding the symbiosis between Deep Learning and Dynamical systems. Deep Learning has shown remarkable results for applications in high-dimensional dynamical systems [7] [20].
Although most of the approaches are heuristical, some authors such as Grüne develop specific approximation results for each setting.

Other architecture, for example based in convolutions, have not been explained or mentioned. They require a very specific study [42], and the explanation of Deep Learning as Dynamical System point out a connection between convolutional networks an PDEs [53].

Some authors point out that the good results when using Deep Learning in Dynamical System settings is due to the fact that there is some intrinsic and fundamental relation between them.
In fact, Deep Learning has outstanding results in quite different tasks. This work does not provide any fundamental relation between Deep Learning and Dynamical Systems, but rather a framework in which to study approximation and control of neural networks. The time that defines the dynamical system in the neural networks is related to the number of layer, and so it is a numerical parameter with no physical meaning nor relation with the input data, in principle.

The problems regarding the numerical implementation are avoided. Automatic differenciation is not well-understood from a mathematical perspective, and the numerical errors when computing the succesive intermediate activations may play a very important role, that it is not taken into consideration.

The theory is as yet the most succesful theory of Deep Learning, as it provides a solid and simple framework from which to study a wide range of architectures and phenomena, as well as proposing new ones.

This approach would also serve as an inspiration to study other ML tasks such as unsupervised learning and reinforcement learning, that seem to be closely related to classical problems in optimal control and dynamical systems, yet the mathematical formulation has not been developed.

Part VI

Appendices

Appendix A

Gradient descent

A.1 Simplest linear model

We consider an unkown dynamic system that is stimulated by an input vector $\mathbf{X}(i) = [x_1(i), x_2(i), ..., x_M(i)]^T$, where i is a time index $i = 1, 2, ..., n$ denoting the time at which the stimulus is applied to the system.

The external behaviour of the system is described by the set of pairs *input-output* given by

$$\mathcal{F} : \{\mathbf{X}(i),\ d(i) \mid i = 1, 2, ..., n, ...\} \tag{A.1}$$

where $d(i)$ is the output of the dynamic system as excited by $\mathbf{X}(i)$.

The problem is to consider $M > 1$ and $\dim(d(i)) = 1$. Moreover, we want to approximate the dynamical system by a *single linear neuron*, that is, a *neural network* with no hidden layers, and an activation function given by $\psi(x) = x$.

Thus, the output of our model would be given by

$$y(i) = \sum_{k=1}^{M} w_k(i) x_k(i) \tag{A.2}$$

where in this case we have considered the bias $b(i) = 0$. [1]

[1] In this simplified case, the bias is not taken into consideration for simplicity. The main results of this section could be later generalised to the *biased* case.

Using a more compact notation, we have that

$$y(i) = \mathbf{X}^T(i)\mathbf{w}(i) \tag{A.3}$$

where

$$\mathbf{w}(i) = [w_1(i), w_2(i), ..., w_M(i)]^T$$

The unkown dynamical system with output $d(i)$ is now simulated by a simple *neural network* with linear activation and output $y(i)$. We may define the error as

$$e(i) = d(i) - y(i) \tag{A.4}$$

The problem is recasted to finding the weights $\mathbf{w}(i)$ such that the error e_i (in absolute value) is minimized.

This problem is closely related to that of optimization, and so a review of *unconstrained optimiztion methods* is hereafter introduced.

A.1.1 Unconstrained optimization

Consider $\xi(\mathbf{w})$ a cost function

$$\begin{aligned} \xi : \mathbb{R}^M &\to \mathbb{R} \\ \mathbf{w} &\mapsto \xi(\mathbf{w}) \end{aligned} \tag{A.5}$$

Being $\xi(\cdot)$ a measure of how close is a set of weights $\mathbf{w}$ to the optimal case.
The cost function $\xi(\cdot)$ is assumed to be *continuously differentiable* with respect to $\mathbf{w}$.
We want to find an optimal solution $\mathbf{w}^*$, that is given by the fact that

$$\xi(\mathbf{w}^*) \leq \xi(\mathbf{w}) \quad \forall \mathbf{w} \in \mathbb{W} \tag{A.6}$$

We assume that $\mathbb{W} = \mathbb{R}^M$.

Note that $\mathbf{w}^*$ may be not unique.

Problem A.1.1. *Unconstrained optimization:*

Minimize $\xi(\mathbf{w})$ with respect to $\mathbf{w}$

A.6 implies that

$$\nabla\xi(\mathbf{w}^*) = \mathbf{0} \tag{A.7}$$

Where ∇ is given by

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, ..., \frac{\partial}{\partial w_M}\right]^T \tag{A.8}$$

And so, if applied to $\xi(w)$ gives

$$\nabla\xi(\mathbf{w}) = \left[\frac{\partial xi}{\partial w_1}, \frac{\partial xi}{\partial w_2}, ..., \frac{\partial \xi}{\partial w_M}\right]^T \tag{A.9}$$

Proof. If $\nabla\xi(\mathbf{w}^*) = a \neq 0$, then we define $p = -\nabla\xi(\mathbf{w}^*)$ such that $p^T\nabla\xi(\mathbf{w}^*) = -\|\nabla\xi(\mathbf{w}^*)\|^2 < 0$.

Because $\xi(\cdot)$ is continuously differentiable, there would be a scalar T such that $p^T\nabla\xi(\mathbf{w}^* + tp) < 0 \; \forall t < T$.

Using Taylor's theorem, we find a direction (p) from which $\xi(\mathbf{w}^*)$ decreases, so $\mathbf{w}^*$ would not be a minimum.

■

A.1.2 Training algorithms

Problem A.1.2. *Training algorithm*

We aim to find an algorithm such that, by starting with an initial guess for $\mathbf{w}(0)$, it generates a sequence $\{\mathbf{w}(1), \mathbf{w}(2), ...\}$ such that

$$\xi(\mathbf{w}(n+1)) < \xi(\mathbf{w}(n)) \quad \forall n \tag{A.10}$$

A *training algorithm* is expected to eventually reach the minimum $\mathbf{w}^*$, but that may not happen in practice.

A.1.2.1 Gradient descent

The *gradient descent method* (or *steepest descent method* updates the weights $\mathbf{w}$ in the opposite direction to the gradient vector $\nabla\xi(\mathbf{w})$.

The method is formally described as

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \nu\mathbf{g}(n) \tag{A.11}$$

where $\mathbf{g} = \nabla\xi(\mathbf{w})$, and $\nu > 0$ is the *stepsize.*

Proposition A.1.1. *The steepest-descent algorithm is a good candidate for a training algorithm.*

Proof. Idea of the proof Doing the Taylor approximation around $\xi(\mathbf{w}(n))$,

$$\xi(\mathbf{w}(n+1)) \approx \xi(\mathbf{w}(n)) + \mathbf{g}^T(n)\Delta\mathbf{w}(n) = \xi(\mathbf{w}(n)) + \mathbf{g}^T(n)\left(\mathbf{w}(n+1) - \mathbf{w}(n)\right) =$$

$$= \xi(\mathbf{w}(n)) - \nu\mathbf{g}^T(n)\mathbf{g}(n) = \xi(\mathbf{w}(n)) - \nu\|\mathbf{g}(n)\|^2 < \xi(\mathbf{w}(n)) \text{ for } \|\mathbf{g}(n)\| \neq 0$$

■

Where we have used that ν is small. However, the number of iterations T in order to achieve $\mathbf{w}^*$ goes as $O(1/\nu)$.

We may have three different regimes depending on the value of ν.

1. ν is small. The trajectory $\{\mathbf{w}(i)\}$ follows a smooth path in the phase space.
2. ν is large. The trajectory of the weights follows an oscillatory path.
3. $\nu > \nu_0$. The algorithm becomes unstable (i.e. the solution diverges).

The reason for the existence of those three different regimes is that we can only guarantee that $\xi(\mathbf{w}(n+1)) = \mathbf{g}^T(n)\Delta\mathbf{w}(n) + O\left((\Delta\mathbf{w}(n))^2\right) < \xi(\mathbf{w}(n))$ when $\Delta\mathbf{w}(n)$ is sufficiently small.

Since we do not have an expression for $\Delta\mathbf{w}(n)$ in the general case, it is not trivial to choose the value ν such that it converges to a good solution in finite computing time.

This needs to be done case-by-case, tuning the value of ν such that it is smaller than a critical value.

A.1.2.2 Newton's method

There are more elaborated methods than *steepest optimization method.* For example, it is convenient to look at the second order Taylor approximation rather than at the first-order one; i.e. one has

$$\begin{aligned}\Delta\xi(\mathbf{w}(n)) = \xi(\mathbf{w}(n+1) - \xi(\mathbf{w}(n)) \approx \\ \approx \mathbf{g}^T(n)\Delta\mathbf{w}(n) + \frac{1}{2}\delta\mathbf{w}^T(n)\mathbf{H}(n)\Delta\mathbf{w}(n)\end{aligned} \tag{A.12}$$

where

$$\mathbf{H} = \nabla^2\xi(\mathbf{w}) =$$

$$\begin{bmatrix} \frac{\partial^2\xi}{\partial w_1^2} & \frac{\partial\xi}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2\xi}{\partial w_1 \partial w_M} \\ \frac{\partial^2\xi}{\partial w_2 \partial w_1} & \frac{\partial^2\xi}{\partial w_2^2} & \cdots & \frac{\partial^2\xi}{\partial w_2 \partial w_M} \\ \cdots & \cdots & \cdots & \\ \frac{\partial^2\xi}{\partial w_M \partial w_1} & \frac{\partial^2\xi}{\partial w_M \partial w_2} & \cdots & \frac{\partial^2\xi}{\partial w_M^2} \end{bmatrix}$$

We can differenciate the second-order Taylor approximation by $\mathbf{w}$, and minimize $\Delta\xi(\mathbf{w})$,

$$\mathbf{g} + \mathbf{H}(n)\Delta\mathbf{w}(n) = \mathbf{0} \tag{A.13}$$

therefore a candidate for a training algorithm would be given by

$$\begin{aligned}\mathbf{w}(n+1) = \mathbf{w}(n) + \Delta\mathbf{w}(n) = \\ = \mathbf{w}(n) - \mathbf{H}^{-1}(n)\mathbf{g}(n)\end{aligned} \tag{A.14}$$

This is known as the *Newton Method.*

In this case it is required that $\xi(\mathbf{w})$ is twice continuously differentiable, which is not guaranteed in general. We also need that $\mathbf{H}$ is positive definite, unless we do some modifications [47] [33] . Moreover, the computational complexity is greater of that of the *steepest descent* [6] . The *Gauss-Newton* method reduces the complexity of this algorithm [17] [13], although most *training algorithms* used with more complex *neural networks* are based on the *steepest method*.

In general, *Newton's method* converges quickly asymptotically and does not exhibit zigzagging (that *steepest method* exhibits for certain values α). Although *Newton's method* outperforms the *steepest method* [22], the scope of this work is to make an overview of the methods used in *Deep Learning* using standard libraries such as *Tensorflow*.

Some authors have studied the implementation of *Newton's method*-inspired methods to more complex *neural networks* [26] [10] [40], but they are not widely used in practice.

A.2 Algorithms

A.2.1 Stochastic Gradient Descent

The number of samples so as to compute the cost function $\xi(\mathbf{w})$ can be too large. The *stochastic gradient descent*, or SGD is based on the steepest descent, but in this case the cost function $\xi(\mathbf{w})$ is not computed by looking at all the samples, but just to one of them. [50]

That is, in the *deterministic gradient descent* (steepest descent) the cost function $\xi(\mathbf{w})$ is a function of the error

$$\xi(\mathbf{w}) = \frac{1}{M}\sum_{j=1}^{M} \|y(j;\mathbf{w}) - d(j)\|_{\mathcal{L}} \tag{A.15}$$

where $\{x_j, d_j\}_{j=1}^{M}$ is the dataset, and $\|\cdot\|_{\mathcal{L}}$ is a distance defined by a loss function $\mathcal{L}$ and $y(\cdot;\mathbf{w})$ is the function represented by the neural network (in this case, given by A.2) and with weights $\mathbf{w}$.

In the SGC case, the gradient descent uses an approximation of ξ, $\hat{\xi}$, such that

$$\hat{\xi}(j;\mathbf{w}) = \|y(j;\mathbf{w}) - d(j)\|_{\mathcal{L}} \tag{A.16}$$

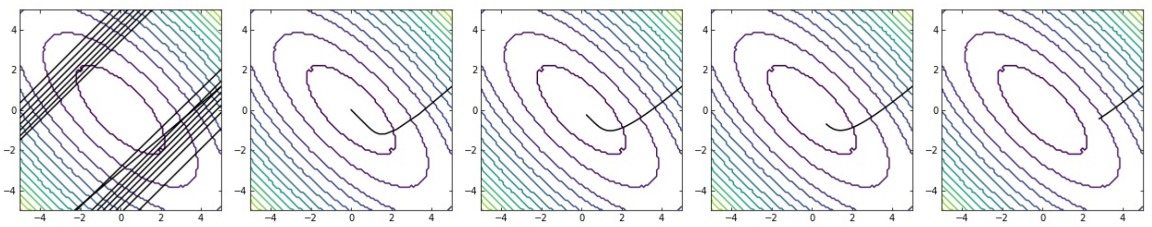
The implementation is the same as in the case of *steepest descent*, but in this case it is more efficient since it is easier to compute $\hat{\xi}$ than ξ, in the sense that the number of operations required to compute ξ, $C(\xi)$ is in general related to the cost of computing $C(\hat{\xi})$ such that

$$C(\xi) = C(\hat{\xi})\mathcal{O}(M) \tag{A.17}$$

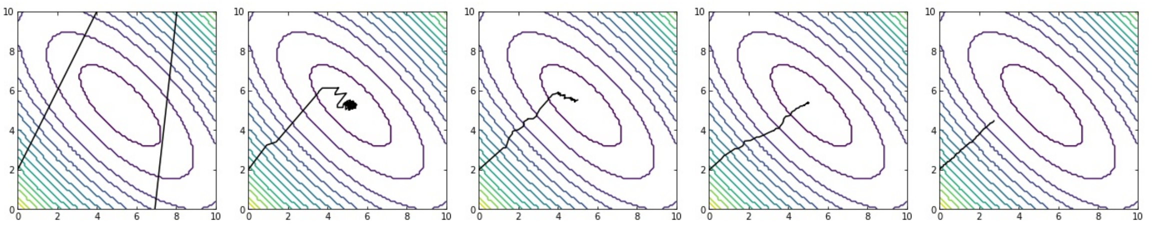
with M the number of samples.

The SGD can be trivially generalised such that the approximation of the error $\hat{\xi}$ uses more than one data $\{x_j, d_j\}$. In this case, the algorithm is called *mini-batch gradient descent*.

In this case,



(a) $T = 1$, $\tau = 0$ (b) $T = 1$, $\tau = 1$ (c) $T = 1$, $\tau = 2$ (d) $T = 1$, $\tau = 3$ (e) $T = 2$, $\tau = 0$



(f) $T = 1$, $\tau = 0$ (g) $T = 1$, $\tau = 1$ (h) $T = 1$, $\tau = 2$ (i) $T = 1$, $\tau = 3$ (j) $T = 2$, $\tau = 0$

Figure A.1: Level sets with a ring distribution using a multilayer perceptron with ReLU activation, 2 neurons per layer and 4 hidden layers.

$$\hat{\xi}(j_1, j_2, ...j_p; \mathbf{w}) = \sum_{i=1}^{p} \|y(j_i; \mathbf{w}) - d(j_i)\|_{\mathcal{L}} \tag{A.18}$$

Trivially, in case $p = 1$ this is equivalent as the SGD, and for $p = M$ it is the *steepest descent*. It is not trivial to choose a value of p. [49]

Since the number of datapoints use to be huge in real applications, SGD-based or *mini-batch learning* algorithms are widely used. [52] [35]

In practice, the most succesful algorithms are the adaptive ones, in which the parameter ν is not constant, but adaptive.

Some of the most used ones are AdaGrad and RMSProp, and ADAM algorithm, which combines both. [30][46][**?**]

Appendix B

Training neural networks

A *multilayer perceptron* is the same as a *totally connected neural network.*

From an information theory perspective, we can understand the training of a *multilayer perceptron* by defining two different signals,

- The *function signals*, or *activations*, are the signals originated from an *input signal* (the input data), and are obtained by the composition of functions as

  $$h(\hat{m}, k)(x) = \left(A_m^k z^k + b_m^k\right)$$

  where k indicates the layer, and m the position of that neuron in the layer, and

  $$\begin{cases} z^{i+1} = \sigma(A^i z^i + b^i) & \text{for } i = 0, ..., k-1 \\ z^0 = \vec{x}_j \in \mathbb{R}^d \end{cases}$$

- The *Error signals* are the signals that originate at the output neuron of the network, propagate backwards (layer by layer) and help tune the weights of the neurons so as to minimize the *loss function.* In this section it will be discussed how can we describe / define this signal.

In order to train *multilayer perceptrons* we need that each neuron stores information both of its weights and of some information that we need in order to update its weights.

The idea of training *multilayer perceptrons* is based on *gradient descent*, as seen in previous sections. However, it is not trivial to know which of the output error is consecuence of having a bad weight at a specific hidden layer. That is, we have a problem, the *credit-assignment problem.*

B.1 Batch learning and on-line learning

We may have data used to train our neural network,

$$\mathcal{X} = \{\mathbf{x}_i\ \mathbf{y}_i\}_{i=1}^{K} \tag{B.1}$$

Such that the *loss* is computed by

$$\xi = \ \|\hat{f}_\theta(\mathbf{x}) - \mathbf{y}\|_{\mathcal{L}} \tag{B.2}$$

where $\|\ \ \|_{\mathcal{L}}$ is a distance defined by a given *loss function.*

As in a generalisation of the *steepest descent* algorithm, one can take into account the error of all the data in $\mathcal{X}$ to as to update the weight in a gradient-descent manner. If this is the case, we say that we are using *batch learning.*

In the case that we use

$$\xi(i) = \frac{1}{2}e^2(i) \quad e(i) = \hat{f}_{\theta)}(\mathbf{x_i}) - \mathbf{y}_i \tag{B.3}$$

That is, we use the L^2 norm as the distance (except for the constant 1/2). In order to keep the notation simple, and to make more explicit the relation betwen training *multilayer perceptrons* and a simple linear model, we can define

$$d(i) \equiv \hat{f}_\theta(\mathbf{x}_i) \tag{B.4}$$

In the last layer (with C neurons) we can assign an error to each neuron as

$$\xi_j(i) = \frac{1}{2}e_j^2(i), \quad e_j(i) = d_j(i) - \mathbf{y}_j(i) \quad \forall j \in \mathcal{C} = \{1, ..., C\} \tag{B.5}$$

such that

$$\xi(i) = \sum_{j \in \mathcal{C}} \xi_j(i) = \frac{1}{2} \sum_{j \in \mathcal{C}} e_j^2(i) = \frac{1}{2} \sum_{j \in \mathcal{C}} (d_j - y_j(i))^2 \tag{B.6}$$

where

$$(d_j - y_j)^2 \equiv \frac{1}{N} \sum_{i=1}^{N} \|d_j(i) - \mathbf{y}_j(i)\|_2 \tag{B.7}$$

This is the case of *batch learning.*

Instead, we may divide $\mathcal{X}$ in smaller subsets, for example $\mathcal{X}_1, \mathcal{X}_2, ...\mathcal{X}_p$. Supposing $m{\cdot}p = n$ and $m, p, n \in \mathbb{Z}$,

$$\mathcal{X}_i \; = \; \{\{\mathbf{x}_k\, , \mathbf{y}_k\} \in \mathcal{X} \mid m \cdot i < k \leq (m+1) \cdot i\} \; \forall i < p \tag{B.8}$$

and then use as an approximation of $\xi(N)$ at each training time,

$$\hat{\xi} = \frac{1}{m} \sum_{i=1}^{m} |d(\mathbf{x}_i) - \mathbf{y}_i|^2 \quad \{\mathbf{x}_i, \mathbf{y}_i\} \in \mathcal{X}_q \text{ for a given q} \tag{B.9}$$

instead of

$$\xi = \frac{1}{K} \sum_{i=1}^{K} |d(\mathbf{x}_i) - \mathbf{y}_i|^2 \quad \{\mathbf{x}_i, \mathbf{y}_i\} \in \mathcal{X} \tag{B.10}$$

If the formula used is as given in B.9 we say it is on-line learning. Indeed, it is sometimes referred to using only a sample per training time, that is $m = 1$.

The case $m = K$ corresponds to *batch learning*, and it is trivial to generalise the on-line learning by relaxing the condition $m \cdot p = n$ (making one of $\mathcal{X}_i$ contain less elements than the others). Therefore *on-line learning* is more general and, in fact, more common, since it requires less storage requirements.

B.2 Backpropagation algorithm

Looking at the last layer of the *neural network*, we can define the *induced local field* that is produced at the input of the neuron j in the last layer as

$$v_j(n) \equiv \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{B.11}$$

where n is the index of the sample, j the index of the neuron in the last layer and i the index of the neurons of the previous layer, and w is the *synaptic weight.*

And the *output signal*is given by

$$y_j(n) = \phi\left(v_j(n)\right) \tag{B.12}$$

Then, as with the *gradient descent algorithm*, we update the weights w_{ij} by $\Delta w_{ij}(n)$, as

$$\Delta w_{ij}/\alpha = \frac{\partial \xi(n)}{\partial w_{ji}(n)} = \frac{\partial \xi(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{B.13}$$

After some simple algebra [24], B.13 becomes

$$\frac{\partial \xi(n)}{\partial w_{ji}(n)} = -e_j(n) \phi_j'(v_j(n)) y_i(n) \tag{B.14}$$

and so the correction made to the wegiths $w_{ij}(n)$ is given by

$$\Delta w_{ij}(n) = -\nu \frac{\partial \xi(n)}{\partial w_{ij}(n)} \tag{B.15}$$

being ν the *learning-rate parameter.* Using B.14,

$$\Delta w_{ji}(n) = \nu e_j(n) \phi_j'(v_j(n)) y_i(n) \tag{B.16}$$

if we define the *local gradient* $\delta_j(n)$ as

$$\delta_j(n) = \frac{\partial \xi(n)}{\partial v_j(n)} = e_j(n) \phi_j'(v_j(n)) \tag{B.17}$$

then

$$\Delta w_{ji}(n) = \nu \delta_j(n) y_i(n) \tag{B.18}$$

If the neuron is located in a hidden layer, the error signal would have to be determined recursively, which is not efficient in practice. Therefore the local gradient $\delta_j(n)$ should be redefined as

$$-\frac{\partial \xi(n)}{\partial y_j(n)} \phi'(v_j(n)) \tag{B.19}$$

in order to compute $\frac{\partial \xi(n)}{\partial y_j(n)}$, the chain rule is used:

$$\frac{\partial \xi(n)}{\partial y_j(n)} = \sum_k e_k(n) \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \tag{B.20}$$

by using previous relations, we get

$$\frac{\partial \xi(n)}{\partial y_j(n)} = -\sum_k \delta_k(n) w_{kj}(n) \tag{B.21}$$

where

$$\delta_j(n) = \phi'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \quad \text{if neuron is hidden}$$

B.3 Implementation

When implementing the *backpropagation algorithm*, the computation needs to be done in two different steps:

- In the *forward pass*, the weights are fixed and the function signals are computed as

  $$y_j(n) = \phi(v_j(n))$$

  where

  $$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)$$

  Being $w_{ji}(n)$ the synaptic weights and $y_i(n)$ the input signal (generated by the neurons in the previous layer).

  For $i = 1$, the input is $x(n)$ the data, and for $i = L$ the number of layers, the output is the predicted output of the neural network, which is wanted to be close to the true label of $x(n)$, $y(n)$ (with respect to a loss function $\mathcal{L}$).

- The *backward pass* starts at the output layer, and pass the error signals throught the neural network, layer by layer, computing the local gradient δ.

B.4 Training ResNets

The *backpropagation algorithm* is general for all *neural networks*, although the expression can be simplified for each specific architecture.

In the case of *ResNets*, the architecture is given by

$$\begin{cases} y_t = y_{t-1} + \mathcal{G}(y_{t-1}) \quad \forall t = \{1, ..., T\} \\ y_0 = x_0 \end{cases}$$

Supposing a simple ResNet:

$$y = x + \mathcal{G}(x)$$

such that the error E is a function of y and a function that we want to approximate, $\hat{y}$.

Then the *backpropagation algorithm* would update the weights in x given the fact that

$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial x} = \frac{\partial E}{\partial y} \cdot (1 + \mathcal{G}'(x)) =$$

$$= \frac{\partial E}{\partial y} + \frac{\partial E}{\partial y} \cdot \mathcal{F}'(x)$$

Appendix C

Complexity and capacity of Neural Networks

There are not general principles for choosing architectures in each specific case, but the approach is rather heuristical.

There is an urgent need to explore and study theoretically the problem of architecture selection, so as to be able to treat more efficiently new problems and domains.

The Universal Approximation Theorems apply to different architectures, and they provide a first theoretical analysis and conditions for approximation. However, the question of which architecture to choose in each specific case is still unknown.

It seems reasonable to think that there are more complex functions than others, as seen by neural networks. Indeed, this is closely related to the problem of control.

There may be two functions that can be approximated by a given architecture, but in practice it is easier to find a solution using backpropagation for one function than to the other. This is related to the problem of initialization, that is produces the stochasticity.

C.1 Capacity of ReLU networks

If using ReLU activations, it can be easily seen that the represented functions are piecewise continuous linear functions PWCL, since the composition of piecewise linear functions is piecewise linear.

Since every *ReLU* function divides the input space by an hiperplane, the number of linear regions can be studied by means of combinatorics.

The problem of knowing how many linear regions can be with the composition of ReLU functions is analogue to the problem of how many times N hiperplanes intersect.

Using this analogy, Serra obtained the following results

Theorem C.1.1 (Bound on the number of linear regions [55])**.** *The number of linear regions in a ReLU neural network is at most*

$$\sum_{(j_1,\dots,j_L)\in J}\prod_{l=1}^{L}\binom{n_l}{j_l}$$

where $J = \{(j_1, ..., j_L \in \mathbb{Z}^L : 0 \leq j_l \leq \min\{n_0, n_1 - j_1, ..., n_{l-1} - j_{l-1}, n_l\}\forall l = 1, ..., L\}$*,* n_l *the number of components per layer* l*, and* L *the number of layers.*

This can be seen as that the number of linear regions goes as $\mathcal{O}(C^L)$.

Then, the expressivity of neural networks is exponentially better with depth.
This does not necessarilly means that deep neural networks are better at approximation functions, but rather that the represented functions are more complex.

In practice, when using multilayer perceptrons, it is known that the more depth is not necessarily better, since the problem of vanishing gradient arises.

So as to study this notion of better / worse the notion of reachability spaces needs to be introduced. The following results are informal, and are aimed at giving a notion of what needs to be done.

Definition C.1.1 (Reachability spaces)**.** *The reachability space with respect to a function* g *and a multilayer perceptron with* C *components and* L *layers is*

$$\mathcal{R}_g[L, C, T, Pr; \epsilon] = \{\hat{g}_\theta \in S_{DNN}[L, C] \mid \mathbb{P}(\mathcal{L}(g, \hat{g}_\Theta) < \epsilon) < Pr\}$$

where T refers to a training time, and being the network initialized with $\Theta = \Theta_0$ following an arbitrary protocol.

Now we have a notion of difficulty of the functions, and on the power of the architectures. Non-rigorous but fundamented definitions are:

- The function h_1 is more difficult than h_2 if $\mathcal{R}[\ ,]$ is dense (in the sense of ϵ and using the distance defined by $\mathcal{L}$) in h_2 but not on h_1 (using the same parameters).
- A DNN $\mathcal{N}_L^C$ is more powerful than $\mathcal{N}_{L'}^{C'}$ if $\mathcal{R}[L', C';\ ] \subset \mathcal{R}[L, C;\ ]$

Once the notion

Definition C.1.2. *An operator $\mathcal{C}$ acting on the function g is a complexity measure iff fulfills:*

- $0 \leq \mathcal{C}(g) \leq \infty$
- *$sup(\mathcal{C}(\hat{g}_\Theta)) = r(L, C)$ with $r(L, C) < r(L+1, C)$ and $r(L, C) < r(L, C+1)$*
- *$\mathcal{C}(h_1) < \mathcal{C}(h_2)$ iff $\mathcal{R}[L, C; T, Pr, \epsilon]$ is dense in h_1 but not on h_2.*

Candidates for complexity measures are

- Number of linear regions, as we know from [55] that $r(L, C) \sim C^{nL}$
- The number of oscillations, which is related to the number of linear regions but can also be applied to other activations such as *tanh*.
- Measures from algebraic topology [21]
- ...

Using the number of linear regions as a complexity measure, an interesting experiment is to train a multilayer perceptron with functions $f_m(x)$.

This functions divide the interval $[-1, 1]$ into $2m$ subitervals $I_1, ..., I_{2m}$ such that $I_1 = [-1, -1 + 1/m]$, $I_{i+1} = I_i + 1/m$ (where the sum translates the interval) and such that $f_m(x) = 0$ for $x \in I_i$ with i even and $f_m(x) = 1$ for $x \in I_i$ for i odd.

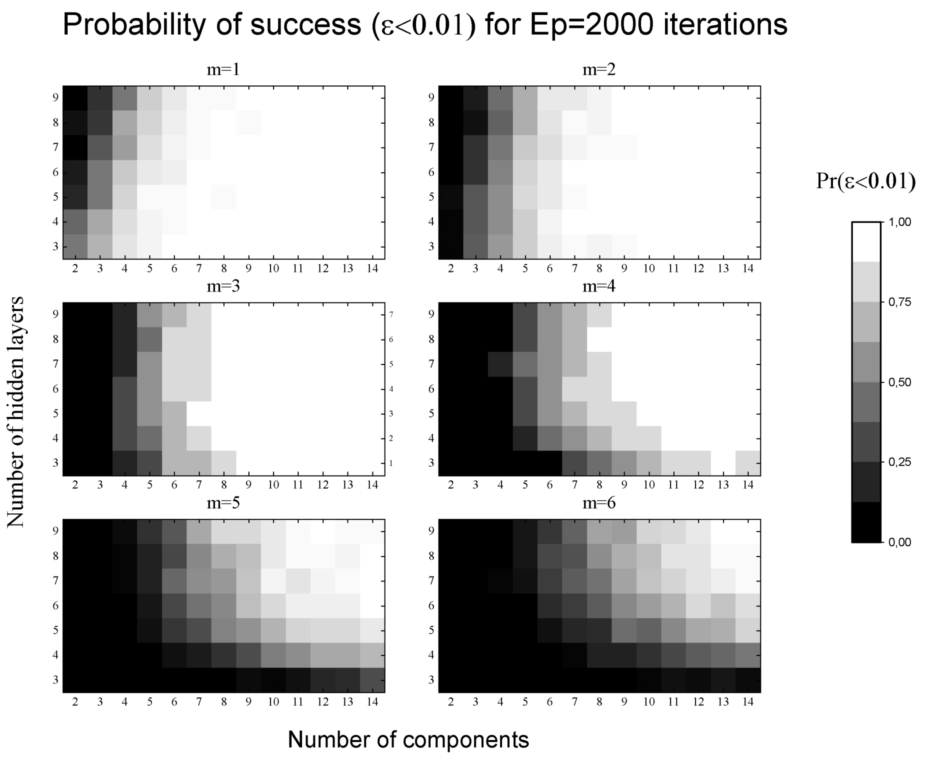

Probability of success (ε<0.01) for Ep=2000 iterations
m=1
m=2
m=3
m=4
m=5
m=6
Number of hidden layers
Number of components
Pr(ε<0.01)
1,00
0,75
0,50
0,25
0,00

Figure C.1

Appendix D

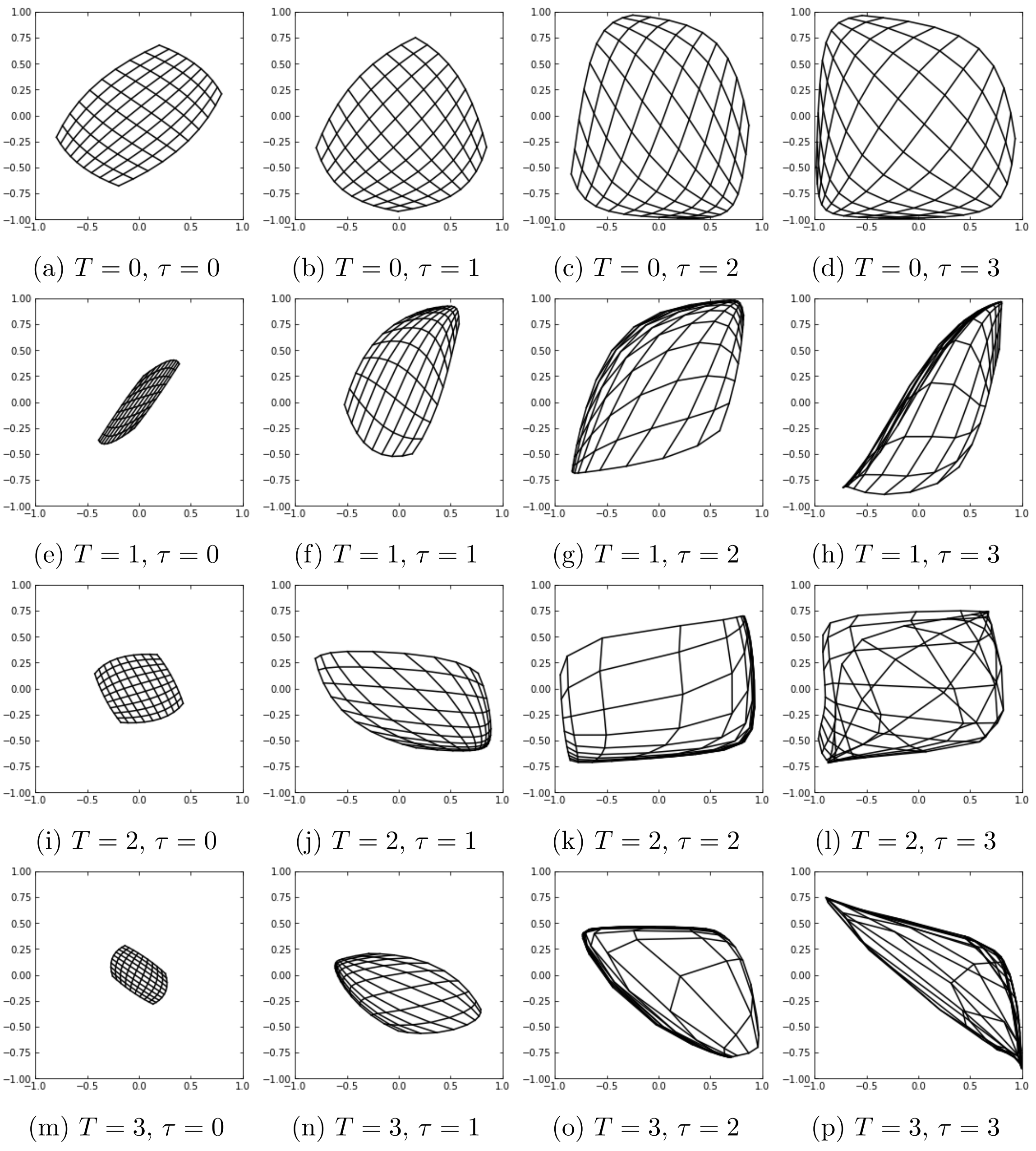
Experimental results

Some experimental results are presented for visualization purposes.

The neural networks are trained using ADAM algorithm, and the loss function is a binary crossentropy.

The implementation has been done using Keras and Tensorflow.

Figures show the level sets and the grid deformation of the same distribution, being noiseless as such that the distance between points corresponding to different classes is nonzero.



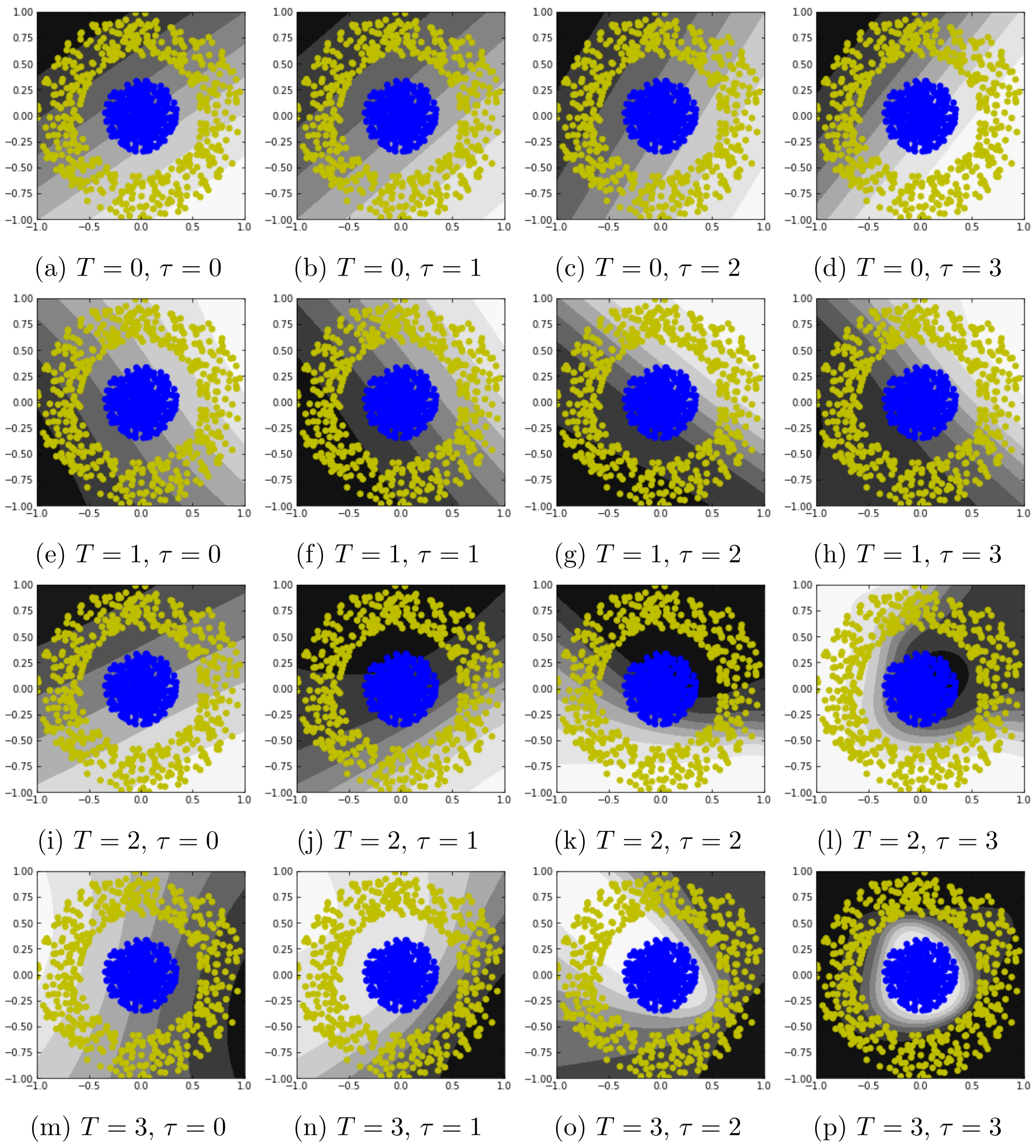
(a) $T = 0$, $\tau = 0$ (b) $T = 0$, $\tau = 1$ (c) $T = 0$, $\tau = 2$ (d) $T = 0$, $\tau = 3$

(e) $T = 1$, $\tau = 0$ (f) $T = 1$, $\tau = 1$ (g) $T = 1$, $\tau = 2$ (h) $T = 1$, $\tau = 3$

(i) $T = 2$, $\tau = 0$ (j) $T = 2$, $\tau = 1$ (k) $T = 2$, $\tau = 2$ (l) $T = 2$, $\tau = 3$

(m) $T = 3$, $\tau = 0$ (n) $T = 3$, $\tau = 1$ (o) $T = 3$, $\tau = 2$ (p) $T = 3$, $\tau = 3$

Figure D.1: Grid defformation with a ring distribution using a multilayer perceptron with tanh activation and 4 hidden layers.



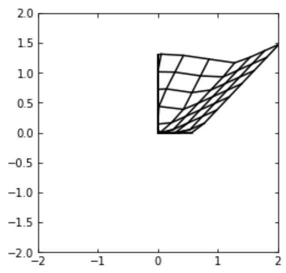
(a) $T = 0, \tau = 0$ (b) $T = 0, \tau = 1$ (c) $T = 0, \tau = 2$ (d) $T = 0, \tau = 3$

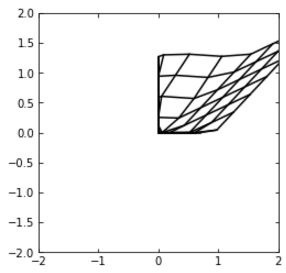
(e) $T = 1, \tau = 0$ (f) $T = 1, \tau = 1$ (g) $T = 1, \tau = 2$ (h) $T = 1, \tau = 3$

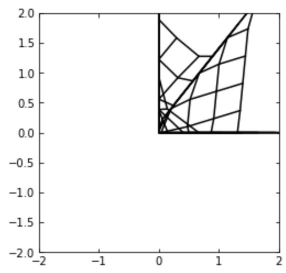
(i) $T = 2, \tau = 0$ (j) $T = 2, \tau = 1$ (k) $T = 2, \tau = 2$ (l) $T = 2, \tau = 3$

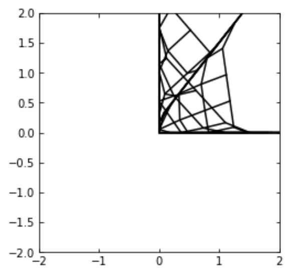
(m) $T = 3, \tau = 0$ (n) $T = 3, \tau = 1$ (o) $T = 3, \tau = 2$ (p) $T = 3, \tau = 3$

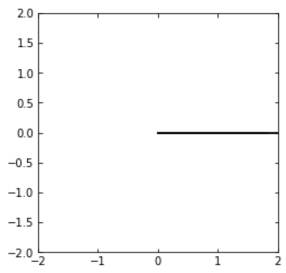
Figure D.2: Level sets with a ring distribution using a multilayer perceptron with tanh activation, 2 neurons per layer and 4 hidden layers.

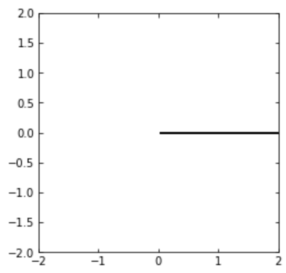

(a) $T = 1$, $\tau = 0$

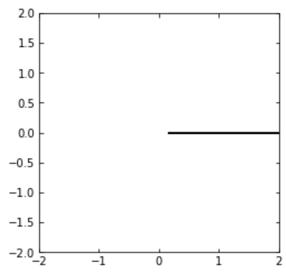

(b) $T = 1$, $\tau = 1$

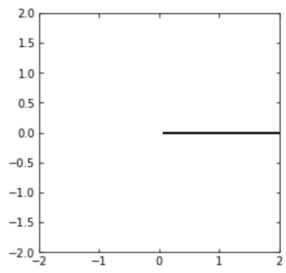

(c) $T = 1$, $\tau = 2$

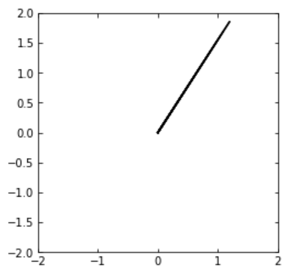

(d) $T = 1$, $\tau = 3$

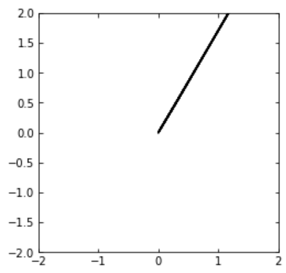

(e) $T = 2$, $\tau = 0$

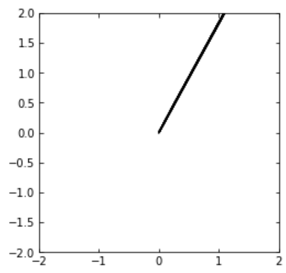

(f) $T = 2$, $\tau = 1$


(g) $T = 2$, $\tau = 2$


(h) $T = 2$, $\tau = 3$


(i) $T = 3$, $\tau = 0$


(j) $T = 3$, $\tau = 1$


(k) $T = 3$, $\tau = 2$

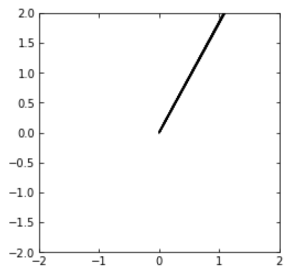
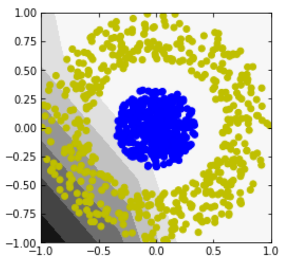
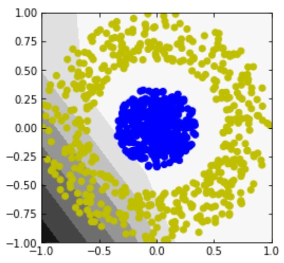

(l) $T = 3$, $\tau = 3$

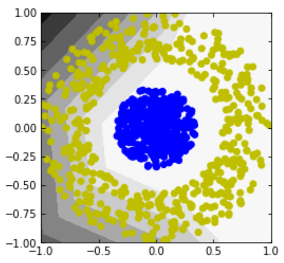
Figure D.3: Grid defformation with a ring distribution using a multilayer perceptron with ReLU activation, 2 neurons per layer and 4 hidden layers.



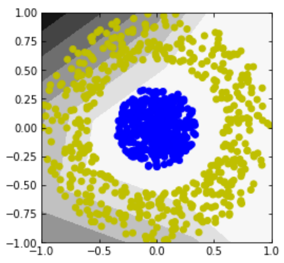
(a) $T = 1$, $\tau = 0$



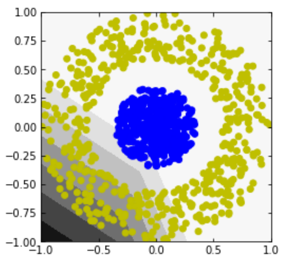
(b) $T = 1$, $\tau = 1$



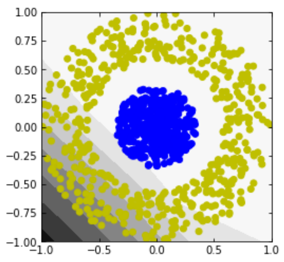
(c) $T = 1$, $\tau = 2$



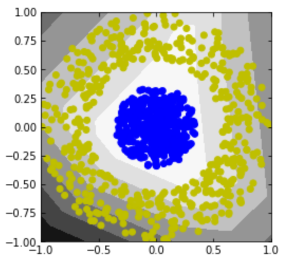
(d) $T = 1$, $\tau = 3$



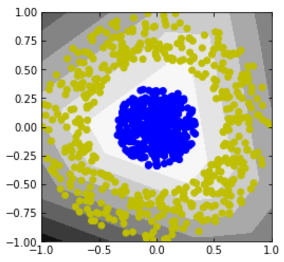
(e) $T = 2$, $\tau = 0$



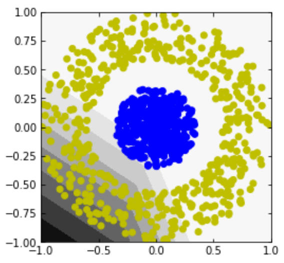
(f) $T = 2$, $\tau = 1$



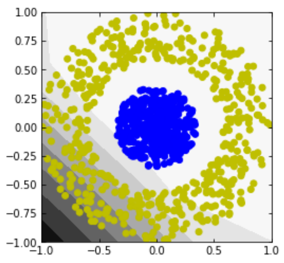
(g) $T = 2$, $\tau = 2$



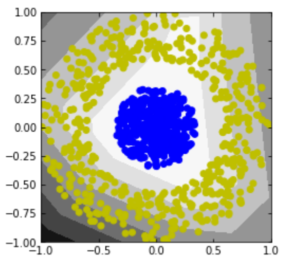
(h) $T = 2$, $\tau = 3$



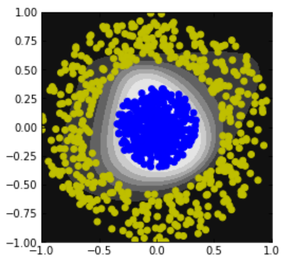
(i) $T = 3$, $\tau = 0$



(j) $T = 3$, $\tau = 1$

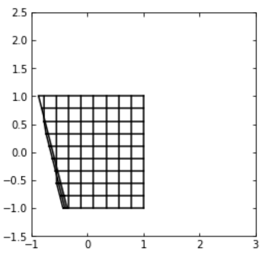


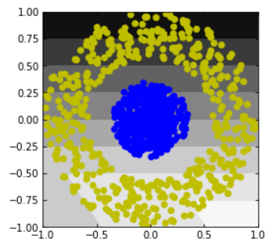
(k) $T = 3$, $\tau = 2$

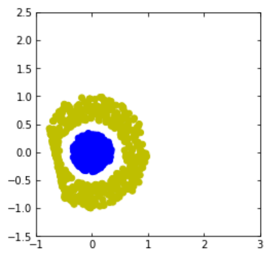


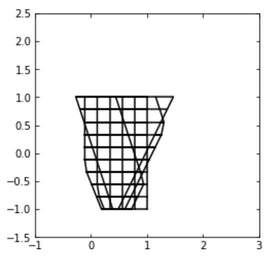
(l) $T = 3$, $\tau = 3$

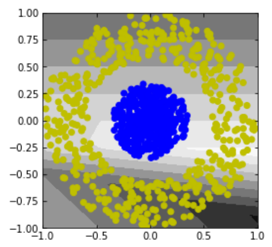
Figure D.4: Level sets with a ring distribution using a multilayer perceptron with ReLU activation, 2 neurons per layer and 4 hidden layers.

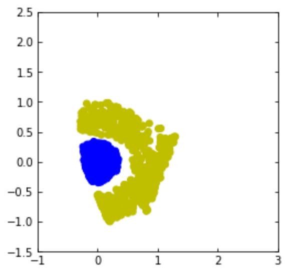

(a) $T = 1$, $\tau = 0$

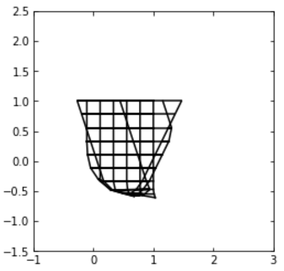

(b) $T = 1$, $\tau = 1$

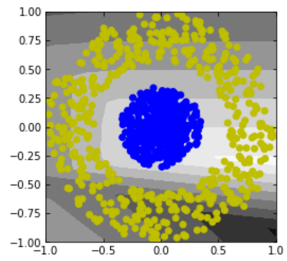

(c) $T = 1$, $\tau = 2$

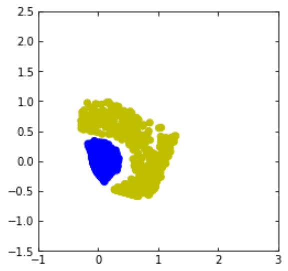

(d) $T = 1$, $\tau = 3$

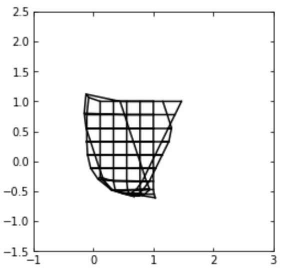

(e) $T = 2$, $\tau = 0$

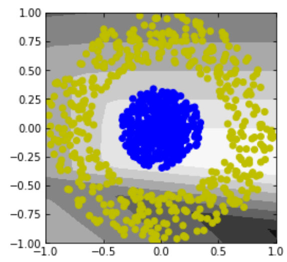

(f) $T = 2$, $\tau = 1$

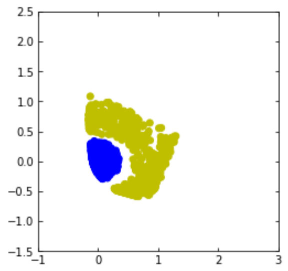

(g) $T = 2$, $\tau = 2$

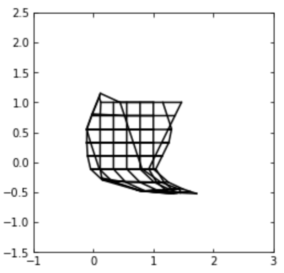

(h) $T = 2$, $\tau = 3$

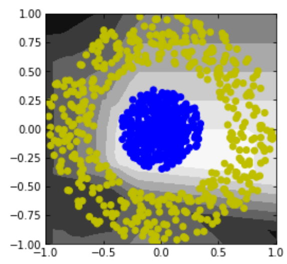

(i) $T = 3$, $\tau = 0$

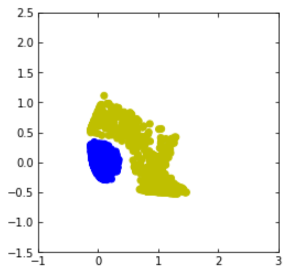

(j) $T = 3$, $\tau = 1$

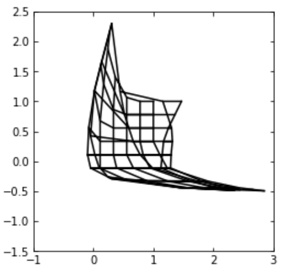

(k) $T = 3$, $\tau = 2$

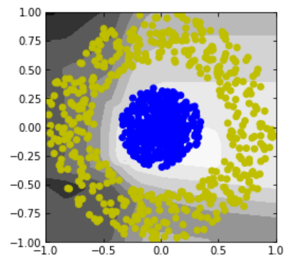

(l) $T = 3$, $\tau = 3$


(m) $T = 3$, $\tau = 1$


(n) $T = 3$, $\tau = 2$


(o) $T = 3$, $\tau = 3$


(p) $T = 3$, $\tau = 1$


(q) $T = 3$, $\tau = 2$

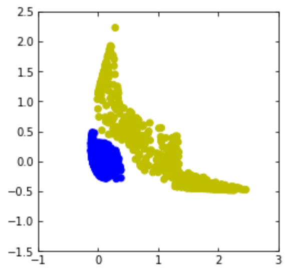
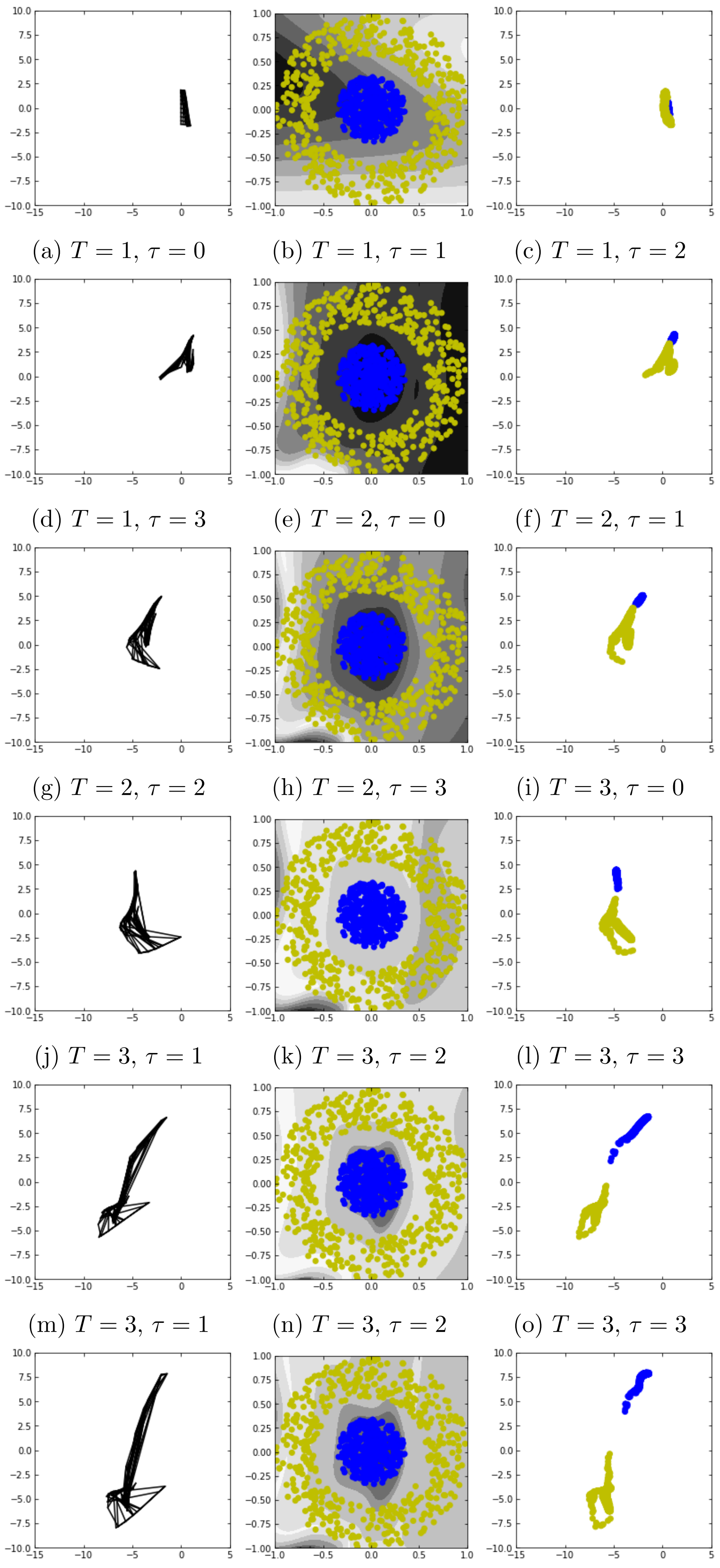

(r) $T = 3$, $\tau = 3$

Figure D.5: Grid deformation, level sets and input propagation with a ring distribution using a ResNet with ReLU activation, 2 neurons per layer and 50 hidden layers.



(a) $T = 1$, $\tau = 0$ (b) $T = 1$, $\tau = 1$ (c) $T = 1$, $\tau = 2$

(d) $T = 1$, $\tau = 3$ (e) $T = 2$, $\tau = 0$ (f) $T = 2$, $\tau = 1$

(g) $T = 2$, $\tau = 2$ (h) $T = 2$, $\tau = 3$ (i) $T = 3$, $\tau = 0$

(j) $T = 3$, $\tau = 1$ (k) $T = 3$, $\tau = 2$ (l) $T = 3$, $\tau = 3$

(m) $T = 3$, $\tau = 1$ (n) $T = 3$, $\tau = 2$ (o) $T = 3$, $\tau = 3$

(p) $T = 3$, $\tau = 1$ (q) $T = 3$, $\tau = 2$ (r) $T = 3$, $\tau = 3$

Figure D.6: Grid deformation, level sets and input propagation with a ring distribution using a ResNet with tanh activation, 2 neurons per layer and 50 hidden layers.

Bibliography

[1] A. F. Agarap, *Deep Learning using Rectified Linear Units (ReLU)*, (2018), pp. 2–8.

[2] I. Arnekvist, J. F. Carvalho, D. Kragic, and J. A. Stork, *The effect of Target Normalization and Momentum on Dying ReLU*, (2020), pp. 1–15.

[3] A. V. Arutyunov, *Pontryagin's maximum principle in optimal control theory*, Journal of Mathematical Sciences, 94 (1999), pp. 1311–1365.

[4] M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon, *Automatic differentiation of algorithms*, Journal of Computational and Applied Mathematics, 124 (2000), pp. 171–190.

[5] R. Bellman, *The theory of dynamic programming*, Bull. Amer. Math. Soc., 60 (1954), pp. 503–515.

[6] C. Cartis, N. I. Gould, and P. L. Toint, *On the complexity of steepest descent, Newton's and regularized Newton's methods for nonconvex unconstrained optimization problems*, SIAM Journal on Optimization, 20 (2010), pp. 2833–2852.

[7] F. Charton, A. Hayat, and G. Lample, *Deep Differential System Stability – Learning advanced computations from examples*, (2020).

[8] V. Chatziafratis, S. G. Nagarajan, I. Panageas, and X. Wang, *Depth-Width Trade-offs for ReLU Networks via Sharkovsky's Theorem*, (2019).

[9] F. Chazal and B. Michel, *An introduction to Topological Data Analysis: fundamental and practical aspects for data scientists*, (2017), pp. 1–38.

[10] S.-W. Chen, C.-N. Chou, and E. Chang, *EA-CG: An Approximate Second-Order Method for Training Fully-Connected Neural Networks*, Proceedings of the AAAI Conference on Artificial Intelligence, 33 (2019), pp. 3337–3346.

[11] M. G. Crandall and P.-L. Lions, *Viscosity Solutions of Hamilton-Jacobi Equations*, Transactions of the American Mathematical Society, 277 (1983), pp. 1–42.

[12] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Mathematics of Control, Signals and Systems, 2 (1989), pp. 303–314.

[13] P. Deuflhard, *Least Squares Problems: Gauss-Newton Methods*, 2011, pp. 173–231.

[14] S. C. Douglas and J. Yu, *Why RELU Units Sometimes Die: Analysis of Single-Unit Error Backpropagation in Neural Networks*, Conference Record - Asilomar Conference on Signals, Systems and Computers, 2018-Octob (2019), pp. 864–868.

[15] M. Drozdzal, E. Vorontsov, G. Chartrand, S. Kadoury, and C. Pal, *The importance of skip connections in biomedical image segmentation*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 10008 LNCS (2016), pp. 179–187.

[16] S. J. Ebski, D. Arpit, N. Ballas, V. Verma, T. Che, and Y. Bengio, *Residual connections encourage iterative inference*, 6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings, (2018), pp. 1–14.

[17] W. R. Esposito and C. A. Floudas, *Gauss-Newton method: Least squares*, in Encyclopedia of Optimization, C. A. Floudas and P. M. Pardalos, eds., Springer US, Boston, MA, 2001, pp. 733–738.

[18] A. Gholami, K. Keutzer, and G. Biros, *ANODE: Unconditionally accurate memory-efficient gradients for neural ODEs*, IJCAI International Joint Conference on Artificial Intelligence, 2019-Augus (2019), pp. 730–736.

[19] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks*, tech. rep.

[20] L. Grüne, *Computing Lyapunov functions using deep neural networks*, (2020).

[21] W. H. Guss and R. Salakhutdinov, *On Characterizing the Capacity of Neural Networks using Algebraic Topology*, (2018).

[22] B. D. Hahn and D. T. Valentine, *Chapter 14 - Introduction to Numerical Methods*, in Essential MATLAB for Engineers and Scientists (Seventh Edition), B. D. Hahn and D. T. Valentine, eds., Academic Press, seventh ed ed., 2019, pp. 299–328.

[23] M. Hauser and A. Ray, *Principles of Riemannian Geometry in Neural Networks*, in Advances in Neural Information Processing Systems 30, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds., Curran Associates, Inc., 2017, pp. 2807–2816.

[24] S. Haykin, *Neural Networks and Learning Machines*, vol. 3, Pearson Education, Upper Saddle River, NJ, third ed., 2008.

[25] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2016-Decem (2016), pp. 770–778.

[26] J. Henriques, S. Ehrhardt, S. Albanie, and A. Vedaldi, *Small Steps and Giant Leaps: Minimal Newton Solvers for Deep Learning*, in 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 10 2019, pp. 4762–4771.

[27] L. G. Jun, *Revisiting Weight Initialization of Deep Neural Networks.*

[28] R. P. Kanwal, *Linear Integral Equations: Theory & Technique*, Modern Birkhäuser Classics, Springer New York, 2012, ch. Method of.

[29] P. Kidger and T. Lyons, *Universal Approximation with Deep Narrow Networks*, (2019).

[30] D. P. Kingma and J. L. Ba, *Adam: A method for stochastic optimization*, 3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings, (2015), pp. 1–15.

[31] A. Knauf, *Neural Ordinary Differential Equations*, UNITEXT - La Matematica per il 3 piu 2, 109 (2018), pp. 31–60.

[32] S. Koturwar and S. Merchant, *Weight Initialization of Deep Neural Networks(DNNs) using Data Statistics*, (2017).

[33] J. S. Kowalik and M. R. Osborne, *Methods for unconstrained optimization problems*, Mathematical Linguistics and Automatic Language Processing, American Elsevier Pub. Co., 1968.

[34] S. K. Kumar, *On weight initialization in deep neural networks*, (2017).

[35] J. Latz, *Analysis of Stochastic Gradient Descent in Continuous Time*, (2020), pp. 1–29.

[36] Y. Lecun, *A Theoretical Framework for Back-Propagation*, (2001).

[37] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, *Efficient BackProp*, in Neural Networks: Tricks of the Trade: Second Edition, G. Montavon, G. B. Orr, and K.-R. Müller, eds., Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 9–48.

[38] Q. Li, L. Chen, C. Tai, and E. Weinan, *Maximum principle based algorithms for deep learning*, Journal of Machine Learning Research, 18 (2018), pp. 1–29.

[39] Q. Li, T. Lin, and Z. Shen, *Deep Learning via Dynamical Systems: An Approximation Perspective*, (2019), pp. 1–30.

[40] L. Liu, X. Liu, C.-J. Hsieh, and D. Tao, *Stochastic Second-order Methods for Non-convex Optimization with Inexact Hessian and Gradient*, ArXiv, abs/1809.0 (2018).

[41] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis, *Dying ReLU and Initialization: Theory and Numerical Examples*, 107 (2019), pp. 1–32.

[42] S. Mallat, *Understanding deep convolutional networks*, Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 374 (2016), pp. 1–17.

[43] S. Massaroli, M. Poli, J. Park, A. Yamashita, and H. Asama, *Dissecting Neural ODEs*, (2020).

[44] A. Mazumdar and A. S. Rawat, *Learning and Recovery in the ReLU Model*, 2019 57th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2019, 1 (2019), pp. 108–115.

[45] H. Mhaskar and T. Poggio, *Deep vs. shallow networks : An approximation theory perspective*, (2016).

[46] M. C. Mukkamala and M. Hein, *Variants of RMSProp and adagrad with logarithmic regret bounds*, 34th International Conference on Machine Learning, ICML 2017, 5 (2017), pp. 3917–3932.

[47] J. Nocedal and S. Wright, *Numerical Optimization*, Springer Series in Operations Research and Financial Engineering, Springer New York, 2006.

[48] A. E. Orhan and X. Pitkow, *Skip connections eliminate singularities*, 6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings, (2018).

[49] X. Qian and D. Klabjan, *The Impact of the Mini-batch Size on the*, (2019).

[50] H. E. Robbins, *A Stochastic Approximation Method*, Annals of Mathematical Statistics, 22 (2007), pp. 400–407.

[51] S. Roweis and L. Saul, *Nonlinear Dimensionality Reduction by Locally Linear Embedding*, Science (New York, N.Y.), 290 (2001), pp. 2323–2326.

[52] S. Ruder, *An overview of gradient descent optimization algorithms*, (2016), pp. 1–14.

[53] L. Ruthotto and E. Haber, *Deep Neural Networks Motivated by Partial Differential Equations*, Journal of Mathematical Imaging and Vision, 62 (2020), pp. 352–364.

[54] M. Schmitt, *Lower bounds on the complexity of approximating continuous functions by sigmoidal neural networks*, Advances in Neural Information Processing Systems, (2000), pp. 328–334.

[55] T. Serra, C. Tjandraatmadja, and S. Ramalingam, *Bounding and Counting Linear Regions of Deep Neural Networks*, (2017).

[56] S. Sonoda and N. Murata, *Double Continuum Limit of Deep Neural Networks*, ICML 2017 Workshop on Principled Approaches to Deep Learning, (2017), pp. 1–5.

[57] ——, *Transport Analysis of Infinitely Deep Neural Network*, Journal of Machine Learning Research, 20 (2019), pp. 1–52.

[58] M. Taki, *Deep Residual Networks and Weight Initialization*, (2017), pp. 1–10.

[59] Y. Tian, T. Jiang, Q. Gong, and A. Morcos, *Luck Matters: Understanding Training Dynamics of Deep ReLU Networks*, 2 (2019).

[60] A. Veit, M. Wilber, and S. Belongie, *Residual networks behave like ensembles of relatively shallow networks*, Advances in Neural Information Processing Systems, (2016), pp. 550–558.

[61] C. Villani, *Optimal Transport: Old and New*, Grundlehren der mathematischen Wissenschaften, Springer Berlin Heidelberg, 2008.

[62] G. Walschap, *Metric Structures in Differential Geometry*, Graduate Texts in Mathematics, Springer New York, 2004.

[63] L. Wasserman, *Topological Data Analysis*, Annual Review of Statistics and Its Application, 5 (2018), pp. 501–532.

[64] E. Weinan, J. Han, and Q. Li, *A mean-field optimal control formulation of deep learning*, vol. 6, Springer International Publishing, 2019.

[65] D. Wu, Y. Wang, S.-T. Xia, J. Bailey, and X. Ma, *Skip Connections Matter: On the Transferability of Adversarial Examples Generated with ResNets*, (2020), pp. 1–15.

[66] Z.-Q. J. Xu, Y. Zhang, T. Luo, Y. Xiao, and Z. Ma, *Frequency Principle: Fourier Analysis Sheds Light on Deep Neural Networks*, 10 (2019), pp. 1–22.

[67] A. Zaeemzadeh, N. Rahnavard, and M. Shah, *Norm-Preservation: Why Residual Networks Can Become Extremely Deep?*, IEEE Transactions on Pattern Analysis and Machine Intelligence, (2020), pp. 1–1.

[68] Y. Zhang, J. Gao, and H. Zhou, *Breeds Classification with Deep Convolutional Neural Network*, ACM International Conference Proceeding Series, (2020), pp. 145–151.

[69] X. Y. Zhou, *Maximum principle, dynamic programming, and their connection in deterministic control*, Journal of Optimization Theory and Applications, 65 (1990), pp. 363–373.