Universidad del País Vasco

Euskal Herriko Unibertsitatea

Máster Universitario en Modelización e Investigación Matemática, Estadística y Computación 2019/2020

Trabajo Fin de Máster

On the use of Neural Networks to solve Differential Equations

---

Alberto García Molina

Tutor/es

Carlos Gorria Corres

Lugar y fecha de presentación prevista

12 de Octubre del 2020

Abstract

English.

Artificial neural networks are parametric models, generally adjusted to solve regression and classification problem. For a long time, a question has laid around regarding the possibility of using these types of models to approximate the solutions of initial and boundary value problems, as a means for numerical integration. Recent improvements in deep-learning have made this approach much attainable, and integration methods based on training (fitting) artificial neural networks have begin to spring, motivated mostly by their mesh-free nature and scalability to high dimensions. In this work, we go all the way from the most basic elements, such as the definition of artificial neural networks and well-posedness of the problems, to solving several linear and quasi-linear PDEs using this approach. Throughout this work we explain general theory concerning artificial neural networks, including topics such as vanishing gradients, non-convex optimization or regularization, and we adapt them to better suite the initial and boundary value problems nature. Some of the original contributions in this work include: an analysis of the vanishing gradient problem with respect to the input derivatives, a custom regularization technique based on the network's parameters derivatives, and a method to rescale the subgradients of the multi-objective of the loss function used to optimize the network.

Spanish.

Las redes neuronales son modelos paramétricos generalmente usados para resolver problemas de regresiones y clasificación. Durante bastante tiempo ha rondado la pregunta de si es posible usar este tipo de modelos para aproximar soluciones de problemas de valores iniciales y de contorno, como un medio de integración numérica. Los cambios recientes en deep-learning han hecho este enfoque más viable, y métodos basados en entrenar (ajustar) redes neuronales han empezado a surgir motivados por su no necesidad de un mallado y su buena escalabilidad a altas dimensiones. En este trabajo, vamos desde los elementos más básicos, como la definición de una red neuronal o la buena definición de los problemas, hasta ser capaces de resolver diversas EDPs lineales y casi-lineales. A lo largo del trabajo explicamos la teoría general relacionada con redes neuronales, que incluyen tópicos como los problemas de desvanecimiento de gradientes (vanishing gradient), optimización no-convexa y técnicas de regularización, y los adaptamos a la naturaleza de los problemas de valores iniciales y de contorno. Algunas de las contribuciones originales de este trabajo incluyen: un análisis del desvanecimiento de gradientes con respecto a las variables de entrada, una técnica de regularización customizada basada en las derivadas de los parámetros de la red neuronal, y un método para rescalar los subgradientes de la función de coste multi-objectivo usada para optimizar la red neuronal.

Acknowledgements

To my advisor Carlos Gorria Corres, for his advice, and to my family and friends who have given me their support in all these months.

Preamble

The structure of this work is divided into 5 chapters and 2 annexes.

Chapter 0 starts by giving an initial pragmatic overview of multi-linear algebra. Its purpose is to give anyone foreign to this subject a working knowledge of tensors: defining their notation and how to operate with them. Tensors will be extensively used throughout Chapter 2 when describing artificial neural networks.

Chapter 1 contains the actual introduction to problem at hand. Here we will be exploring the motivations for using artificial neural networks to numerically integrate initial/boundary value problems. On top of this, we will also be listing the differential operators that will be used, describe the general conditions under which we will be guaranteeing well-posedness, and examine state of the art.

Chapter 2 will layout the theoretical framework of artificial neural networks. It will be covering the everything necessary to define and train a deep learning model from ground zero. The topics covered in this section include: definition and design choices, establishment of an objective (loss) function and non-convex optimization, and the use of regularization techniques. Although these topics are general to deep-learning, throughout this whole chapter we have adapted them, where necessary, to fit the subject of this work.

Chapter 3 is the experimental part of this work. The first three sections contain the discussion on some practical issues, namely, the programming, approximating capacities of artificial neural networks and training multi-objective functions. Following these sections, lie the experiments and simulations of this work. Here we put into practice all the previous knowldege that we have build up to numerically integrate some instances of initial/boundary value problems. On each instance we benchmark and discuss the results for several set-ups based on the different architectures and training options seen up to this point.

Chapter 4 has the final conclusions to this work. An analysis on the limitations and the advantages of this technique with respect to others, as a way to approximate solutions of differential equations, is made. Also, based on the experience from this work, we suggest possible lines of work and open related questions, which can be consider for further work.

Annexes A & B include: a linear algebra perspective of some expressions in Chapter 2 for further clarity, and the code, respectively.

Contents

List of Figures

Table Index

Chapter 0
Overview of Multi-linear Algebra

Generally, when working in the context of artificial neural networks, the framework linear algebra is more than enough to describe the elements and operations taking place. Even when dealing with convolutional networks, which may involve operations on 3 dimensional arrays of objects, one can be decompose everything into vectors matrices, matrix multiplications and element-wise products. Thus, many times, when in the context of artificial neural networks, any explicit reference to multi-linear algebra or the tensor nature of such objects is disregarded.

In this work, however, we will be taking the multi-linear algebra approach. There are two main draws for doing this, i.e. generalizing vectors and matrices to tensors:

- First, the tensor notation is very powerful. This notation serves two purposes: it allows us to represent operations between tensors in a very compact way and it also helps to keep track of dimensions at any time.
- Second, multi-linear algebra provides a simple and natural framework to characterize high order derivatives of multidimensional objects such as vectors or matrices, which is a particularity of this work. In this framework derivatives and the chain rule are really easy to interpret as they visually take the form of the one dimensional case.

For the next part of this chapter we will be covering the basics of tensors. However, since the objective of this work is not to discuss multi-linear algebra, and the only purpose of this chapter is to serve as an entry point to the concepts and the notation of tensors, we will be taking a hands-on informal approach. This means that, there will be no formal definitions and every concept will be explained through an example. For a proper introduction with due rigour one can refer to chapters 2 to 4 in [1].

0.1 What is a tensor?

Perhaps the simplest way to define a tensor is as an element in a tensor space, which is nothing else than a direct product of vector spaces and dual vector spaces. So, for example, lets imagine a random tensor T in the following tensor space:

$$T \in \mathbb{R}^{4^*} \otimes \mathbb{R}^{2^*} \otimes \mathbb{R}^3, \tag{1}$$

then T is of the form $T = v \otimes w \otimes z$, where $v \in \mathbb{R}^{4^*}$, $w \in \mathbb{R}^{2^*}$, $z \in \mathbb{R}^3$. Observe that T is uniquely defined in the tensor space by $4 \times 2 \times 3 = 24$ scalar components (the individual coordinates of v, w and z, having fixed a base in each (dual) vector space).

The previous is essentially a definition of a tensor, but in practice we want to describe a tensor not by a direct product of vectors but by a set of scalar coordinates, the same way we do with a vector space. This is achieved by defining a tensor base. So, given a base for each of the (dual) vector spaces in the tensor space; for the previous example $\{e_1, e_2, e_3, e_4\}_{\mathbb{R}^{4^*}}$, $\{\hat{e}_1, \hat{e}_2\}_{\mathbb{R}^{2^*}}$, $\{\tilde{e}^1, \tilde{e}^2, \tilde{e}^3\}_{\mathbb{R}^3}$; we can intuitively build a base for the tensor space as follows:

$$\{e_i \otimes \hat{e}_j \otimes \tilde{e}^k \mid i = 1, 2, 3, 4,\ j = 1, 2,\ k = 1, 2, 3\}_{\mathbb{R}^{4^*} \otimes \mathbb{R}^{2^*} \otimes \mathbb{R}^3} \tag{2}$$

Now we can describe any tensor in the tensor space through its coordinates in the tensor base, the same way we do with vectors in vector spaces. Thus, T can be represented through its 24 scalar coordinates of the tensor base as:

$$T \equiv \sum_{i,j,k} T_k^{i,j}\ e_i \otimes \hat{e}_j \otimes \tilde{e}^k, \tag{3}$$

where the $T_k^{i,j}$'s are the coordinates. This latest characterization of a tensor as coordinates of a tensor base is the main way of defining tensors formally, and the one that can be found in almost every source, including [1], being any other characterization equivalent to this.

From (3) we can derive the abstract notation for tensors. Looking back at (3) we realize that once the tensor base has been selected, the explicit reference for the summation over all indices and the tensor base can be implicitly assumed, thus we can omit it. So in the end we end up writing the tensor T as $T_k^{i,j}$, where where i, j and k become free indexes which are general place-holders that can take values in $i = 1, 2, 3, 4$, $j = 1, 2$ and $k = 1, 2, 3$. It is important to note that while $T_k^{i,j}$ denotes a tensor in abstract notation, once we replace the free indexes with actual values, for example $T_1^{2,1} \in \mathbb{R}$ we denote a component. In fact, every time we replace a free index with an actual value, we fix the base element associated with that index, hence we obtain a new tensor in a lower dimension tensor space. For instance:

$$T_k^{2,j} \equiv \sum_{j,k} T_k^{2,j}\ e_2 \otimes \hat{e}_j \otimes \tilde{e}^k \in \mathbb{R}^{2^*} \otimes \mathbb{R}^3. \tag{4}$$

Just for the sake of completeness, we will give some important facts and ideas about tensor spaces that will help understand tensors better:

- In principle, since the tensor space is a direct product, the order in which the (dual) vector spaces appear in the product does not affect the nature of the space. The only convention is that the dual vector spaces must appear before the vector spaces; so in practice $\mathbb{R}^{4^*} \otimes \mathbb{R}^{2^*} \otimes \mathbb{R}^3$ can describe the same tensors as $\mathbb{R}^{2^*} \otimes \mathbb{R}^{4^*} \otimes \mathbb{R}^3$. However, in terms of notation the order is important as the indexes must appear in the same order as the (dual) vector spaces in the direct product, by convention. Given the previous example, it is said that the tensors in the first tensor space $T_k^{i,j}$ and the tensors $T_k^{j,i}$ in the second tensor space are related by index juggling.

- A tensor space is said to be of dimension (r, s), if it is the direct product of r dual vector spaces and s vector spaces. Hence, in the example (1), the tensor space is of dimension (2, 1). Note that one of the perks of the abstract notation is that we can know a tensor's dimension easily by simply counting the number of free super-indexes (which yields r) and the number of free sub-indexes (which yields s).

0.2 Tensor Operations and Summation Convention

The are a few operations that can be performed on tensors. For example, if two tensors exist in the same tensor space, $a_k^{i,j}, b_k^{i,j} \in \mathbb{T}$ we can add them by simply adding the coordinates that correspond to the same element of the base. This is:

$$c_j^i = a_k^{i,j} + b_j^{i,j} \equiv \sum_{i,j,k} \left(a_k^{i,j} + b_k^{i,j}\right)\ e_i \otimes \hat{e}_j \otimes \tilde{e}^k. \tag{5}$$

Two tensors $a^i_k \in \mathbb{T}$ and $b^{i',j'}_{k'} \in \mathbb{T}'$, which can exist in very different tensor spaces can always be multiplied in what is known as the tensor product. The resulting tensor exists in the tensor space formed by the direct product combination of the original tensor spaces, $\mathbb{T} \otimes \mathbb{T}'$, the new base becomes the direct product of the original tensor spaces bases, and the new components are the product of the originals. For this example:

$$c^{i,i',j'}_{k,k'} = a^i_k \cdot b^{i',j'}_{k'} \equiv \sum_{i,k} \sum_{i',j',k'} \left(a^i_k \cdot b^{i',j'}_{k'} \right) \; e_i \otimes \tilde{e}^k \otimes \; e_{i'} \otimes \hat{e}_{j'} \otimes \tilde{e}^{k'} \in \mathbb{T} \otimes \mathbb{T}'. \tag{6}$$

There is another "way" in which we can multiply tensors. Lets imagine that we have two tensors $a^i_j \in \mathbb{R}^{2^*} \otimes \mathbb{R}^3$ and $b^{j'}_{k'} \in \mathbb{R}^{3^*} \otimes \mathbb{R}^4$. Observe that in this particular case, the first tensor space contains the vector space $\mathbb{R}^3$, and the second tensor space contains its dual counterpart, the dual $\mathbb{R}^{3^*}$. Since $\mathbb{R}^{3^*}$ is the space of linear applications $\mathbb{R}^3 \to \mathbb{R}$, instead of taking the direct product $\hat{e}_j \otimes \hat{e}^{j'}$ we can apply one to the other $\hat{e}_j(\hat{e}^{j'}) = \delta^{j'}_j$, where $\delta^{j'}_j$ is the Kronecker 's delta, and we are assuming the basis are orthonormal. The rest of the elements of the basis will behave as in the tensor product. This "way" of multiplying tensors is called a contraction, and under this rules the multiplication looks like:

$$\begin{aligned} c^i_{k'} = a^i_j \cdot b^{j'}_{k'} &\equiv \sum_i \sum_{k'} \sum_{j,j'} \hat{e}_j(\hat{e}^{j'}) \left(a^i_j \cdot b^{j'}_{k'} \right) \; e_i \otimes \tilde{e}^{k'} \\ &= \sum_i \sum_{k'} \sum_{j,j'} \delta^{j'}_j \left(a^i_j \cdot b^{j'}_{k'} \right) \; e_i \otimes \tilde{e}^{k'} \\ &= \sum_i \sum_{k'} \sum_{j} \left(a^i_j \cdot b^{j}_{k'} \right) \; e_i \otimes \tilde{e}^{k'} \in \mathbb{R}^{2^*} \otimes \mathbb{R}^4. \end{aligned} \tag{7}$$

To this point we have defined how to sum tensors, and two ways two multiply them: the tensor product and the contraction (when compatible). However, it is confusing that we have been using the same notation to denote the tensor product and the contraction. This is where the Einstein Summation Convention comes in. Note that in (7) we have denoted the contraction operation by $a^i_j \cdot b^{j'}_{k'}$, which contains the free indexes j and j', but looking at the final expression, because we are applying the Kronecker's delta, one of the indexes has become irrelevant. Hence, in the product of two tensors, when the same free index is repeated, the element of the basis for that index gets contracted (Einstein Summation Convention). This means that from now on $a^i_j \cdot b^{j'}_{k'}$ will be a tensor product as in (6), and $a^i_j \cdot b^j_{k'}$ will be a contraction on the j-th index, defined as:

$$c^i_{k'} = a^i_j \cdot b^j_{k'} \equiv \sum_i \sum_{k'} \sum_j \left(a^i_j \cdot b^j_{k'} \right) \; e_i \otimes \tilde{e}^{k'}, \tag{8}$$

and we will call both tensor multiplications. In a practical sense the Einstein summation convention provides some kind of cancellation intuition. We can imagine as if repeating indexes up and down cancel each other in the resulting tensor, $c^i_{k'} = a^i_{\not j} \cdot b^{\not j}_{k'}$.

As a final note, we can also define the transposition of a tensor as $(a^{i,j}_k)^\top = a^k_{i,j}$. In essence the transposition operator acts in each of the vector spaces that compose the tensor space.

0.3 Linear Algebra as Multi-linear Algebra

Multi-linear algebra is a generalization on linear algebra. Thus we can express all objects in linear algebra through tensors and any linear algebra operators as tensor operators. In this section we will draw a parallel between one and the other for the most basic concepts.

The most simple objects that can be described as tensors are vector and dual vectors. A vector in a vector space (a.k.a contravariant vector), which is represented in linear algebra as column array, is a (0,1) dimensional tensor. Similarly, a dual vector in a dual space (a.k.a dual vector, covariant vector, covector or one-form), which is represented in linear algebra as row array, is a (1,0) dimensional tensor. Next, there is an example of a vector and a dual vector resp., in the left written in tensor form and in the right in its linear algebra representation:

$$v^i \equiv \begin{pmatrix} v^1 \\ v^2 \\ v^3 \end{pmatrix}, \qquad v_j \equiv \begin{pmatrix} v_1 & v_2 & v_3 \end{pmatrix}. \tag{9}$$

A matrix in its tensor form simply is a (1,1) dimensional tensor. If we think of a matrix as a linear application, i.e. a vector of one-forms (a vector of applications from $\mathbb{R}^n \to \mathbb{R}$), it is natural that its tensor representation would be a^i_j. Note that $a^{i,j}$ and $a_{i,j}$ would also be able to describe the same number of elements, but they define objects of different nature than a matrix. An example of a matrix $a^i_j \in \mathbb{R}^{3^*} \otimes \mathbb{R}^3$ in tensor and linear algebra representation would be the following:

$$a^i_j = \begin{pmatrix} a^1_j \\ a^2_j \\ a^3_j \end{pmatrix} \equiv \begin{pmatrix} \begin{pmatrix} a^1_1 & a^1_2 & a^1_3 \end{pmatrix} \\ \begin{pmatrix} a^2_1 & a^2_2 & a^2_3 \end{pmatrix} \\ \begin{pmatrix} a^3_1 & a^3_2 & a^3_3 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} a^1_1 & a^1_2 & a^1_3 \\ a^2_1 & a^2_2 & a^2_3 \\ a^3_1 & a^3_2 & a^3_3 \end{pmatrix}. \tag{10}$$

A special case is the identity matrix I^i_j. Observe that its components satisfy that $I^i_j = 1$ when $i = j$, and $I^i_j = 0$ when $i \neq j$. This is actually the definition of the Kronecker's delta, and indeed, it is the Kronecker's delta that is the tensor generalization of the identity, and as a consequence of the identity matrix in linear algebra. Hence, from now on $I^i_j = \delta^i_j$.

Finally, regarding operations we can represent: the scalar product of two vectors as a transposition and a contraction, $w^i \cdot (v^i)^\top$; the outer product of two vectors as transposition and a tensor product, $w^i \cdot (v^j)^\top$; a matrix-vector multiplication as a contraction, $a^i_j \cdot v^j$; and a matrix-matrix multiplication as a contraction, $a^i_j \cdot b^j_k$.

0.4 Derivatives of Vector Functions and Tensors

In this section we will discuss the need of tensors as a tool to deal with high order derivatives of vector functions through an example. Lets suppose we have a vector function f which takes as an input a vector in $\mathbb{R}^3$ and outputs a vector in $\mathbb{R}^2$, through a linear algebra representation this would be:

$$\begin{aligned} f: \quad \mathbb{R}^3 \quad &\longrightarrow \quad \mathbb{R}^2 \\ x = \begin{pmatrix} x^1 \\ x^2 \\ x^3 \end{pmatrix} \quad &\longrightarrow \quad \begin{pmatrix} f^1(x) \\ f^2(x) \end{pmatrix}. \end{aligned} \tag{11}$$

Note that the application does not have to be lineal, as the components f^1 and f^2 can be non-linear.

Now we would like to get the partial derivatives with respect to the components x^1, x^2, x^3, of the input vector. These partial derivatives define the Jacobian matrix, which is well known in vector calculus:

$$J = \begin{pmatrix} \frac{\partial f^1(x)}{\partial x^1} & \frac{\partial f^1(x)}{\partial x^2} & \frac{\partial f^1(x)}{\partial x^3} \\ \frac{\partial f^2(x)}{\partial x^1} & \frac{\partial f^2(x)}{\partial x^2} & \frac{\partial f^2(x)}{\partial x^3} \end{pmatrix}. \tag{12}$$

Considering further higher order derivatives into account is already a problem. The function (11) is a one dimensional object, and its first order derivatives (12) already require a two dimensional object to be represented. Therefore its second order derivatives will require a three dimensional object (a matrix whose elements are dual vectors), to be represented. This is impractical and it is where tensors become really useful.

It is possible to describe the vector function as a tensor, $f^k(x^i) \in \mathbb{R}^2$, and take advantage of multi-linear algebra to represent its partial derivatives. However, first we need to know how to differentiate in the context of tensors. This is part of what is known as Ricci calculus or tensor calculus (a subdomain of differential geometry), which has several ways of defining derivatives on tensors. Nevertheless, in practice, since we will only make use of partial derivatives (which are done with respect to a single component, ignoring the rest), the only thing we need to do is to multiply the following dual vector (one-form) through a tensor product:

$$\frac{\partial}{\partial x^i} = \begin{pmatrix} \frac{\partial}{\partial x^1} & \frac{\partial}{\partial x^2} & \frac{\partial}{\partial x^3} \end{pmatrix} \in \mathbb{R}^{3^*}. \tag{13}$$

Applying the operator to $f^k(x^i)$ once we obtain:

$$\frac{\partial}{\partial x^{i'}} \cdot f^k(x^i) \in \mathbb{R}^{3^*} \otimes \mathbb{R}^2, \tag{14}$$

which has, in fact, the dimensions of a matrix as shown in the previous section. Indeed, this is the Jacobian matrix, and if we were to replace the free indices k and i' to obtain every component and write them in matrix form, we would exactly recover (12).

Applying the operator to $f^k(x^i)$ multiple times we obtain its higher order derivatives as:

$$\frac{\partial^2}{\partial x^{i'} \partial x^{i''}} \cdot f^k(x^i) \equiv \frac{\partial}{\partial x^{i'}} \cdot \frac{\partial}{\partial x^{i''}} \cdot f^k(x^i) \in \mathbb{R}^{3^*} \otimes \mathbb{R}^{3^*} \otimes \mathbb{R}^2, \tag{15}$$

$$\frac{\partial^3}{\partial x^{i'} \partial x^{i''} \partial x^{i'''}} \cdot f^k(x^i) \equiv \frac{\partial}{\partial x^{i'}} \cdot \frac{\partial}{\partial x^{i''}} \cdot \frac{\partial}{\partial x^{i'''}} \cdot f^k(x^i) \in \mathbb{R}^{3^*} \otimes \mathbb{R}^{3^*} \otimes \mathbb{R}^{3^*} \otimes \mathbb{R}^2. \tag{16}$$

0.5 The Chain Rule in Tensor Notation

As we have seen in (14-16), derivatives using the Einstein Summation Convention take a really simple form, which resembles in appearance to the one variable calculus framework. Another case in which this happens is when applying the chain rule. For instance, given $(f^k \circ g^l)(x^i) = f^k(g^l(x^i)) = f^k(y^l) : \mathbb{R}^2 \to \mathbb{R}^3 \to \mathbb{R}^2$ where $f^k : \mathbb{R}^3 \to \mathbb{R}^2$ and $g^l : \mathbb{R}^2 \to \mathbb{R}^3$ the chain rule is applied just by making use of contractions as follows:

$$\frac{\partial}{\partial x^{i'}} \cdot f^k(g^l(x^i)) = \frac{\partial f^k(g^l(x^i))}{\partial g^l(x^i)} \cdot \frac{\partial g^l(x^i)}{\partial x^{i'}} = \frac{\partial f^k}{\partial y^l} \cdot \frac{\partial y^l}{\partial x^{i'}} \in \mathbb{R}^{2^*} \otimes \mathbb{R}^2 \tag{17}$$

Chapter 1
Introduction

Many phenomenons in physics, finance and other branches of science are modelled through equations or systems of equations involving rates of change, i.e. derivatives. These systems are generally classified as: systems of ordinary differential equations or ODEs, when every equation depends on the derivatives of a single variable (usually called time); or systems of partial differential equations or PDEs, when some equation in the system depends on the derivatives of more than one variable.

Finding the solutions to these systems of differential equations is of great importance to make predictions based on these models. Nevertheless, there are very few cases in which one can find an analytic expression for these solutions. Usually this involves having a very specific type of equations and making some kind of assumption on the form of solution. This is the case of methods such as the integrating factor to solve linear ODEs; or the method of characteristics to solve certain types of PDEs, whereby we assume that there is a parametric form to the solutions of the system. Therefore, in many situations, the only way to gain insights from the solutions of these systems of differential equations is to compute them numerically (known as numerical integration).

There are a wide variety of methods to integrate numerically ODEs and PDEs. In general, the main methods follow the following ideas:

- In the case of a system of ODEs we can think of its solutions as functions of the time variable, or simply as trajectories in time. Hence, starting at an initial point (the initial condition), the idea is to numerically compute the trajectory moving along the tangent in discrete intervals (steps) of time. The two principal methods that use this idea are the Runge-Kutta and the Taylor Series methods; where the first "averages" the tangent of intermediate substeps of time, and the second uses the Taylor theorem locally up to some order, to get better approximations of the tangent for larger steps of time.

- When it comes to PDEs the approach of moving through the trajectory becomes useless, as the solutions are no more one dimensional curves, but at least two dimensional or higher surfaces for which there is no unique way to move through them starting at any given point. For these types of systems the most widely used methods are the ones in the Galerkin projection family, whose general oversimplified idea consist in using a base of functionals $\{\phi_k\}_{k=0}^{\infty}$ in an adequate function space (a Sobolev space) to approximate the solution via a linear combination of base elements $u(x) = \sum_{k=0}^{\infty} c_k \phi_k(x)$. Replacing the expression $u(x)$ into the differential equations and projecting into the function space $< \cdot, \phi_k >$, a linear system of equations is obtained which has to be solved to obtain the coefficients c_k. Actually, in this process the main computational expense usually comes with the resolution of the linear system, which has complexity $O(m^3)$, where m is the number of variables, i.e. the number coefficients c_k. This is why selecting a good base is very important, since a good choice can lead to a sparser matrix for the linear system after the projection which can greatly reduce the number of operations.

The two principal methods in the Galerkin family are the spectral method and the finite elements method (FEM), the main difference of which relies in the choice of the base. While the first uses a truncated base of functions which are globally defined in the solution's domain, like a truncated Fourier base $\{e^{2\pi k/L}\}_{k=0}^{N}$; the second uses a finite base $\{\phi_k\}_{k=0}^{N}$ of piece-wise locally defined "tent-like" functions, each associated to a node of a mesh of the solution's domain $\{x_k\}_{k=0}^{N}$, in a way that $\phi_k(x_j) = \delta_j^k$. This last type of methods that use a mesh approach are referred to as collocation methods.

Each of these methods have their own pros and cons. The spectral methods are typically used in very specific equations where a particular base is known to work well, for example the Fourier base on the Laplace equation, or in general, where solutions are known to be periodic. One good quality of these methods is that they are very stable and enjoy very good error properties, however, they are difficult to generalize. Applying an ill-thought choice of base to an equation, can lead to projections $< \cdot, \phi_k >= \int_\Omega \cdot\ \phi_k(x)\ dx$ with very hard integrals to compute and a dense linear system matrix to solve for c_k. In contrast, FEMs are very generalizable as they use piece-wise polynomial and locally defined functions as a base, which make projection integrals $< \cdot, \phi_k >$ relatively easy to compute and the linear system matrix to obtain the c_ks is very sparse given that there is very little overlapping in the base function's compact support. As a downside, its error and stability are strongly tied to the number of nodes (usually large) in the mesh, which cannot be trivially generated either, and requires some computational complexity.

These are some of the most popular methods, but there are many more. For PDEs in particular, some other notable mentions are: the finite differences methods, which makes use of differences between nodes to approximate derivatives; the pseudo-spectral methods, which are similar to the spectrals, but it uses quadrature formulas for the projection integrals; and the Monte Carlo methods, which consist in simulating and averaging copies of the system.

In general, there is no single best method, each has its own trade-offs. Nonetheless, in most instances for PDEs (especially in engineering), the preferred method of numerical integration is the FEM due to its versatility. It avoids any complications of having to choose a particular basis of functions, the mesh can be adapted easily to irregular solution domains, the projection integrals are easy to compute, the linear system matrix to obtain the coefficients c_k is sparse, the stability is good enough for many equations, and the error behaviour is well known and can be reduced refining (adding points to) the mesh. However, as good as this sounds, this method incurs in what is known as a dimensionality curse (impracticality when scaling to larger dimensions). For instance, suppose we want to solve some PDE within a certain error threshold in various dimensions. Solving the 1D version on a segment would require a n node mesh; the 2D version on a square would require a n^2 node mesh; the 3D version on a cube would require a n^3 node mesh; and so on. Hence, to maintain the same error threshold in a system with d dimensions requires a mesh with n^d nodes. If we recall, solving the linear system for the coefficients c_k had computational complexity of $O(m^3)$, with m being the number of nodes in the mesh (same as functions in the basis in the FEMs). Putting all this together means that in order to keep the error under a threshold as the dimension increases the complexity scales as $O((n^d)^3) \equiv O(n^{3d})$. Thus, the conclusion is that: *to keep the error within a certain threshold, the computational complexity of the FEM at least grows exponentially with the dimension of the equations*, not even taking into account the increase in computations for the mesh or the projections. Already a system in 4 dimensions starts to be impractical to solve with FEMs. (From now on, 4 or more dimensions is high-dimensional).

High dimensional systems are not very common in physics, but arise in many fields such as sociology and economics. For example, if we were to consider option pricing in finance, assuming the market parameters constant (to not incur in a stochastic problem), the system would be modelled after a PDE which has at least as many variables and dimensions as stocks in the portfolio as well as the time, which is generally a large number [2]. In cases such as the one we have just exposed, FEMs are impractical and Monte Carlo methods are used [3], but still have some stability limitations. For this reason in recent times, with the many improvements in artificial neural network, new machine learning methods have resurged as potential candidates to deal with these kinds of high dimensional problems. The main idea is based in using the good qualities of artificial neural networks as function approximators.

An artificial neural network is just a complex parametrized function $\mathcal{N}(x;\theta)$, which uses modular architecture based on the concept of neurons, has a structure optimized for computer processing, and makes use of non-linear optimization algorithms to train its parameters to fit some model (we will cover this in Chapter 2). Basing ourselves in the previous simplified definition, the deep-learning approach should be straight forward, simply put: the method will approximate the exact solution by taking an artificial neural network, replacing it into the differential equation and using an optimization algorithm to train its parameters so that the equation is satisfied; all while making use of deep-learning strategies to speed up the process. In [4] this type of methods is referred to as "Deep Galerkin Method", the reason being that: both methodologies revolve around approximating the exact solutions of a differential equation via a parametrized function, either a linear combination of base functions or an artificial neural network; and both involve replacing this approximation into the differential equation and solving an inverse problem to find its coefficients or parameters. However, the artificial neuron strategy differ much in nature and lacks many of the elements of the methods in the Galerkin family as it does not: take into account the idea of weak formulation (which we have not explained here for simplicity); use linear combination of base functions and projections; and the resulting inverse problem does not lead to solving a linear system of equations in favour all in favour of a pure non-convex optimization. In fact, it is because of these differences that this machine learning approach should be, in theory, able to scale well with dimension, since in using non-convex optimization, all dimensions are trained at the same time, which should not increase much computational cost. On the other hand, one of the main problems is that the error is not bound by an order and is unpredictably subject to optimization and training particularities. The following table summarizes all the above:

Galerkin Methods (FEM)	**Artificial Neural Network Methods**
Approximates the solution with a base of linear functions.	Approximates the solution with an artificial neural network.
Requires computing some integrals (or quadratures) and solving a linear system.	Requires solving a non-convex optimization problem.
Error order and stability properties known.	Error and stability unknown and depends on the specifics of the optimization.
The complexity scales exponentially with the dimension.	Generalizes well to higher dimensions with just a few more neurons.

Table 1.1: Comparison between FEMs and the Artificial Neural Network Methods.

In this work, we will be using deep-learning techniques and methodologies to try and solve some instances of differential equation. The objective will be to analyse the viability and capabilities of these methods. Although the main interest for this methods is in integrating PDEs (as the existing ODE integration methods are already very efficient), we will start in a progressive way, by studying its application on ODEs (which can be seen as a particular case of PDEs). Then we will scale up the complexity of the operators until we are able to solve some low-dimensional PDEs. Higher dimensional equations will be out of scope since the aim is to illustrate the feasibility, and strengths-weaknesses of this strategy for which two dimensions will be enough.

1.1 Posing the Problem

In its most general form, a system of differential equations with solution in the real space may be represented as follows:

$$\mathcal{L}[u(x)] = f(x), \qquad x \in \Omega \subseteq \mathbb{R}^n, \tag{1.1}$$

where $\Omega \subseteq \mathbb{R}^n$ is a compact manifold, $\mathcal{L}[\cdot]$ is a differential operator, $f(x) : \Omega \to \mathbb{R}^m$ is the external force, and $u : \Omega \to \mathbb{R}^m$ is a solution of the system. Note that (1.1) may represent either a system of ODEs or PDEs depending of the differential operator. The list of operators which we will be solving in Chapter 3 are:

Name	**Expression**
Identity Operator	$\mathcal{L}[u(x)] = u(x)$
1D Divergence Operator	$\mathcal{L}[u(x)] = \dfrac{\partial u(x)}{\partial x}$
2D Divergence Operator	$\mathcal{L}[u(x,y)] = \dfrac{\partial u(x,y)}{\partial x} + \dfrac{\partial u(x,y)}{\partial y}$
2D Laplacian Operator	$\mathcal{L}[u(x,y)] = \dfrac{\partial^2 u(x,y)}{\partial x^2} + \dfrac{\partial^2 u(x,y)}{\partial y^2}$
1D Advection Operator	$\mathcal{L}[u(x)] = u(x) \cdot \dfrac{\partial u(x)}{\partial x}$
2D Clairaut Operator	$\mathcal{L}[u(x,y)] = x \cdot \dfrac{\partial u(x,y)}{\partial x} + y \cdot \dfrac{\partial u(x,y)}{\partial y}$
2D Burgers Operator	$\begin{aligned}\mathcal{L}[\mathbf{u}(x,y)] = \; & \mathcal{L}[(u_x(x,y), u_y(x,y))] \\ = \; & \left(u_x(x,y) \cdot \dfrac{\partial u_x(x,y)}{\partial x} + u_y(x,y) \cdot \dfrac{\partial u_x(x,y)}{\partial y}, \right. \\ & \left. u_x(x,y) \cdot \dfrac{\partial u_y(x,y)}{\partial x} + u_y(x,y) \cdot \dfrac{\partial u_y(x,y)}{\partial y} \right)\end{aligned}$

Table 1.2: List of differential operators used in Chapter 3.

Recall that at the start of this section, in defining (1.1) we indicated that u was "a" solution to the system of differential equations. In fact, there are usually many solutions or none may even exist. To ensure existence and uniqueness of the solution we need to impose some additional conditions to (1.1), namely initial conditions on ODEs and boundary conditions on PDEs. The most common set of these conditions are:

$$\text{Cauchy (ODE):} \qquad u(x_0) = u_0 \tag{1.2}$$

$$\text{Dirichlet (PDE):} \qquad u(x) = g(x), \quad x \in \Gamma \equiv \partial\Omega \tag{1.3}$$

$$\text{Neumann (PDE):} \qquad \frac{\partial u(x)}{\partial n(x)} = g(x), \quad x \in \Gamma \equiv \partial\Omega \tag{1.4}$$

$$\text{Cauchy (PDE):} \qquad u(x) = g_1(x) \wedge \frac{\partial u(x)}{\partial n(x)} = g_2(x), \quad x \in \Gamma \equiv \partial\Omega \tag{1.5}$$

where Γ or $\partial\Omega$ (depending on the convention) is the border of the domain Ω and $n(x)$ is the normal vector at the point $x \in \Omega$. Descriptively, Cauchy initial conditions fix the solution value at a certain point; Dirichlet border conditions, fix the solution values at the border of the domain; Neumann border condition, fix the flow coming in and out of the domain; and finally, Cauchy border conditions are a mix of Dirichlet and Neumann conditions. [5]

Before proceeding, one observation has to be made on ODEs. Given a single ODE of n-th order (the highest derivative in the equation has order n) with $n > 1$, it is common practice to transform the equation into a system of first order ODEs by simply introducing the following set of $n-1$ equations $u_1 = u'$, ..., $u_{n-1} = u'_{n-2} = u^{(n-1)}$, and using them to replace any derivatives of order higher than one in the original equation. This means that any given n-th order ODE is equivalent to a system of n first order ODEs; thus finding a solution to the original n-th order equation, $u(x)$, is equivalent to finding an extended manifold solution in the corresponding system of first order equations, $\mathbf{u}(x) = (u, u_1, ..., u_{n-1})(x)$, which includes its derivatives. The (1.2) definition of Cauchy initial conditions is based on this last paradigm where we consider systems of first order ODEs. Hence, when considering a n-th order ODE in its original form, the equivalent of fixing an initial point on the manifold solution $\mathbf{u}(x_0) = (u_0, u_{1,0}, ..., u_{n-1,0})$ is to fix the value of its first $n-1$ derivatives, and on those premises these conditions should be written as $u(x_0) = u_0$, ..., $u^{(n-1)}(x_0) = u_{n-1,0}$.

Summarizing, we will be considering systems of differential equations (1.1) in combination some initial/boundary conditions (1.2-1.5), mainly Cauchy conditions, to formulate what are known as an initial/boundary value problems. The main objective is to formulate a "well-posed" problem: a set of basic properties which is required to apply any numerical integration successfully. A system of differential equations is said to be well-posed in the Hadamard sense [5], if it holds the following three conditions:

- A solution exists.
- The solution is unique.
- The is stable, i.e. it changes continuously with small variations of its initial conditions, boundary conditions and external force.

Proving that a given problem is well-posed is a really tricky matter. There are very few general results and many of the proofs are case specific: they may apply to certain types of differential operators (for example linear or Poisson operators), require a certain type of boundary conditions and impose several degrees of regularity.

As shown in Table 1.2, in this work we will be using very simple differential operators, all of them linear or quasi-linear. Also, the external force terms will always be an analytic functions (actually polynomials) and the initial/boundary conditions will be for the most part of Cauchy type. Under these specific conditions the Cauchy-Kovalevskaya theorem guarantees the existence of an unique analytic solution to the expression in both the ODE and PDE cases.

Nonetheless, this theorem has its limitations:

- First, it is a local theorem, although this can be remedied if all the terms are analytic everywhere to form a global version by "stitching" the local solutions in several local neighbourhoods to form a cover of the domain and build a global solution. Since the solution has to be unique in the intersection of the neighbourhoods the global solution has to be unique.

- Second, the proof is very dependant on the analyticity of the coefficients in the operator and external force as its proof relies on the methods of majorants. The sketch of this proof goes as follows [6, 7]: first we assume that the solution can be written as power series in some neighbourhood $U \subseteq \Omega$, the coefficients of which are obtained from the initial conditions and replacing the power series into the differential equation. Then we attempt to find some power series that majorates the solution power series, the definition of which is that $\sum_k a_k x^k$ majorates $\sum_k b_k x^k$ iff $|a_k| < b_k$. Finally, we use the property that states that if a series is majorated by a series that converges, so does that series. If the majorating series to the solution power series is adequately chosen and converges, so does the solution power series which converges to the local unique analytic solution.

  When the differential equation is linear or quasi-linear, for every $x_0 \in U \subseteq \Omega$, there is always a systemic change of variables $h : U \to V$ such that $h(x_0) = 0$. Then, on this new domain V, we can always construct a power series that converges to 0 with radius of convergence $\rho = 1$, and majorates the power series solution $u(h(x)) : V \to \mathbb{R}^n$. This makes the proof independent of the differential operator as long as it is linear or quasi-linear. Note that, this proof is constructive as the power series solution satisfies the differential operator and initial/boundary conditions, and it converges on a neighbourhood $h^{-1}(D_0(1))$ of x_0. Therefore, it is a valid local solution, and likewise its uniqueness is proven from a similar argument.

  The assumptions of analyticity and constructiveness of the proof in this theorem implies that the theorem is proving that there is a unique analytic solution. This is much different than claiming that the only solution is analytic. Hence, the initial/boundary problem could still have other non-analytic solutions. (Actually, in the case of ODEs, the Picard–Lindelöf theorem guarantees general uniqueness over other solutions, so the analytic one is the only one; but there are no similar results for PDEs.)

Despite the two potential limitations in applying the Cauchy-Kovalevskaya theorem that we have just seen, this will be enough for the artificial neural network to approximate the analytic solution of the problem. The reason for this assumption is that the artificial neural networks will be composed of analytic functions (almost everywhere), thus we expect them to fit preferentially that solution. From now on, there will be no further discussions about the well-posedness of the initial/boundary problems that we will attempt to solve in this work, the Cauchy-Kovalevskaya theorem will always apply.

1.2 Relevant Literature

The approach for solving differential equation systems dates relatively "old"; at least, we have found and article [8], dating back to 1994. Although this article uses a graph-like structure acknowledged as an artificial neural network to solve ODEs, it applies a FEM type of "tend-like" activation functions and does not rely in "training" in the modern sense, i.e. defining a non-convex optimization problem, opting instead for some kind of Galerkin method hybrid. Throughout this article there are some references of some papers which use some kind of mean square error and non-convex optimization (the most standard approach nowadays), but the author regards them as computationally expensive. This shows that the state of the field of deep-learning back then did not allow for these strategies to be viable candidates to integrate differential equations.

Moving to more recent times, articles explicitly integrating ODEs with artificial neural networks are hard to come by, since as explained before, there are very efficient methods already to integrate ODEs, and the main interest is in PDEs. A related case that we found very interesting and worth mentioning is [9], which uses a reverse approach. Instead of training an artificial neural network to integrate an ODE, it uses ODE numerical integrators to train artificial neural networks.

With regards to PDEs, [4] is a very complete work. It defines a loss by the discretization over a random collocation of points, of the error of the artificial neural network with respect to the boundary value problem (the same idea we will be using to define a loss in 2.2). Then, it goes to solving very high dimensional free boundary PDEs (for American options), and boundary problems (for the Hamilton-Jacobi-Bellman). Two interesting features in this work are that: in the article is called Monte Carlo method for fast computation of second derivatives, which is a type o synthetic gradient; and proof to restricted version of a universal approximation theorem for the solution of PDEs. A synthetic gradient [10] is usually used for very large networks or very large amounts of data. Instead of computing the exact derivatives of the loss function with respect to the parameters required to minimize the loss, the derivatives are drawn from a distribution which is updated for every step of the training. This technique trades off not having the exact derivatives, with less computational cost and possibility of asynchronous training. The authors of [4] use this Monte Carlo method to avoid the expensive cost of a second order automatic differentiation for very high dimensions. In this work, we will not be considering this technique since, unlike [4] which integrates PDEs of up to 200 dimensions, we only integrate PDEs of up to 2 dimensions (not really high dimensions) like almost all of the other papers that we will review next do. However, the use of a synthetic gradient is a recurrent theme in papers dealing with very high dimensional systems.

Other paper focused on high dimensions are [11] and [12]. Both are similar in that they do not consider deterministic PDEs but BSDEs (Backward Stochastic Differential Equations) such as the Allen-Cahn (physics) or Black-Scholes (economics) equations, and they consider systems of up to 100 dimensions.

Closer to the line of work of this project are [13, 14, 15, 16, 17, 18]. The outlines of these work are quite similar: they simulate PDEs of up to 2 dimensions and do some kind of error analysis. Some the characteristics of [15] is that it analyses the effect in the error of the mesh and number of hidden nodes, and in [16, 17] the method is compared to an standard FEM method. On the more interesting side of things lie [13] and [18].

[18] follows up on the architecture of [4], which uses a special kind of feed-forward artificial neural network. In a regular feed-forward artificial neural network the neurons are divided into sequential layers, then the outputs of a layer strictly get fed as input to the next layer (we will see this in section 2.1). However, [4] used an architecture where the outputs of a layer feed all of its successive layers. This seem to yield good results although there is no comparison to other type of architectures.

[13] is focused in, instead of using the artificial neural network approach for its capabilities to integrate systems in high dimensions, in using its mesh-free nature to integrate over irregular domains. In this paper artificial neural networks are trained to fit the advection and diffusion operators for collocations of very irregular domains. It also applies a very original idea which is to consider the approximated solution as $\hat{u}(x) = g(x) + D(x) \cdot \tilde{u}(x)$, where $g(x)$ is the boundary condition, $D(x)$ is a distance function to the border such that $D(x) = 0$ iff $x \in \Gamma$, and $\tilde{u}(x)$ is a regular artificial neural network. This way $\hat{u}(x)$ always satisfies the boundary conditions by construction and it is only required to train the model to fit differential equation, thus one can pre-compute the distance from the collocation to the border since it does not change throughout the training, and focus on a single objective loss function. For this work we reckoned that this idea would only work well with Neumann or Dirichlet boundary conditions, but with Cauchy boundary conditions which include both at the same time.

Other papers that relate artificial neural networks to PDEs are: [19], which is a version of [9] relating PDEs to the dynamics of non-convex optimization in convolutional networks; and [20] which draw the same relation between the dynamics of the optimization of general artificial neural networks and PDEs, using statistical physics techniques. Also [21, 22, 23] are a series of papers by the same author which train artificial neural networks with experimental data from physics to learn the underlying behaviours modelled by PDEs.

Finally, we want to remark that the need for the derivatives of artificial neural networks with respect to inputs, which seem to be something that would not appear in simple regression or classification problems, thus only related to this topic, has been used in other contexts. [24, 25] are examples of this, both make use of information about the derivatives as regularization techniques and to speed up training in problems with no imperative use for them.

Chapter 2
Artificial Neural Networks Framework

In this chapter we will be covering from scratch everything about artificial neural networks that we will be using to solve differential equations in the next chapter. We will start by defining what an artificial neuron and artificial neural network are; explain the architecture of a fully-connected feed-forward neural network; their possible activation functions and initializations; show how to assign a loss function to train the model to fit the initial/boundary value problem; discuss the pros and cons of the main optimizer options available to train the model; and examine diverse regularization techniques which help improve training results.

2.1 What are Artificial Neural Networks?

Artificial neural networks are ensembles of units called artificial neurons. There are many designs for these artificial neurons, and by combining and arranging them in different ways we can create networks with very different behaviours and results.

In this work, the only type of artificial neuron that we will be using is known as the perceptron, which is probably the simplest and the most widely used. Figure 2.1 shows the basic scheme of a perceptron. A percepton takes in n inputs, which we can view as coordinates of a vector x_n; it combines them linearly multiplying weights and adding a bias; and then applies a (mostly) non-linear function a to the linear combination.

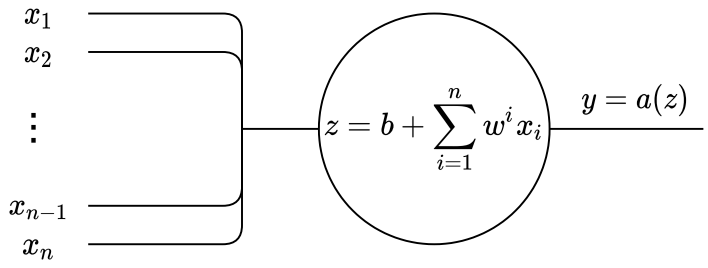

Figure 2.1: Perceptron scheme.

Some other popular design are convolutional neurons, which are used in image recognition; and memory cells, which are used in data with time dependencies such as video processing.

The way in which we combine artificial neurons to form artificial neural networks is such that the outputs of group of neurons become the input of another group of neurons. In this light, one can think of an artificial neural network as a directed graph with entry and exit edges, where each of the nodes correspond to a neuron and the directed edges indicate which neurons outputs feed another neuron as inputs. This is why many times the neurons in a network are referred to as nodes.

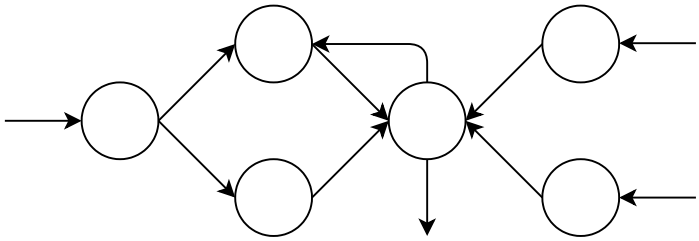

Figure 2.2: A directed graph which could be a possible representation of the architecture or an artificial neural network. Nodes are artificial neurons and edges indicate which neurons feed into each other.

One approach for organizing these neurons is using a feed-forward scheme. By this scheme we divide the neurons into sequential layers (groups). Then the outputs of a layer can only become inputs of the next layer. Using the graph characterization, this would correspond to artificial neural networks defined by directed graphs without cycles. The main advantage of this scheme is it allows for the use standard back-propagation algorithm (which we will be explaining in the next sections) to train the parameters in the neurons to fit a certain model.

A particular case of feed-forward artificial neural networks is the fully-connected. This happens when all the neurons in a layer are connected to all in the neurons in the next layer.

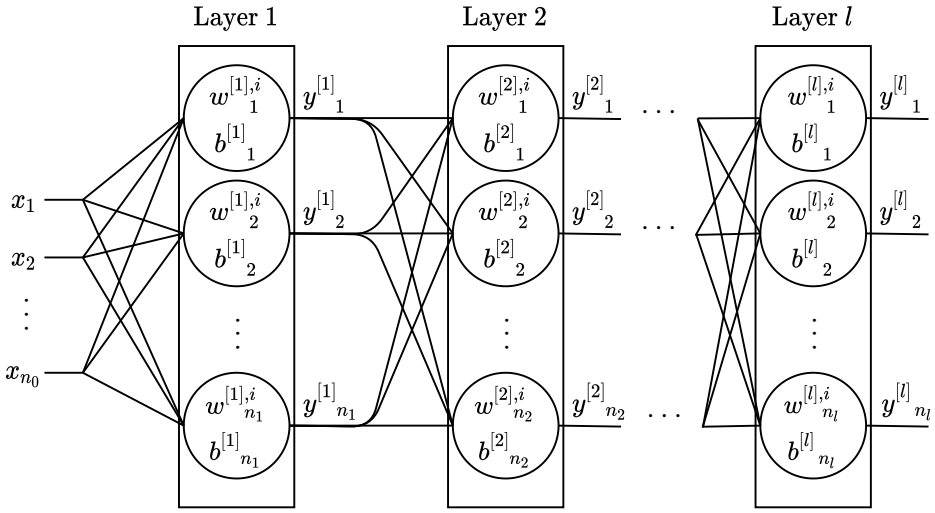

Figure 2.3: General scheme of a perceptron based fully-connected feed-forward artificial neural network.

Throughout this work we will be exclusively using perceptron fully-connected feed-forward artificial neural networks with different number of layers and different number of nodes per layer to approximate the solutions of differential equations. Figure 2.3 shows the standard graph representation of the structure of such neural networks based on the concepts explained up to this point. Note that the edges are not directed, as it is unnecessary, since by standard convention the flow of the neurons goes from left to right.

An artificial neural network provides a frame to define complex parametric functions in a modular way as a composition of simpler operations encapsulated in artificial neurons (we will insist in this point the next sections).

The following observations are a way to better understand artificial neural networks:

- A single neuron can form a neural network. In that case, if the activation function is identity, and we adjust the parameters of the neuron so the outputs fit a continuous dataset, we perform a linear regression. Indeed w^i and b are simply the slope and the intercept. Similarly, if the activation function is a sigmoid, and we adjust the parameters of the neuron so the outputs fit a binary dataset, we perform a logistic regression.

- In more complex deep neural networks, we kind of expand the same ideas as in the single neuron. In general, in artificial neural networks, what happens in regression problems is that we fit the parameters to make the hyper-surface defined by network structure get as close as possible to the sample data; and in classification problems we fit the parameters as much as possible to match the underlying marginal distribution of each category with the network structure.

Finally we will end this introductory section with some nomenclature for the rest of the work:

- A feed-forward neural network is said to be deep when it contains more that one layer.

- In deep neural networks, layers are classified as input, hidden and output layers. The input layer is the one that simply takes in the inputs and does no transformations; the output layer is the last layer of the sequence of layers and its outputs are the output results of all the whole network; and the hidden layers are all that lie between the input and output layers. Based on Figure 2.3, the input vector corresponds to the input layer (layer 0, even though it is not explicitly referenced as that), layers 1 to $l-1$ would be the hidden layers and layer l would be the output layer.

- A feed-forward fully-connected network is defined by its number of layers, the number of neurons (nodes) in each layer, and its types of neurons. A neuron is defined by its weights, bias and activation function. Therefore, from now on we will use the following standard nomenclature, which contains all the elements that we need, to completely define our networks:

ℓ	layers indexes,
n_ℓ, m_ℓ	neurons indexes for layer ℓ,
$w^{[\ell]}{}_{n_\ell}^{m_{\ell-1}}$	weights of the neuron n_ℓ in the layer ℓ (parameters),
$b^{[\ell]}{}_{n_\ell}$	bias of the neuron n_ℓ in the layer ℓ (parameters),
$a^{[\ell]}{}_{n_\ell}$	activation function of the neuron n_ℓ in the layer ℓ,
$z^{[\ell]}{}_{n_\ell}$	result of the linear combiantion of the neuron n_ℓ in the layer ℓ,
$y^{[\ell]}{}_{n_\ell}$	result of applying the activation function of the neuron n_ℓ in the layer ℓ.

  By this notation we will consider the input vector as layer 0 for which no transformations are performed, therefore, $x_i = z^{[0]}{}_i = y^{[0]}{}_i$. Also, the output layer (the n-th layer) never has an activation function, therefore $\hat{u}_i = z^{[n]}{}_i = y^{[n]}{}_i$.

2.2 From Numerical Integration to Deep-Learning

The intended use of the artificial neural networks in this work is for them to approximate the solution of some initial/boundary value problem. In order to achieve this, the parameters of the network should be tweaked to minimize some measure representing how well the network satisfies the differential operator and initial/boundary conditions.

Given the artificial neural network approximation of the solution, $\hat{u}(x; w, b)$, which depends parametrically on the set of all weights w and all biases b, we can define a positively defined loss or cost function, $L(w, b)$, that quantifies the degree of satisfaction of the network to the problem, in the following terms:

$$L(w, b) = L_1(w, b) + L_2(w, b) + R(w, b), \tag{2.1}$$

where $L_1(w, b)$ is the loss term measuring how well the neural network approximates the differential operator (1.1), $L_2(w, b)$ is the loss term measuring how well the neural network approximates the initial/boundary conditions (1.2-1.5), and R(w,b) is the regularization term of the loss which is a term that will help to stabilize and improve the convergence in the optimization (this terms will be covered in a later section). In particular, using Cauchy boundary conditions, which are the most complex of all, the terms would be:

$$L_1(w, b) = \Big|\Big|\mathcal{L}[\hat{u}(x; w, b)] - f(x)\Big|\Big|_{\Omega,2} = \int_\Omega \Big(\mathcal{L}[u(x; w, b)] - f(x)\Big)^2 dx, \tag{2.2}$$

$$\begin{aligned} L_2(w, b) &= \Big|\Big|\hat{u}(x; w, b) - g_1(x)\Big|\Big|_{\Gamma,2} + \Big|\Big|\frac{\partial \hat{u}(x; w, b)}{\partial n(x)}] - g_2(x)\Big|\Big|_{\Gamma,2} \\ &= \int_\Gamma \Big(\hat{u}(x; w, b) - g_1(x)\Big)^2 dx + \int_\Gamma \Big(\frac{\partial \hat{u}(x; w, b)}{\partial n(x)} - g_2(x)\Big)^2 dx, \end{aligned} \tag{2.3}$$

with $||\cdot||_{\Omega,2}$ the norm of the $L^2(\Omega)$ Hilbert space (which is the space of square integrable functions in Ω), and the same concept applies to $||\cdot||_{\Gamma,2}$. However, in practice, as the integrals in (2.2-2.3) are virtually impractical to compute, instead of using the $||\cdot||_{\Omega,2}$ and $||\cdot||_{\Gamma,2}$ norms, a discrete approximation is used. The is achieved is by taking a random collocation of N_Ω points in Ω and N_Γ points in Γ, which can be obtained by using a Monte Carlo hit-and-miss approach, and discretizing as $\int_\Omega \to 1/N_\Omega \sum_{N_\Omega}$ and $\int_\Omega \to 1/N_\Gamma \sum_{N_\Gamma}$. Thus the actual loss terms become:

$$L_1(w, b) \approx \frac{1}{N_\Omega} \sum_{i \in N_\Omega} \Big(\mathcal{L}[u(x_i; w, b)] - f(x_i)\Big)^2, \tag{2.4}$$

$$L_2(w, b) \approx \frac{1}{N_\Gamma} \sum_{i \in N_\Gamma} \Big(\hat{u}(x_i; w, b) - g_1(x_i)\Big)^2 + \frac{1}{N_\Gamma} \sum_{i \in N_\Gamma} \Big(\frac{\partial \hat{u}(x_i; w, b)}{\partial n(x_i)} - g_2(x_i)\Big)^2. \tag{2.5}$$

Observe that we have transformed the continuous loss functions into the MSE (mean squared error) on a random collocation of points of the domain and the border. Now, on these premises, the problem has changed in nature, from a numerical integration problem, to an almost purely deep-learning regression type of problem. Other discretizations using the absolute error or the Huber error would have yielded equally valid approximations.

The next steps are straightforward, train the artificial neural network as done in any other deep-learning regression type of problem. This entails, taking the loss $L(w, b)$ which defines a hyper-surface in the parameter space, and given a certain initialization of parameters (w_0, b_0), use a gradient-based optimization technique to minimize the loss function which is equivalent to minimizing the error of the approximation.

2.3 Derivatives: Back Propagation and Gradient Issues

As we have just stated we would like to use some kind of gradient-based optimization technique to optimize $L(w, b)$. The idea of this kind of techniques is to start at some point and move that point in steps along the direction of the gradient until we reach the minimum or sufficiently low value of the loss function. This strategy implies that in every step we are required to compute the gradient $\nabla_{(w,b)} L(w_0, b_0)$ at the point we are in.

Using a feed-forward scheme allows for the application of "standard" back-propagation algorithms to compute the derivatives of these gradients, which other schemes do not allow, or require of many modifications. In actuality, back-propagation is the name that receives in deep-learning a backward accumulation automatic differentiation algorithm (or autodiff), which is a type of dynamic programming algorithm used to compute derivatives of composed functions efficiently in computers. [26]

We will illustrate how a backward autodiff works with the following example, imagine we wanted to calculate in a computer the derivative in (x_0,y_0) of:

$$f(x, y) = \sin(e^x + xe^y). \tag{2.6}$$

What an autodiff algorithm does is decompose progressively the function into primitive operations (sums, subtractions, multiplications, divisions and basic functions: exponentials, sinus, cosines...). A computer can take any of this primitive operations and compute their values and the values of its derivatives at any point with arithmetic precision. For (2.6), the decomposition would be:

$$\begin{aligned}
z_1 = \sin(z_2) &\longrightarrow \frac{\partial z_1}{\partial z_2} = \cos(z_2), & z_5 = x &\longrightarrow \frac{\partial z_5}{\partial x} = 1 \\
z_2 = z_3 + z_4 &\longrightarrow \frac{\partial z_2}{\partial z_3} = 1,\ \frac{\partial z_2}{\partial z_4} = 1 & z_6 = x &\longrightarrow \frac{\partial z_6}{\partial x} = 1 \\
z_3 = e^{z_5} &\longrightarrow \frac{\partial z_3}{\partial z_5} = e^{z_5}, & z_7 = e^{z_8} &\longrightarrow \frac{\partial z_7}{\partial z_8} = e^{z_8} \\
z_4 = z_6 \cdot z_7 &\longrightarrow \frac{\partial z_4}{\partial z_6} = z_7,\ \frac{\partial z_4}{\partial z_5} = z_6, & z_8 = y &\longrightarrow \frac{\partial z_8}{\partial y} = 1.
\end{aligned} \tag{2.7}$$

From a implementation stance, this decomposition is used to build what is known as a computational graph, which is a directed graph such that, its nodes store the computed numerical values of the primitive operations z_i and their derivatives at the point of calculation (x_0,y_0), and its directed edges store the precedence of all operations. Figures (2.4-2.4) show the computational graph of (2.6). In Tensorflow v2.3 which is the deep-learning framework that we will be using in the next chapter to build artificial neural networks, computational graphs are natively implemented in a class called *tape*, which automatically builds these graphs.

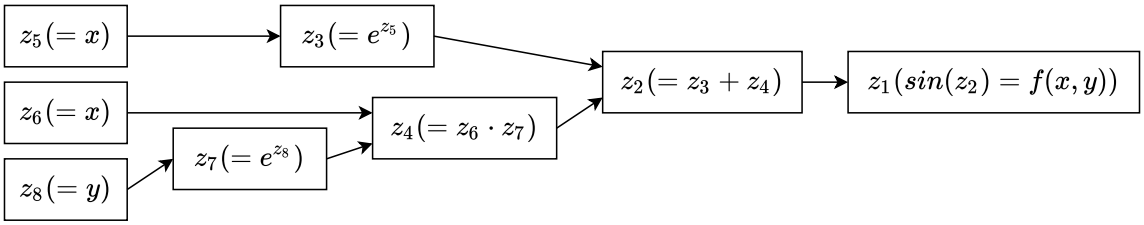

Figure 2.4: Computational graph of example (2.6).

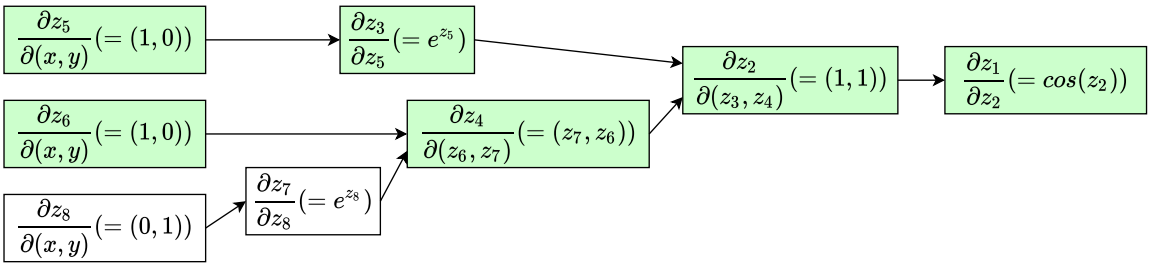

Figure 2.5: Computational graph (derivatives) of example (2.6). In green the flow of nodes required to compute $\partial f(x,y)/\partial x$

The computation of the derivatives at the point (x_0,y_0) is done by multiplying and adding (when more than one edge enters a node) the values of derivatives (actually, the Jacobians) at the nodes following the directions of the edges. This algorithm in its backward version is exactly the same as successively applying the chain rule in order, from the most fundamental operation to the most composed one, in a systematic way. Its forward version applies the chain rule in the other order. For (2.6) the algorithm would be the same as using the chain rule as follows:

$$\begin{aligned}\frac{\partial f(x,y)}{\partial(x,y)} &= \left(\left(\frac{\partial z_1}{\partial z_2}\cdot\frac{\partial z_2}{\partial z_3}\right)\cdot\frac{\partial z_3}{\partial z_5}\right)\cdot\frac{\partial z_5}{\partial(x,y)} \\ &+ \left(\left(\frac{\partial z_1}{\partial z_2}\cdot\frac{\partial z_2}{\partial z_4}\right)\cdot\frac{\partial z_4}{\partial z_6}\right)\cdot\frac{\partial z_6}{\partial(x,y)} \\ &+ \left(\left(\left(\frac{\partial z_1}{\partial z_2}\cdot\frac{\partial z_2}{\partial z_4}\right)\cdot\frac{\partial z_4}{\partial z_7}\right)\cdot\frac{\partial z_7}{\partial z_8}\right)\cdot\frac{\partial z_8}{\partial(x,y)}.\end{aligned} \tag{2.8}$$

The dynamic programming element to this algorithm is very helpful as it allows to use the same computation graph of a given point to compute with no additional cost the derivatives with respect to intermediate operations and parameters (which will be extremely useful). It also provides better errors and stability than numerical differentiation which approximates derivatives by differences. This is also different from symbolic differentiation. [27]

2.3.1 Derivatives Behaviour (Vanishing and Exploding Gradients)

In this subsection we will compute the derivatives of an example artificial neural network and look at how they behave. Although by using the back propagation algorithm we are able to numerically compute derivatives up to any order with great precision by successively applying the algorithm to its outputted derivatives, this does not imply that the derivatives we compute (in a structural sense) are well suited to train the model. Differences in magnitude and sensitivity to variations of the parameters can make training almost impossible.

When computing the gradient of the loss function with respect to the parameters to train the artificial neural network, we obtain the following:

$$\nabla_{(w,b)} L(w,b) := \frac{\partial L(w,b)}{\partial (w,b)} = \frac{\partial L_1(w,b)}{\partial (w,b)} + \frac{\partial L_2(w,b)}{\partial (w,b)} + \frac{\partial R(w,b)}{\partial (w,b)}, \tag{2.9}$$

where, in particular,

$$\frac{\partial L_1(w,b)}{\partial (w,b)} = \frac{\partial \mathcal{L}[\widehat{u}(x;w,b)]}{\partial (w,b)} - \frac{\partial f(x)}{\partial (w,b)}. \tag{2.10}$$

and thus, we observe from the term in L_1 that, in order to obtain the gradient, we will require the derivatives of up to first order for (w,b), and up to the highest order in the differential operator $\mathcal{L}$ for (x), which in this work can be of up to order 2 from the Laplacian operator.

Next, we will use an example artificial neural network consisting of 4 layers: an input layer of 2 neurons, two hidden layers of 3 and 4 neurons respectively, and an output layer of 2 neurons (see Figure 2.6). Using this example for greater clarity, we will first characterize its elements in detail, which will serve as a consolidation example of the first section of this chapter. Then, we will be applying the chain rule to express some derivatives of up to order 2 in variables and order 1 in parameters, and we will discuss their potential issues when used in gradient based training, namely the issues with vanishing and exploding gradients. This whole subsection will be written in the tensor formulation introduced in Chapter 0. In Appendix A, we reproduce the results in this subsection that make sense in linear algebra notation, for didactic reasons.

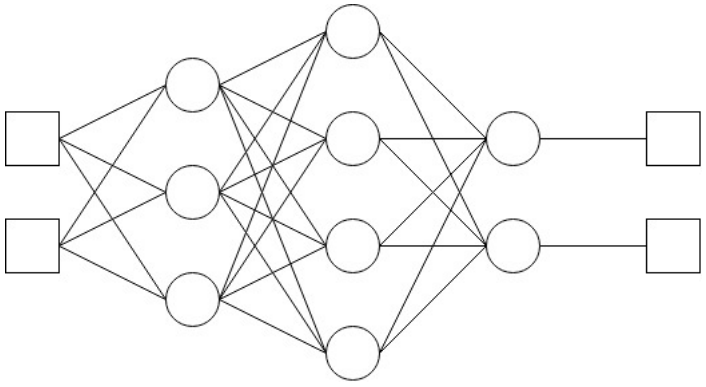

Figure 2.6: Example model: A 2-3-4-2 artificial neural network.

Given our example artificial neural network (Figure 2.6), the elements that characterize it (variables, parameters and activation functions), in the most general way for perceptron type neurons, are:

$$n_0, m_0 = 1,2, \quad n_1, m_1 = 1,2,3, \quad n_2, m_2 = 1,2,3,4, \quad n_3, m_3 = 1,2,$$

$$\begin{aligned} &w^{[1]}{}_{n_1}^{m_0},\ w^{[2]}{}_{n_2}^{m_1},\ w^{[3]}{}_{n_3}^{m_2}, && z^{[0]}{}_{n_0} = x_{n_0},\ z^{[1]}{}_{n_1},\ z^{[2]}{}_{n_2},\ z^{[3]}{}_{n_3} = \widehat{u}_{n_3}, \\ &b^{[1]}{}_{n_1},\ b^{[1]}{}_{n_2},\ b^{[1]}{}_{n_3}, && y^{[0]}{}_{n_0} = x_{n_0},\ y^{[1]}{}_{n_1},\ y^{[2]}{}_{n_2},\ y^{[3]}{}_{n_3} = \widehat{u}_{n_3}, \\ &a^{[1]}{}_{n_1},\ a^{[2]}{}_{n_2} : \mathbb{R} \longrightarrow \mathbb{R}. \end{aligned} \tag{2.11}$$

Recall that each of this elements is a tensor in itself.

In particular, for each of the layers ℓ, the output variables ${y^{[\ell]}}_{n_\ell}$ and ${z^{[\ell]}}_{n_\ell}$ of each neuron are (where remember that from now on everything is written in Einstein convention notation):

$$\begin{aligned} {z^{[1]}}_{n_1} &= {b^{[1]}}_{n_1} + {w^{[1]}}_{n_1}^{\ n_0} \cdot {y^{[0]}}_{n_0} \ , & {y^{[1]}}_{n_1} &= {a^{[1]}}_{n_1}\left({z^{[1]}}_{n_1}\right), \\ {z^{[2]}}_{n_2} &= {b^{[2]}}_{n_2} + {w^{[2]}}_{n_2}^{\ n_1} \cdot {y^{[1]}}_{n_1} \ , & {y^{[2]}}_{n_2} &= {a^{[2]}}_{n_2}\left({z^{[2]}}_{n_2}\right), \\ {z^{[3]}}_{n_3} &= {b^{[3]}}_{n_3} + {w^{[3]}}_{n_3}^{\ n_2} \cdot {y^{[2]}}_{n_2} \ , & {y^{[3]}}_{n_3} &= {z^{[3]}}_{n_3}. \end{aligned} \tag{2.12}$$

If we were compose all these output variables we can see that this instance of an artificial neural network actually represents the composite function:

$${y^{[3]}}_{n_3} = {b^{[3]}}_{n_3} + {w^{[3]}}_{n_3}^{\ n_2} \cdot {a^{[2]}}_{n_2}\left({b^{[2]}}_{n_2} + {w^{[2]}}_{n_2}^{\ n_1} \cdot {a^{[1]}}_{n_1}\left({b^{[1]}}_{n_1} + {w^{[1]}}_{n_1}^{\ n_0} \cdot {y^{[0]}}_{n_0}\right)\right), \tag{2.13}$$

or equivalently,

$$\hat{u}_{n_3} = {b^{[3]}}_{n_3} + {w^{[3]}}_{n_3}^{\ n_2} \cdot {a^{[2]}}_{n_2}\left({b^{[2]}}_{n_2} + {w^{[2]}}_{n_2}^{\ n_1} \cdot {a^{[1]}}_{n_1}\left({b^{[1]}}_{n_1} + {w^{[1]}}_{n_1}^{\ n_0} \cdot x_{n_0}\right)\right). \tag{2.14}$$

Now, we will compute some derivatives applying the chain rule:

Derivatives of order 1 in x:

$$\begin{aligned} \frac{\partial \hat{u}_{n_3}}{\partial x_{m_0}} &:= \frac{\partial {y^{[3]}}_{n_3}}{\partial {z^{[0]}}_{n_0}} \\ &= \frac{\partial {y^{[3]}}_{n_3}}{\partial {z^{[3]}}_{m_3}} \cdot \frac{\partial {z^{[3]}}_{m_3}}{\partial {y^{[2]}}_{n_2}} \cdot \frac{\partial {y^{[2]}}_{n_2}}{\partial {z^{[2]}}_{m_2}} \cdot \frac{\partial {z^{[2]}}_{m_2}}{\partial {y^{[1]}}_{n_1}} \cdot \frac{\partial {y^{[1]}}_{n_1}}{\partial {z^{[1]}}_{m_1}} \cdot \frac{\partial {z^{[1]}}_{m_1}}{\partial {y^{[0]}}_{n_0}} \cdot \frac{\partial {y^{[0]}}_{n_0}}{\partial {z^{[0]}}_{m_0}} \\ &= \delta^{m_3}_{n_3} \cdot {w^{[3]}}_{n_3}^{\ n_2} \cdot D^{n_2}\left({a^{[2]}}_{n_2}({z^{[2]}}_{n_2})\right) \cdot \delta^{m_2}_{n_2} \cdot {w^{[2]}}_{m_2}^{\ n_1} \cdot D^{n_1}\left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot \delta^{m_1}_{n_1} \cdot {w^{[1]}}_{m_1}^{\ n_0} \cdot \delta^{m_0}_{n_0} \\ &= {w^{[3]}}_{n_3}^{\ n_2} \cdot D^{n_2}\left({a^{[2]}}_{n_2}({z^{[2]}}_{n_2})\right) \cdot {w^{[2]}}_{n_2}^{\ n_1} \cdot D^{n_1}\left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot {w^{[1]}}_{n_1}^{\ m_0}. \end{aligned} \tag{2.15}$$

Derivatives of order 2 in x:

$$\begin{aligned} \frac{\partial^2 \hat{u}_{n_3}}{\partial x_{\overline{m}_0} \partial x_{m_0}} &:= \frac{\partial}{\partial {z^{[0]}}_{\overline{m}_0}} \left(\frac{\partial \hat{u}_{n_3}}{\partial x_{m_0}}\right) \\ &= {w^{[3]}}_{n_3}^{\ n_2} \cdot \frac{\partial}{\partial {z^{[0]}}_{\overline{m}_0}} D^{n_2}\left({a^{[2]}}_{n_2}({z^{[2]}}_{n_2})\right) \cdot {w^{[2]}}_{n_2}^{\ n_1} \cdot D^{n_1}\left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot {w^{[1]}}_{n_1}^{\ m_0} \\ &\quad + {w^{[3]}}_{n_3}^{\ n_2} \cdot D^{n_2}\left({a^{[2]}}_{n_2}({z^{[2]}}_{n_2})\right) \cdot {w^{[2]}}_{n_2}^{\ n_1} \cdot \frac{\partial}{\partial {z^{[0]}}_{\overline{m}_0}} D^{n_1}\left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot {w^{[1]}}_{n_1}^{\ m_0} \\ &= {w^{[3]}}_{n_3}^{\ n_2} \cdot D^{n_2,n_2}\left({a^{[2]}}_{n_2}({z^{[2]}}_{n_2})\right) \\ &\quad \cdot \left({w^{[2]}}_{n_2}^{\ \overline{n}_1} \cdot D^{\overline{n}_1}\left({a^{[1]}}_{\overline{n}_1}({z^{[1]}}_{\overline{n}_1})\right) \cdot {w^{[1]}}_{\overline{n}_1}^{\ \overline{m}_0} \right) \cdot \left({w^{[2]}}_{n_2}^{\ n_1} \cdot D^{n_1}\left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot {w^{[1]}}_{n_1}^{\ m_0} \right) \\ &\quad + {w^{[3]}}_{n_3}^{\ n_2} \cdot D^{n_2}\left({a^{[2]}}_{n_2}({z^{[2]}}_{n_2})\right) \cdot {w^{[2]}}_{n_2}^{\ n_1} \cdot D^{n_1,n_1}\left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot {w^{[1]}}_{n_1}^{\ \overline{m}_0} \cdot {w^{[1]}}_{n_1}^{\ m_0}. \end{aligned} \tag{2.16}$$

Derivatives of order 0 in x and order 1 in w:

$$\begin{aligned}
\frac{\partial \hat{u}_{n_3}}{\partial w^{[3]}{}_{m_3}^{n_2}} &= \delta_{n_3}^{m_3} \cdot y^{[2]}{}_{n_2}, \\
\frac{\partial \hat{u}_{n_3}}{\partial w^{[2]}{}_{m_2}^{n_1}} &= w^{[3]}{}_{n_3}^{n_2} \cdot D^{n_2}\left(a^{[2]}{}_{n_2}(z^{[2]}{}_{n_2})\right) \cdot \delta_{n_2}^{m_2} \cdot y^{[1]}{}_{n_1}, \\
\frac{\partial \hat{u}_{n_3}}{\partial w^{[1]}{}_{m_1}^{n_0}} &:= \frac{\partial y^{[3]}{}_{n_3}}{\partial w^{[1]}{}_{m_1}^{n_0}} \\
&= \frac{\partial y^{[3]}{}_{n_3}}{\partial z^{[3]}{}_{m_3}} \cdot \frac{\partial z^{[3]}{}_{m_3}}{\partial y^{[2]}{}_{n_2}} \cdot \frac{\partial y^{[2]}{}_{n_2}}{\partial z^{[2]}{}_{m_2}} \cdot \frac{\partial z^{[2]}{}_{m_2}}{\partial y^{[1]}{}_{n_1}} \cdot \frac{\partial y^{[1]}{}_{n_1}}{\partial z^{[1]}{}_{m_1}} \cdot \frac{\partial z^{[1]}{}_{m_1}}{\partial w^{[1]}{}_{m_1}^{n_0}} \\
= \delta_{n_3}^{m_3} \cdot w^{[3]}{}_{m_3}^{n_2} \cdot D^{n_2}\left(a^{[2]}{}_{n_2}(z^{[2]}{}_{n_2})\right) \cdot \delta_{n_2}^{m_2} \cdot w^{[2]}{}_{m_2}^{n_1} \cdot D^{n_1}\left(a^{[1]}{}_{n_1}(z^{[1]}{}_{n_1})\right) \cdot \delta_{n_1}^{m_1} \cdot y^{[0]}{}_{n_0} \\
= w^{[3]}{}_{n_3}^{n_2} \cdot D^{n_2}\left(a^{[2]}{}_{n_2}(z^{[2]}{}_{n_2})\right) \cdot w^{[2]}{}_{n_2}^{n_1} \cdot D^{n_1}\left(a^{[1]}{}_{n_1}(z^{[1]}{}_{n_1})\right) \cdot \delta_{n_1}^{m_1} \cdot y^{[0]}{}_{n_0}.
\end{aligned} \tag{2.17}$$

Derivatives of order 0 in x and order 1 in b:

$$\begin{aligned}
\frac{\partial \hat{u}_{n_3}}{\partial b^{[3]}{}_{m_3}} &= \delta_{n_3}^{m_3}, \\
\frac{\partial \hat{u}_{n_3}}{\partial b^{[2]}{}_{m_2}} &= w^{[3]}{}_{n_3}^{n_2} \cdot D^{n_2}\left(a^{[2]}{}_{n_2}(z^{[2]}{}_{n_2})\right) \cdot \delta_{n_2}^{m_2}, \\
\frac{\partial \hat{u}_{n_3}}{\partial b^{[1]}{}_{m_1}} &:= \frac{\partial y^{[3]}{}_{n_3}}{\partial b^{[1]}{}_{m_1}} \\
&= \frac{\partial y^{[3]}{}_{n_3}}{\partial z^{[3]}{}_{m_3}} \cdot \frac{\partial z^{[3]}{}_{m_3}}{\partial y^{[2]}{}_{n_2}} \cdot \frac{\partial y^{[2]}{}_{n_2}}{\partial z^{[2]}{}_{m_2}} \cdot \frac{\partial z^{[2]}{}_{m_2}}{\partial y^{[1]}{}_{n_1}} \cdot \frac{\partial y^{[1]}{}_{n_1}}{\partial z^{[1]}{}_{m_1}} \cdot \frac{\partial z^{[1]}{}_{m_1}}{\partial b^{[1]}{}_{m_1}} \\
= \delta_{n_3}^{m_3} \cdot w^{[3]}{}_{m_3}^{n_2} \cdot D^{n_2}\left(a^{[2]}{}_{n_2}(z^{[2]}{}_{n_2})\right) \cdot \delta_{n_2}^{m_2} \cdot w^{[2]}{}_{m_2}^{n_1} \cdot D^{n_1}\left(a^{[1]}{}_{n_1}(z^{[1]}{}_{n_1})\right) \cdot \delta_{n_1}^{m_1} \\
= w^{[3]}{}_{n_3}^{n_2} \cdot D^{n_2}\left(a^{[2]}{}_{n_2}(z^{[2]}{}_{n_2})\right) \cdot w^{[2]}{}_{n_2}^{n_1} \cdot D^{n_1}\left(a^{[1]}{}_{n_1}(z^{[1]}{}_{n_1})\right) \cdot \delta_{n_1}^{m_1}.
\end{aligned} \tag{2.18}$$

Derivatives of order 1 in x and order 1 in $w^{[1]}$:

$$\begin{aligned}
\frac{\partial^2 \hat{u}_{n_3}}{\partial w^{[1]}{}_{\overline{m}_1}^{\overline{n}_0} \partial x_{m_0}} &:= \frac{\partial}{\partial w^{[1]}{}_{\overline{m}_1}^{\overline{n}_0}} \left(\frac{\partial \hat{u}_{n_3}}{\partial x_{m_0}} \right) \\
= w^{[3]}{}_{n_3}^{n_2} \cdot D^{n_2,n_2}\left(a^{[2]}{}_{n_2}(z^{[2]}{}_{n_2})\right) \\
\cdot \Big(w^{[2]}{}_{n_2}^{\overline{n}_1} \cdot D^{\overline{n}_1}\left(a^{[1]}{}_{\overline{n}_1}(z^{[1]}{}_{\overline{n}_1})\right) \cdot \delta_{\overline{n}_1}^{\overline{m}_1} \cdot y^{[0]}{}_{\overline{n}_0} \Big) \cdot \left(w^{[2]}{}_{n_2}^{n_1} \cdot D^{n_1}\left(a^{[1]}{}_{n_1}(z^{[1]}{}_{n_1})\right) \cdot w^{[1]}{}_{n_1}^{m_0} \right) \\
+ w^{[3]}{}_{n_3}^{n_2} \cdot D^{n_2}\left(a^{[2]}{}_{n_2}(z^{[2]}{}_{n_2})\right) \cdot w^{[2]}{}_{n_2}^{n_1} \cdot D^{n_1,n_1}\left(a^{[1]}{}_{n_1}(z^{[1]}{}_{n_1})\right) \cdot \delta_{n_1}^{\overline{m}_1} \cdot y^{[0]}{}_{\overline{n}_0} \cdot w^{[1]}{}_{n_1}^{m_0} \\
+ w^{[3]}{}_{n_3}^{n_2} \cdot D^{n_2}\left(a^{[2]}{}_{n_2}(z^{[2]}{}_{n_2})\right) \cdot w^{[2]}{}_{n_2}^{n_1} \cdot D^{n_1}\left(a^{[1]}{}_{n_1}(z^{[1]}{}_{n_1})\right) \cdot \delta_{\overline{n}_0}^{m_0} \cdot \delta_{n_1}^{\overline{m}_1}.
\end{aligned} \tag{2.19}$$

The vanishing and exploding gradient problems are a recurrent issue in the deep-learning field, and in fact, it was the main reason for which, although the concept artificial neural networks was well known since the 1950s, it was not until the early 2000s when the effect of this issues were "mitigated" with new techniques, that the field of deep-learning exploded became what it is today. The solution up until now has been to design improved optimizers, use regularization techniques, increase computer power and make use of parallel computation, and create new architectures that preserve data structures (such as convolutional networks in image processing). In the following sections we will see the ideas for the typical optimizers and regularizations that are used to avoid the "classical" vanishing and exploding gradients.

From the derivatives with respect to w and b in (2.17) and (2.18), these gradient problems can be straight forward explained. Observe that in the derivatives of the parameters, as we go from the deepest (closest to the input) layer to the shallowest layers (closest to the output) of the network, the number of terms and derivatives of the activation functions increase. Thus, when the weights w become small (or large) with respect to 1, or the activation function has a small (or large) slope in the range of the linear combination of the neurons z, both of which are extremely common, then as we consider the derivatives of deeper parameters, these become the product of an increasing number of small (or large) terms. In conclusion, as we consider deeper layers, the derivatives used to train the parameters become vanishingly small or explosively large with respect to the shallower layers, making the optimization of these deeper parameters almost impossible. Given our problem (only in w, but equivalent in b):

$$\begin{aligned} &\text{"Calssic" Vanishing Gradient: } \left|\frac{\partial \hat{u}_{n_3}}{\partial w^{[3]\ n_2}_{\ \ m_3}}\right| >> \left|\frac{\partial \hat{u}_{n_3}}{\partial w^{[2]\ n_1}_{\ \ m_2}}\right| >> \left|\frac{\partial \hat{u}_{n_3}}{\partial w^{[1]\ n_0}_{\ \ m_1}}\right|. \\ &\text{"Calssic" Exploding Gradient: } \left|\frac{\partial \hat{u}_{n_3}}{\partial w^{[3]\ n_2}_{\ \ m_3}}\right| << \left|\frac{\partial \hat{u}_{n_3}}{\partial w^{[2]\ n_1}_{\ \ m_2}}\right| << \left|\frac{\partial \hat{u}_{n_3}}{\partial w^{[1]\ n_0}_{\ \ m_1}}\right|. \end{aligned} \tag{2.20}$$

At this point, we will refer to the vanishing and exploding gradient problems that we have just explained as "classical" because it is the one that always takes place in any deep-learning classification and regression problem. However, by the very nature of this problem, being that the loss function contains derivatives of artificial neural network in x, we incur in a second type of vanishing and exploding gradient problems which is harder to see. If the classical vanishing/explode type implies that the derivatives of the artificial neural network $\hat{u}$ with respect to the parameters w and b increase/decrease as we consider parameters in deeper layers; in this second type, for higher order derivatives of $\hat{u}$ in x, their derivatives with respect to the same parameter of the same layer also vanishes or explodes as we consider higher derivatives in x. Given our problem (only in w, but equivalent in b):

$$\begin{aligned} &\text{"Second type" Vanishing Gradient: } \left|\frac{\partial \hat{u}_{n_3}}{\partial w^{[\ell]\ n_2}_{\ \ m_3}}\right| >> \left|\frac{\partial^2 \hat{u}_{n_3}}{\partial w^{[\ell]\ n_1}_{\ \ m_2} \partial x_{\overline{m}_0}}\right| >> \dots . \\ &\text{"Second type" Exploding Gradient: } \left|\frac{\partial \hat{u}_{n_3}}{\partial w^{[\ell]\ n_2}_{\ \ m_3}}\right| << \left|\frac{\partial^2 \hat{u}_{n_3}}{\partial w^{[\ell]\ n_1}_{\ \ m_2} \partial x_{\overline{m}_0}}\right| << \dots . \end{aligned} \tag{2.21}$$

Showing this second type of gradient problem a bit less obvious than the classic one. To be able to give an intuition of the problem we will consider the case in which all the activation functions are exponential ${a^{[\ell]}}_{n_\ell}(x) = e^x$. In such case the derivatives of the activation function are simply:

$$\begin{aligned} D^{n_\ell}\left({a^{[\ell]}}_{n_\ell}({z^{[\ell]}}_{n_\ell})\right) &= {a^{[\ell]}}_{n_\ell}({z^{[\ell]}}_{n_\ell}) \cdot \mathbb{1}^{n_\ell}, \\ D^{n_\ell,n_\ell}\left({a^{[\ell]}}_{n_\ell}({z^{[\ell]}}_{n_\ell})\right) &= {a^{[\ell]}}_{n_\ell}({z^{[\ell]}}_{n_\ell}) \cdot \mathbb{1}^{n_\ell,n_\ell}, \end{aligned} \tag{2.22}$$

where $\mathbb{1}^{n_\ell}$ is the tensor whose every component is 1, and $\mathbb{1}^{n_\ell,n_\ell} = \mathbb{1}^{n_\ell} \cdot \mathbb{1}^{n_\ell}$. Then, for an exponential activation function, the derivatives (2.17) and (2.19) of the weights $w^{[1]}$ in the first layer become:

$$\frac{\partial \hat{u}_{n_3}}{\partial {w^{[1]}}_{\substack{n_0 \\ m_1}}} = {w^{[3]}}_{\substack{n_2 \\ n_3}} \cdot \left({a^{[2]}}_{n_2}({z^{[2]}}_{n_2})\right) \cdot \mathbb{1}^{n_2} \cdot {w^{[2]}}_{\substack{n_1 \\ n_2}} \cdot \left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot \mathbb{1}^{n_1} \cdot \delta^{m_1}_{n_1} \cdot {y^{[0]}}_{n_0}, \tag{2.23}$$

$$\begin{aligned} \frac{\partial^2 \hat{u}_{n_3}}{\partial {w^{[1]}}_{\substack{\overline{n}_0 \\ \overline{m}_1}} \partial x_{m_0}} &= {w^{[3]}}_{\substack{n_2 \\ n_3}} \cdot \left({a^{[2]}}_{n_2}({z^{[2]}}_{n_2})\right) \cdot \mathbb{1}^{n_2,n_2} \\ &\cdot \Big({w^{[2]}}_{\substack{\overline{n}_1 \\ n_2}} \cdot \left({a^{[1]}}_{\overline{n}_1}({z^{[1]}}_{\overline{n}_1})\right) \cdot \mathbb{1}^{\overline{n}_1} \cdot \delta^{\overline{m}_1}_{\overline{n}_1} \cdot {y^{[0]}}_{\overline{n}_0} \Big) \cdot \left({w^{[2]}}_{\substack{n_1 \\ n_2}} \cdot \left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot \mathbb{1}^{n_1} \cdot {w^{[1]}}_{\substack{m_0 \\ n_1}} \right) \\ &+ {w^{[3]}}_{\substack{n_2 \\ n_3}} \cdot \left({a^{[2]}}_{n_2}({z^{[2]}}_{n_2})\right) \cdot \mathbb{1}^{n_2} \cdot {w^{[2]}}_{\substack{n_1 \\ n_2}} \cdot \left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot \mathbb{1}^{n_1,n_1} \cdot \delta^{\overline{m}_1}_{n_1} \cdot {y^{[0]}}_{\overline{n}_0} \cdot {w^{[1]}}_{\substack{m_0 \\ n_1}} \\ &+ {w^{[3]}}_{\substack{n_2 \\ n_3}} \cdot \left({a^{[2]}}_{n_2}({z^{[2]}}_{n_2})\right) \cdot \mathbb{1}^{n_2} \cdot {w^{[2]}}_{\substack{n_1 \\ n_2}} \cdot \left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot \mathbb{1}^{n_1} \cdot \delta^{m_0}_{\overline{n}_0} \cdot \delta^{\overline{m}_1}_{n_1}, \end{aligned} \tag{2.24}$$

Since the derivative of the exponential is essentially itself, and the activation function tensor is made of copies of exponentials for which differentiating means multiplying $\mathbb{1}^{n_\ell}$ tensors, both expressions (2.23) and (2.24) are written in the same terms, hence are easy to compare. As the tensor operations (sums and products) are commutative; and given a tensor $T^{i,j}_k$, which lets say has non-zero components for simplicity, we can define a "pseudo-inverse" to the contraction $(T^{i,j}_k)^{-1} = [T^{-1}]^k_{i,j}$ (element-wise) such that it holds $T^{i,j}_k \cdot [T^{-1}]^k_{i,j} = i \cdot j \cdot k$ (if there are zero values, we would have we would have to fix an inverse element for the zero components and discount the zeroes from the count $i \cdot j \cdot k$; here we will assume there are no zeroes for clarity); then we can easily replace (2.23) in (2.24) yielding:

$$\begin{aligned} \frac{\partial^2 \hat{u}_{n_3}}{\partial {w^{[1]}}_{\substack{\overline{n}_0 \\ \overline{m}_1}} \partial x_{m_0}} = \; & \frac{\partial \hat{u}_{n_3}}{\partial {w^{[1]}}_{\substack{\overline{n}_0 \\ \overline{m}_1}}} \cdot \left(\mathbb{1}^{n_2} \cdot {w^{[2]}}_{\substack{n_1 \\ n_2}} \cdot \left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot \mathbb{1}^{n_1} \cdot {w^{[1]}}_{\substack{m_0 \\ n_1}} \right) \\ & + \frac{\partial \hat{u}_{n_3}}{\partial {w^{[1]}}_{\substack{\overline{n}_0 \\ \overline{m}_1}}} \cdot \mathbb{1}^{n_1} \cdot {w^{[1]}}_{\substack{m_0 \\ n_1}} + \frac{\partial \hat{u}_{n_3}}{\partial {w^{[1]}}_{\substack{\overline{n}_0 \\ \overline{m}_1}}} \cdot \delta^{m_0}_{\overline{n}_0} \cdot (\overline{n}_0)^{-1} \cdot \left({y^{[0]}}_{\overline{n}_0}\right)^{-1}, \end{aligned} \tag{2.25}$$

and grouping,

$$\begin{aligned} \frac{\partial^2 \hat{u}_{n_3}}{\partial {w^{[1]}}_{\substack{\overline{n}_0 \\ \overline{m}_1}} \partial x_{m_0}} = \; & \Big(\mathbb{1}^{n_2} \cdot {w^{[2]}}_{\substack{n_1 \\ n_2}} \cdot \left({a^{[1]}}_{n_1}({z^{[1]}}_{n_1})\right) \cdot \mathbb{1}^{n_1} \cdot {w^{[1]}}_{\substack{m_0 \\ n_1}} \\ & + \mathbb{1}^{n_1} \cdot {w^{[1]}}_{\substack{m_0 \\ n_1}} + \delta^{m_0}_{\overline{n}_0} \cdot (\overline{n}_0)^{-1} \cdot \left({y^{[0]}}_{\overline{n}_0}\right)^{-1} \Big) \cdot \frac{\partial \hat{u}_{n_3}}{\partial {w^{[1]}}_{\substack{\overline{n}_0 \\ \overline{m}_1}}}. \end{aligned} \tag{2.26}$$

From (2.26) we see that, in the case of exponential activation functions, we can write the derivatives of a given weight and input in terms of a lower order derivative in terms of the input (this can be done for any order of the input). Now, regarding the factor in parenthesis in (2.26), assuming that the input is normalized and we do not shuffle data: the third term behaves as a very large constant ($>> 1$) which increases with the input dimension; when the weights are small, the second term is the largest, thus since the weights are small, the whole factor is small ($<< 1$); and when the weights are large, it is the first term that dominates, making the whole factor very large ($>> 1$). This creates the vanishing/exploding gradient effect described previously in (2.21). Also, this same analysis can be carried out with respect to any other weight or bias parameter making a few changes, but in case of considering other activation functions, although the end conclusion is the same and can be empirically visualized, the analytic study becomes much harder.

This issue is important when considering operators with that contain derivatives of different order or products of derivatives, which is always ignored. For example, the laplacian operator which contain only additions of second order derivatives does not have this problem as the effect of no derivative vastly dominates the effect of another when deriving over the parameters, but the burger's operator presents it.

2.4 Optimizers

Recall that in section 2.2 we transformed the problem of approximating the solution of an initial/boundary problem into a non-convex optimization problem, whereby we had to find the minimum (or a sufficiently small value) of a loss function $L(w, b)$, defined by (2.1), (2.4) and (2.5). On that section we went on to anticipate that, in order to do that, we would be using a gradient based optimization technique (optimizer), which required the computation of the loss derivatives with respect to the parameters, i.e. the gradient $\nabla_{(w,b)}L(w, b)$. This had led to section 2.3 were we explained the algorithm of back propagation to compute such derivatives and the discussions of their potential problems, namely the vanishing/exploding gradients. Now everything is set up and it is finally time to get into the process of actually making the parameters of the artificial neural network approximate the solution (optimizing the loss function), which in deep-learning jargon is known as the training process.

The following discussion will be devoted to explaining the design of some of the most important gradient based optimizers used in deep-learning, and the ones we will be using in this work. These optimizers are the so called methods of steepest descent or methods of gradient descent, which are a family of methods used to solve general non-linear (convex or non-convex) unrestricted optimization problems. Intuitively, the idea behind these methods relies on thinking of the loss function as a hyper-surface $L(w, b) : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$, where $w \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$. Then, starting at some (w_0, b_0), initial point, the method goes on to calculate new points which should reduce the loss function value by moving within a certain rate, η, named the *learning rate*, in the direction of $\nabla_{(w,b)}L(w, b)$. The typical analogy for this idea is thinking of it as having a ball (initial point), and letting it roll downhill along the slope (the direction of the gradient) until it reaches the bottom.

As simple as these method look conceptuality, in practice it non that easy to reach the minimum. If we were to apply one of this methods to a linear or quadratic bowl loss function ($(ax+b)^2,\ a>0$), we are guaranteed that the gradient at any point would always point in the direction of the only existing minimum, thus given adequate learning rates, these methods would have perfect convergence. However, with almost every other loss function, the direction of steepest descent (gradient) does not necessarily point to the global minimum. Moreover, if the problem is non-convex, as all the ones we will be considering in this work, we are almost guaranteed that there are many local minimums, and the direction of steepest descent may lead the method to a local minimum and not the global one.

Another tricky issue for these methods is the presence of saddle or "saddle-like" regions of the loss function. These are regions for which we have very small derivatives of the gradient in certain directions, and very large in others. Visualizing these regions in the loss function, they resemble to, and thus are often called, "valleys". What happens in these areas is that, in the ball analogy, the ball start oscillating up and down along the valley's walls (directions of large value derivatives) but is unable to make any progress across the valley (directions of low value derivatives). When using these steepest descent, this "saddle-like" region effect, as well as the effect of not being unable to escape a local minimum, is often reflected in the method when the point and loss function start oscillating between the same two very similar values.

These are the main three problems with steepest descent: the gradient not pointing in the direction of the global minimum; getting trapped in a local minimum; and stagnating when passing through "saddle-like" regions of the loss function. In order to avoid or mitigate these issues as much as possible, there are also three measures that can be applied: choosing a good initialization (starting point); applying some regularization technique, which somewhat has the effect of smoothing the loss function; and adjusting "properly" the learning rate at each step. In the next sections we will be looking at the initialization (which is tightly related to the selection of activation function), and the regularization techniques. For the rest of this section we will see different designs of steepest descent methods which adjust the learning rates for every step based on different ideas. We will divide these designs into first order if they require only the gradient, and second order if they also require estimates of the curvature.

For an extensive qualitative survey on gradient based methods [28] has a good coverage; in particular, in Table I and Table II there is a very complete comparison among first and higher order methods respectively. Other non gradient based methods are quite rare, for instance, in [29] a bio-inspired approach is used: a population of artificial neural networks is generated using different weights and architectures (hyper-parameters); the networks get tested and ranked by complexity and performance; then, a new population is generated based on the best performing networks with small alterations; and the process gets repeated.

2.4.1 First Order Methods

As we have already explained, these methods only depend on the gradient. The idea behind being so many variations is to have the method correct its learning rate by keeping some kind of memory of the gradients at previous points (steps) to improve convergence [30]. Next, we will discuss this methods grouping them in the following categories, from least to most refined:

- Vanilla (No Learning Rate Correction)
- Momentum Learning Rate Correction
- Component Learning Rate Adaptation
- Momentum + Component Learning Rate Adaptation

Vanilla (No Learning Rate Correction)

This group is the simplest and easiest to implement. It is actually the plain idea we have just explained, thus at every new step $t+1$ we update the previous point with the formula:

$$(w_{t+1}, b_{t+1}) \longrightarrow (w_t, b_t) - \eta \nabla_{(w,b)} L(w_k, b_k). \tag{2.27}$$

Generally, if the batches of input data (here the random collocation of points in Ω) is large, computing $\nabla_{(w,b)} L(w_t, b_t)$ in every step can be computationally very expensive. Recall that the factors in the loss function are of the form $1/N \sum_{i=1}^{N}(...)$, which means that if N is large, in every step we have to compute a large sum of sub-gradients, $1/N \sum_{i=1}^{N} \nabla_{(w,b)}(...)$, which can be costly. The fix is to take what is known as an stochastic or on-line approach, this is, to divide the input data into partitions, $\{x_i\}_{i \in M_1}, ..., \{x_i\}_{i \in M_k}$, with $M_1 + ...M_k$, and compute the gradient in every step just for the data of one of those partitions. If the input data is well randomized into the partitions, and the partitions are rotated consistently at every step, the differences in the gradient from not using the whole batch should be evened out throughout the many steps. When partitions of more than one data input are used the method is called **mini-batch gradient descent**, when the partitions contain a single data input the method is called **stochastic gradient descent** (SGD), and when the full batch is used the method is simply called **gradient descent** (GD). Often times, no distinction is made between mini-batch and stochastic, and both get referred to as stochastic gradient descent.

This stochastic approach can be applied to all the variations that we will be seeing next. In this work, however, we will not be sampling very large input data batches and the artificial neural networks will not be very large either, so we will always take full-batch approaches.

In terms of usage one would think that these version being the least refined would also be the least used, but it is far from the truth. It is true that the learning rate, η, has to be manually adapted in every step, which requires much try-and-error experimentation. This is done by setting a **learning schedule**, which is the set of instructions on how to vary the learning rate (for example, one could be as follows: for the first 1000 steps use $\eta = 0.001$, then every 1000 steps reduce $\eta/10$). Nonetheless, in recent times, and specially for artificial neural networks with large amounts of parameters (something that happens in very deep percepton neural networks, or in convolutional networks by design), there have been many papers that claim plain SGD (or at most the momentum we will be seeing next) can outclass any other variation that we will see here. The strategy in these papers is to use an unusually large learning rate, which means moving too far in the direction of the gradient and straying from the optimal path of minimal loss value, in order to create an **annealing** effect [31]. This annealing effect is a direct parallel from its homonym in metallurgy. Using these very "long jumps" allows for greater mobility for the point we are at in the method, giving it the capacity to get over "walls" and explore the loss hyper-surface to get into a better region, before switching to the regular small learning rate strategy used to achieve convergence. This works the same way as heating a metal to allow for greater mobility of its molecules, and then letting them settle by cooling the metal. Adding noise to the gradient has been for a long time an extremely successful regularization technique in deep-learning problems following the same principle of annealing of adding some exploration component. However, this concept takes it further, the objective being achieving **superconvergence**, which happens when entering in a very good region where the method suffers a drastic drop in its loss value, and convergence can be obtained many orders of magnitude faster than with a standard approach. In [32] successive cycles of short and long learning rates are used to obtain superconvergence, and [33] develop an adapted version called **SGD with Entropy** following these same ideas.

As we will see the idea behind the next variations is to speed up the method by auto-adjusting the initial learning rate at every step. This implies less tweaking of the learning rates as the method will reduce it natively when the loss is worsening to stay in the right track, and increase it when the loss is improving to go faster. This also make these variations incompatible wil superconvergence, as in the first step where the loss worsens, the method will immediately damp the learning rate.

Momentum Learning Rate Correction

Adding momentum to correct the learning rate in GD is very old and one of the first improvements on GD, the idea being based on keeping the inertia. In the ball analogy, if a ball is located at certain point but was carrying some velocity in a some direction, at that point it would not stop cold and resume its movement following the steepest descent. The ball will combine its previous inertia with the movement defined by the slope it is in. This is the idea behind **classical momentum** (CM), where the effective change v_{t+1} at the step $t+1$ is not only given by the gradient at that point, but also by a certain proportion μ by the effective change of the previous step v_t:

$$\begin{aligned} v_{t+1} &\longrightarrow \mu v_t - \eta \nabla_{(w,b)} L(w_t, b_t), \\ (w_{t+1}, b_{t+1}) &\longrightarrow (w_t, b_t) + v_{t+1}. \end{aligned} \tag{2.28}$$

Intuitively, if we were are moving in a very consistent direction through the loss hypersurface, the inertial term from the previous step v_t adds to the gradient making larger jumps in that direction. Conversely if the direction suddenly changes, v_t dampens the jump as we might have overstepped into a bad area by taking to large of a jump in the previous step. A second variation of this idea is the **Nesterov's accelerated gradient** (NAG), which tend to yield better results than CM. The difference is that in NAV we look at the gradient not at the point we are in, but in a point projected ahead as if we had done a second jump in the previous step:

$$\begin{aligned} v_{t+1} &\longrightarrow \mu v_t - \eta \nabla_{(w,b)} L((w_t, b_t) + \mu v_t), \\ (w_{t+1}, b_{t+1}) &\longrightarrow (w_t, b_t) + v_{t+1}. \end{aligned} \tag{2.29}$$

This subsection is based on the article [34].

Component Learning Rate Adaptation

While momentum uses the information of the previous gradient to speed up or slow down the method when there is consistent or changing behaviour, it does little to move forward in saddle regions. Recall that this kind of regions occur when some derivatives are several order of magnitude larger than others, i.e. some components of the gradient are much larger than others, which can be caused by vanishing or exploding gradient problems. In these cases we cannot increase the global learning rate to take longer jumps in the flatter directions because this would also make the jumps longer in the steeper directions, which require shorter steps to not stray from the convergence path. Also, momentum cannot help either, as it only adds up on the previous gradient, which is still small for the flatter directions. The solution is to rescale the learning rate for each component in the gradient individually based on previous gradients. So, if there have been directions which have had consistently small derivatives, we want to take larger jumps just in those directions, and conversely for directions which have had consistently large derivatives, we want to make smaller jumps to not to overstep out of the convergence path.

The first method that we are going to review is the **Adaptative gradient Algorithm** (AdaGrad). In its original paper [35], the method is presented as follows:

$$\begin{aligned} G_t &= \textstyle\sum_{\tau=1}^{t} (\nabla_{(w,b)} L(w_\tau, b_\tau)) \cdot ((\nabla L_{(w,b)}(w_\tau, b_\tau))^\top \in R^{n+m \times n+m}, \\ (w_{t+1}, b_{t+1}) &\longrightarrow (w_t, b_t) - \eta \ (diag(G_t) + \varepsilon Id)^{-1/2} \ \nabla_{(w,b)} L(w_\tau, b_\tau). \end{aligned} \tag{2.30}$$

where G_t is the cumulative matrix of products of the past gradients, $diag(G_t)$ is the diagonal of such matrix, Id corresponds to the identity matrix, and ε is a small constant to avoid diving by zero. As the matrix G_t can be computed accumulatively and only its diagonal elements are used, we suggest rewriting the method in the following vectorized way:

$$\begin{aligned} \mathcal{G}_t &\longrightarrow \mathcal{G}_{t-1} + \nabla_{(w,b)} L(w_t, b_t) \odot \nabla_{(w,b)} L(w_t, b_t) \in R^{n+m}, \\ (w_{t+1}, b_{t+1}) &\longrightarrow (w_t, b_t) - \eta \ (\mathcal{G}_t + \varepsilon \ 1)^{-1/2} \odot \nabla_{(w,b)} L(w_\tau, b_\tau). \end{aligned} \tag{2.31}$$

where products, inverses and root are all element-wise, and 1 es the vector consisting of all ones. Observe that if some direction of the gradient has consistently small derivatives, the cumulative value of $\mathcal{G}_t$ will be small, and thus dividing by the square root of that value will increase the learning rate of direction (the inverse happens for components with large derivatives). This is sort of approximating the curvature in the principal directions by the values of its past gradients. However, this cumulative nature is this method's main problem, as we are constantly accumulating positive values, $\mathcal{G}_t$ becomes increasingly large at each step, and since we are constantly dividing the learning rate by it, the methods halts the progress and is unable to scape local minima as time passes.

An improvement of AdaGrad cames with **AdaDelta**, [36], which mitigates the effect of the strong decay in learning rates of AdaGrad. Instead of using the accumulated information of all the squared previous gradients, it uses an exponential decay moving average of the square values of the gradient. This is instead of $\mathcal{G}_t$, it uses $E[(\nabla_{(w,b)} L(w_t, b_t))^2]$:

$$\begin{aligned} E[(\nabla_{(w,b)} L(w_t, b_t))^2] = \ & \rho \ E[(\nabla_{(w,b)} L(w_{t-1}, b_{t-1}))^2] \\ & +(1-\rho) \ \nabla_{(w,b)} L(w_t, b_t) \odot \nabla_{(w,b)} L(w_t, b_t), \end{aligned} \tag{2.32}$$

where ρ is the decay rate of the moving average. On top of this change, the method also adds a second idea. Since dividing by the square root of (2.32) is sort of a very brute approximation of dividing by the local curvature, in an attempt to resemble a second order Newton method, a term approximating the slope is multiplied. This term is a moving average of the squares of previous increments, $E[(\Delta(w_t, b_t))^2]$ which uses the same decay rate as before:

$$\begin{aligned} E[(\Delta(w_t, b_t))^2] = \ & \rho \ E[(\Delta(w_{t-1}, b_{t-1}))^2] \\ & +(1-\rho) \ \Delta(w_t, b_t) \odot \Delta(w_t, b_t), \end{aligned} \tag{2.33}$$

then the final algorithm at each step t works as:

$$\begin{aligned} &\text{Compute (2.32)}, \\ &\Delta(w_t, b_t) \longrightarrow \frac{\sqrt{E[(\Delta(w_t, b_t))^2] + \varepsilon \ 1}}{\sqrt{E[(\nabla_{(w,b)} L(w_t, b_t))^2] + \varepsilon \ 1}}, \\ &\text{Compute (2.33)}, \\ &(w_t, b_t) \longrightarrow (w_{t-1}, b_{t-1}) + \Delta(w_t, b_t). \end{aligned} \tag{2.34}$$

As a general note on the equations posed for this method, all the operations in (2.32-2.34) have been element wise. Finally, observe that the increment $\Delta(w_t, b_t)$ in (2.34) has in its numerator a term that approximates the slope and in its denominator a term that approximates the curvature, which tries to replicate a structure $H(f)^{-1} \cdot \nabla f$ of a Newton method.

A parallel developed, very popular and much simpler method than AdaDelta to solve the fast damping of AdaGrad is **RMSProp**. This is an unpublished method proposed in a Coursera course by Geoffrey Hinton, in lecture 6.5. [37]. This method was thought as an adaptation of the **RProp** which is a method originally designed for full-batches, to be able to account for mini-batches. This RProp method does not take into account the magnitude of the derivatives in the gradient, and instead, only takes into consideration the sign of the derivatives. Each direction learning rate is increased slightly every time the sign of its corresponding derivative it preserved, and drastically decreased whenever the sign of the derivative changes, everything within a certain threshold. When working with mini-batches this method can have many problems, as some sub-gradient may change in sign for some derivative due to the characteristics of that specific mini-batch, and not because the method has entered into region with a different behaviour. For instance, if the last 9 out of 10 derivatives in a direction have been positive and the only one has been negative, we do not want to drastically reduce its learning rate. To fix this resilience RMSProp uses the following moving average:

$$\begin{aligned} E[(\nabla_{(w,b)} L(w_t, b_t))^2] = \; & 0.9 \; E[(\nabla_{(w,b)} L(w_{t-1}, b_{t-1}))^2] \\ & +0.1 \; \nabla_{(w,b)} L(w_t, b_t) \odot \nabla_{(w,b)} L(w_t, b_t), \\ (w_{t+1}, b_{t+1}) \longrightarrow (w_t, b_t) - \eta \; & (E[(\nabla_{(w,b)} L(w_t, b_t))^2] + \epsilon \; 1)^{-1/2} \odot \nabla_{(w,b)} L(w_\tau, b_\tau), \end{aligned} \tag{2.35}$$

where again all the operations are element-wise. Note that the directional adaptation of the gradient at the current step t is introduced with a factor of 0.1. This gives robustness to the method as it requires persistence in the change of a sign through several steps to change the behaviour of the method. RMSProp also works better than RProp with full-batch, due to this robustness.

Momentum + Component Learning Rate Adaptation

This last type of methods combine the ideas of momentum and component learning rate adaptation. Recall that momentum introduced information about the slope by preserving some of the gradient of the last step, and component adaptation rescaled the components of the gradient in each direction dividing by the square root of the square of the gradient, which is some kind of approximation of the curvature in the principal directions corresponding to the elements in the diagonal of the Hessian, and tell us about the variation of the slope and the directions that we can go faster. Combining slope and curvature to get some sort of first order Newton method has already been done AdaDelta, however, as the information of the slope came from previous increments (already corrected gradients) and not strictly from previous gradients (the definition of momentum), we refrained from including it in this section.

Mainly the first method that embraced this approach is **Adam**, [38] (2014), not taking into account AdaDelta, (2012). Adam at the present time (2020) is one of the best result yielding first order methods, and it has become the almost de facto optimizer in deep-learning applications. It combines the versatility of pure classical momentum (to scape local minima) and component learning rate adaptation (to escape saddle-points), and it is quite fast.

The method design is as follows:

$$
\begin{aligned}
m_t &\longrightarrow \frac{1}{1-(\beta_1)^t}\left(\beta_1\ m_{t-1} + (1-\beta_1)\ \nabla_{(w,b)} L(w_t, b_t)\right), \\
v_t &\longrightarrow \frac{1}{1-(\beta_2)^t}\left(\beta_2\ v_{t-1} + (1-\beta_2)\ \nabla_{(w,b)} L(w_t, b_t) \odot \nabla_{(w,b)} L(w_t, b_t)\right), \\
(w_t, b_t) &\longrightarrow (w_{t-1}, b_{t-1}) - \eta\ \frac{m_t}{\sqrt{v_t} + \varepsilon},
\end{aligned}
\tag{2.36}
$$

where all operations (sums, products, roots and inverses) are element wise, m_t is called the first order momentum in t and β_1 is its decay rate, and v_t is called the second order momentum in t and β_2 is its decay rate.

Observe that inside the big parenthesis of m_t, we have a decaying average of the gradient, which resembles the classical momentum as defined in (2.28), and the big parenthesis of v_t is exactly the same as the decaying average principal direction curvature approximation in the AdaDelta (2.32). Each of the momentums is given an exponential rescale factor in the form of the term $1/(1-(\beta)^t)$, which tends to 1 as t increases. Hence, since $0 < \beta_2 < \beta_1 < 1$, in the beginning m_t dominates over v_t giving some sort of annealing effect by prioritizing momentum over curvature in the first steps. Finally the steps are updated as in AdaDelta following a Newton-like approach.

Other notable variations of Adam are: **AMSGrad** [39], which tries to fix the convergence problem of Adam in some instances (however, it is argued that the very specific instances that AMSGrad fixed do not really occur real problems, thus it is sometimes regarded as more complex and noisier version of Adam); **Nadam** [40] which uses Nesterov's momentum instead of classical momentum (from which the N in its name comes from); and **AdamW**, which tries to incorporate a Tikhonov regularization (which we will see in the Regularization section) inside the optimizer, instead of adding it to the loss function. As an additional comment, very recently a new type of more sophisticated first order methods which do not even require specifying a learning rate have appeared yielding apparently better results than Adam and its variations, one such method is **YellowFin** [41].

2.4.2 Second Order Methods

The previous first order methods yield good results in relatively small artificial neural networks (a few layers deep). They are not very computationally intensive and have linear convergence (which is often times enough), all at the expense of fixing a hyper-parameter, namely the learning rate. In particular, AdaDelta and Adam have proven to work really well against vanishing/exploding gradient and **sparse gradient** problems. Sparse gradients are a "kind" of vanishing gradients which happens when the dataset is sparse, i.e. there are rare features that occur in very few data points. Hence, if we recall that given the loss function form, the gradient is actually a sum of sub-gradients each associated to an individual data point, $1/N \sum_{i=1}^{N} \nabla_{(w,b)}(...)$, then the contributions to the gradient to fit these rare features are small in comparison to more common features, as there are fewer points and sub-gradients that can add to the sum. In that case it is said that there is a weak signal for that feature, and in practice this means that the parameters associated with that feature have smaller derivatives, creating saddle regions as the vanishing gradient problem does.

Nevertheless, as well as these first order methods work in many small problems with a somewhat homogeneous dataset, there are two related motives occurring in more complex problems that may require the consideration of higher order methods:

- The first, motive is computational cost: As we consider larger artificial neural networks, the number of parameters scales up, and the smaller number of steps required with the quadratic convergence (or almost quadratic) of second order methods start to become a computational advantage to the simpler but larger amount of steps required with linear convergence of first order methods.
- The second motive, very related to the first, is the high slope variation: As the number of parameters increase or the input dataset becomes more noisy, the loss function hyper-surface starts becoming more "bumpy", meaning that in using first order methods the strides in the direction of the gradient have to be shorter to account for its variation, i.e. the learning rate has to be reduced. Recall that AdaDelta and Adam corrected the learning rate based on some sort of approximation of diagonal of the Hessian. Therefore, when the Hessian increases (which happen when the number of parameters increase), the effect of elements outside of diagonal aggregate to become more relevant, and the methods lose part of their effectiveness.

Out of all the second order methods, the classic **optimization Newton method** is the principal one. This method relies on the Taylor expansion up to second order to approximate the function to be optimized by a quadratic function, in a local neighbourhood or region of confidence of a point (w_0, b_0). In our case, the loss function can be approximated by:

$$L((w_0, b_0) + p) \approx L(w_0, b_0) + \nabla_{(w,b)} L(w_0, b_0)^\top \; p + \frac{1}{2} \; p^\top \; H(w_0, b_0) \; p, \tag{2.37}$$

where p is an increment within the region of confidence and $H(w_0, b_0)$ is the Hessian matrix in (w_0, b_0). Then, as (2.37) is a quadratic function of p, it should have a unique minimum p_0, thus deriving the expression (2.37) with respect to $p \in \mathbb{R}^{n \times m}$, the minimum p_0 must satisfy:

$$\begin{aligned}
\frac{\partial}{\partial p}\Big(L((w_0, b_0) + p)\Big) &\approx \frac{\partial}{\partial p}\Big(L(w_0, b_0) + \nabla_{(w,b)} L(w_0, b_0)^\top \; p + \frac{1}{2} \; p^\top \; H(w_0, b_0) \; p\Big), \\
\frac{\partial}{\partial p}\Big(L((w_0, b_0) + p)\Big) &\approx \nabla_{(w,b)} L(w_0, b_0) + H(w_0, b_0) \; p, \\
0 = \frac{\partial}{\partial p}\Big(L((w_0, b_0) + p_0)\Big) &\approx \nabla_{(w,b)} L(w_0, b_0) + H(w_0, b_0) \; p_0, \\
p_0 &\approx -H^{-1}(w_0, b_0) \; \nabla_{(w,b)} L(w_0, b_0).
\end{aligned} \tag{2.38}$$

Therefore the optimization Newton method, being at (w_t, b_t) in step t, computes a new step point by approximating the original function in a confidence region around (w_t, b_t) by a quadratic function using Taylor's theorem, then finds the increment p_t that minimizes that quadratic approximation of the original function, and moves using that increment. In summary, the optimization Newton method update rule is:

$$\begin{aligned}
p_k &\to -H^{-1}(w_t, b_t) \; \nabla_{(w,b)} L(w_0, b_0), \\
(w_{t+1}, b_{t+1}) &\to (w_k, b_t) + p_t.
\end{aligned} \tag{2.39}$$

The increment p_t in known as *search direction* these types of methods.

Note that no hyper-parameters are required, and second order convergence is guaranteed by Taylor's theorem. As a major drawback, the method involves computing the Hessian matrix and inverting it, which is an extremely impractical and computationally expensive task, even if the number of parameters is just moderately large. Thus, there are a series of methods that modify this optimization Newton method to use approximations instead of the whole inverse of the Hessian, but preserve many of the good properties of the original. As a trade-off for their decrease in computational complexity, these methods loose their quadratic convergence, but they still get a much better than linear convergence, usually referred to as super-linear convergence, which outclass any first order method's convergence.

Quasi-Newton Method

In the Quasi-Newton family each step update uses the same idea as in the Newton method, with the a small variation. Instead of computing and using the Hessian matrix $H(w_t, b_t)$, we use an approximation matrix B_t which we have to update in every step. This means that, in essence, all the reasoning and derivation for the update rule are completely analogous to that of (2.37-2.38) with the only difference being writing B_t instead of $H(w_t, b_t)$. The final update rule will in fact have the same blueprint as the Newton's,

$$\begin{aligned} p_k &\to -B_t^{-1}\ \nabla_{(w,b)} L(w_0, b_0), \\ (w_{t+1}, b_{t+1}) &\to (w_k, b_t) + p_t, \end{aligned} \tag{2.40}$$

with a few additions (two in particular). The first is that, in every step $H(w_t, b_t)$ is being replaced by B_t, which is a matrix that changes with the curvature, but does not necessarily have to be an exact approximation of the Hessian matrix (for instance, it could be a scaled down version or be displaced). This means that we can trust the search direction p_t for its direction but not for its magnitude, thus, we will require a learning rate α_t to scale p_t at every step. There are two ways to compute α_t at each step, namely *inexact lines search* and *trust regions*. Coincidently, the Quasi-Newton methods that we will be seeing next use inexact lines search, and the truncated Newton methods of the next subsection use trust region. In particular, the inexact lines search that we will use is the satisfaction of *Wolfe conditions* which is given by the following set of inequalities:

$$\begin{aligned} L((w_t, b_t) + \alpha_k p_t) &\leq L(w_t, b_t) + c_1 \alpha_t \nabla L(w_t, b_t)^\top p_t, \\ \nabla L((w_t, b_t) + \alpha_k p_t)^\top p_k &\geq c_2 \nabla L((w_t, b_t))^\top p_t, \end{aligned} \tag{2.41}$$

with $0 < c_1 < c_2 < 1$. Using Wolfe conditions, α_k is progressively decreased until the inequalities (2.41) are satisfied. This guarantees the learning rate holds sufficient decease in curvature conditions.

The second issue is how to compute the matrices B_t at every step. In principle, two general requirements are demanded, that help calculate the matrix: it has to be symmetric like the Hessian and it must satisfy the *secant equation* (or *Quasi-Newton equation*). This secant equation can be obtained by differentiating in terms of the increment variable p for the quadratic approximation (2.37 with B_t) for a given step t:

$$\begin{aligned} \frac{\partial}{\partial p}\Big(L((w_t, b_t) + p) \Big) &\approx \frac{\partial}{\partial p}\Big(L(w_t, b_t) + \nabla_{(w,b)} L(w_t, b_t)^\top\ p + \frac{1}{2}\ p^\top\ B_t\ p \Big), \\ \frac{\partial L((w_t, b_t) + p)}{\partial ((w_t, b_t) + p)} \cdot \frac{\partial ((w_t, b_t) + p)}{\partial p} &\approx \nabla_{(w,b)} L(w_t, b_t) + B_t\ p, \end{aligned} \tag{2.42}$$

$$\nabla_{(w,b)} L((w_t, b_t) + p) \cdot (0 + Id) \approx \nabla_{(w,b)} L(w_t, b_t) + B_t \; p,$$

then, if we take the value of p for the actual increment $s_k = \alpha_k p_k$, we expect $B_t \approx B_{t+1}$, which yields:

$$\begin{aligned} \nabla_{(w,b)} L((w_t, b_t) + s_k) &\approx \nabla_{(w,b)} L(w_t, b_t) + B_{t+1} \; s_k, \\ \nabla_{(w,b)} L(w_{t+1}, b_{t+1}) - \nabla_{(w,b)} L(w_t, b_t) &\approx B_{t+1} \; s_k, \\ y_k &\approx B_{t+1} \; s_k, \end{aligned} \tag{2.43}$$

where $y_k = \nabla_{(w,b)} L(w_{t+1}, b_{t+1}) - \nabla_{(w,b)} L(w_t, b_t)$. The last expression of (2.43) is what is known as the secant equation.

Therefore the general update rule of a Quasi-Newton method would be:

1. Obtain the search direction solving $p_k = -B_t^{-1} \; \nabla_{(w,b)} L(w_0, b_0)$.
2. Use Wolfe conditions (or any other approach) to find α_k.
3. Compute the next gradient $\nabla_{(w,b)} L((w_t, b_t) + \alpha_k p_k)$.
4. Compute the next matrix B_{t+1} imposing symmetry and satisfying the secant equation.

Every Quasi-Newton method varies in their approaches to compute α_t, and speciality B_t, as are many ways to construct symmetric matrices that satisfy the secant equation. Ideally, we want to use a expression of B_t which is easy to compute and easy to invert. The first of these methods was the **Davidon–Fletcher–Powell** method (DFP), but in current times the **Broyden–Fletcher–Goldfarb–Shanno** method (BFGS) and its Limited Memory version (which require less storage memory) are the most widely used methods in this family.

The BFGS impose the extra condition that B_t has to be positive defined, on top of the symmetry and satisfaction of the secant equation, and use Wolfe conditions. The following expression is the one that satisfies the three conditions:

$$B_{t+1} = B_t - \frac{B_t s_t s_t^\top B_t}{s_t^\top B_t s_t} + \frac{y_t y_t^\top}{y_t^\top s_t}, \tag{2.44}$$

which allows for the Sherman–Morrison formula to invert the matrix recurrently:

$$B_{t+1}^{-1} = \left(Id - \frac{s_t y_t^\top}{y_t^\top s_t} \right) B_t^{-1} \left(Id - \frac{y_t s_t^\top}{y_t^\top s_t} \right) + \frac{s_t s_t^\top}{y_t^\top s_t}. \tag{2.45}$$

Section based on [42, 43]. [44] has a really recommendable coverage on, line search conditions, trust regions and Quasi-Newton methods, far beyond this work. Also, [45] proposes a mini-batch adaptation of the L-BFGS method.

Truncated Newton Method

While Quasi-Newton methods such as BFGS have been adapted and used for large deep-learning applications successfully, not many other types second order methods have historically been used. In 2010 though, Martens [46] proposed an adaptation of a type of methods called the truncated Newton methods or Hessian-Free methods, which has been fairly replicated. The set-up for this method also revolves around trying to fixing the main problem of the optimization Newton method. Hence it tries to find the search direction p_t by solving the system in (2.38) without having to invert the Hessian matrix.

Rearranging the last expression of (2.38), the system to find the search direction in step t, can be written as:

$$H(w_t, b_t)\ p_t \approx \nabla_{(w,b)} L(w_t, b_t). \tag{2.46}$$

Here, the point of this method is that we do not have to compute the Hessian or any approximation matrix at all. Given any vector v, its product by the Hessian matrix can be thought of a directional derivative of the gradient as a function. Thus, we can compute the product $H(w_t, b_t)\ v$ by evaluating an extra derivative:

$$H(w_t, b_t)\ v = \lim_{\varepsilon \to 0} \frac{\nabla_{(w,b)} L((w_t, b_t) + \varepsilon v) - \nabla_{(w,b)} L(w_t, b_t)}{\varepsilon}, \tag{2.47}$$

where the limit can actually be replaced by some finite difference taking a small ε or by any other method. Then, we can use the Conjugate Gradient (CG) algorithm to solve the linear system (2.46) combined with (2.47) to obtain the search direction p_t. In summary the steps for step t would be:

1. Randomly initialize a search direction $p_{t,0}$
2. Apply the CG algorithm to solve the linear system of (2.46). Since with regards to the Hessian (which is the matrix of the linear system), the CG only requires the products $p_{t,i}^\top\ H(w_t, b_t)\ p_{t,i}$, use the finite differences of the gradient from (2.47) to compute this product without needing the Hessian.
3. Once the CG has converged (or is close enough) to the solution p_t, update the point $(w_{t+1}, b_{t+1}) = (w_t, b_t) + p_t$ and repeat the process.

This last method is the only one in this section that we have not used, as the implementation required much more technical work than the rest, since there is no package from which to adapt it. However, we still wanted it to be explained as a promising second order alternative to the LBFGS and BFGS.

2.5 Activation Functions and Parameter Initialization

In this section we will be introducing the most commonly used activation functions in deep-learning. We have avoided this issue up to this point because the choice of activation functions is crucial in mitigating the gradient problems and speeding up the optimization process, which we have now covered. Also, in practice, all the activation functions of an artificial neural network are chosen to be the same (except for the output layer), so from now on we will assume this is the case.

Although any function can do theoretically as long as it is continuous and one time piecewise differentiable, there are some other that can proof beneficial. For example, using bounded or dissipative functions can add a component of localized training, making the output of the activation function in certain neurons mostly dominant with respect to others for some range of input values, thus the parameters of those neurons become descriptive of the characteristics of that range of input values. As well as this, functions with derivatives close in magnitude to their primitives, or derivatives that have a close to linear behaviour in some region are desirable, as this helps mitigate the gradient problems and improve convergence through the initialization of the parameters (we will look this last part in the next subsection).

The following table shows the main classic activation functions used in deep-learning (holding many of the properties stated above), in order, from most used (top) to the least used (bottom).

Name	**Value**	**First Derivative**
Sigmoid:	$a(x) = \sigma(x) = \dfrac{1}{1+e^{-x}}$	$a'(x) = \sigma(x)(1-\sigma(x))$
ReLU	$a(x) = \max(x, 0)$	$a'(x) = \Theta(x)$
Hyperbolic Tangent:	$a(x) = \dfrac{\sinh(x)}{\cosh(x)} = \dfrac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$	$a'(x) = \dfrac{1}{\cosh^2(x)} = \dfrac{4}{(e^{x}+e^{-x})^2}$
Exponential:	$a(x) = e^{x}$	$a'(x) = e^{x}$

Table 2.1: List of main activation functions.

About the nomenclature in table above: the sigmoid function is almost always represented as $\sigma(x)$, and $\Theta(x)$ stands for the Heaviside step function, which is 0 if the input is negative and 1 if the input is positive. Also due to the specifics of this work, where the loss function contains derivatives of artificial neural network, we would like to impose the following reasonable condition. If n is the degree of the differential system linear operator (this is, the highest order derivative of the operator), then at least the first $n+1$ derivatives of the activation function should be different from zero. This condition makes sense as we would like to have some contribution (or signal) to model the higher order derivatives and not to have have them vanish. Therefore, from the previous list the ReLU (Rectified Lineal Unit) function is mostly unsuitable for this work, since its second order derivative is strictly zero. We can see this effect clearly from (2.16), where we expressed the Hessian tensor (second order derivatives tensor) of our example network. If we are to consider the tensor arrangement of activation functions with ReLUs, then $D^{n_\ell,n_\ell}(a^{[\ell]}{}_{n_\ell}(z^{[\ell]}{}_{n_\ell})) = 0^{n_\ell,n_\ell}$, is the tensor of zeros, and since these terms appear multiplying in every factor of (2.16), all the second order derivatives of the artificial network with respect to the inputs are zero.

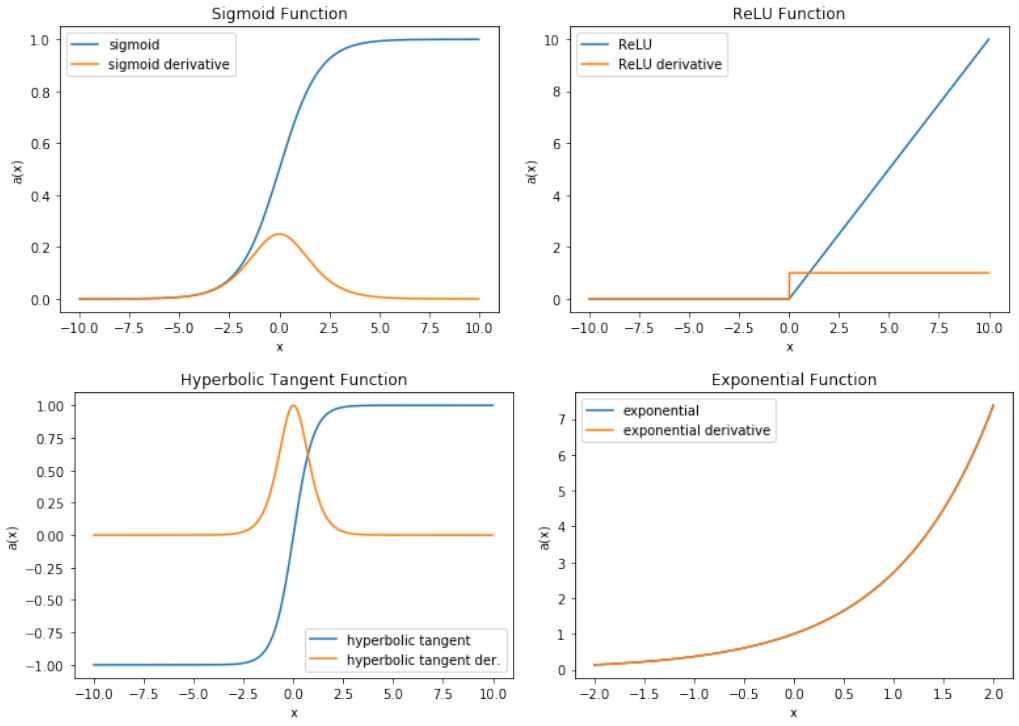

Figure 2.7: Main activation functions and their first order derivatives.

Observe that all the classic activation functions are monotonically increasing and bounded: the exponential and ReLU in the positive values, the sigmoid between 0 and 1 and the hyperbolic tangent between -1 and 1. In general, using activation functions with these two characteristics allows for some sort of localized pulse-like behaviour, which can be helpful in modelling traits over very specific regions. For instance, lets assume we have two neurons that use sigmoids, both share the same one input x, and having weights and biases $w_1 = -3$, $w_2 = 2$, $b_1 = -2$, $b_2 = 10$. Figure 2.8, shows the result of imputing x to the neurons and adding them up in the next layer. Note that the end result is some kind of continuous version of a square pulse function, which in this case differentiates between the region (-2.5,7.5) and the rest of the real line.

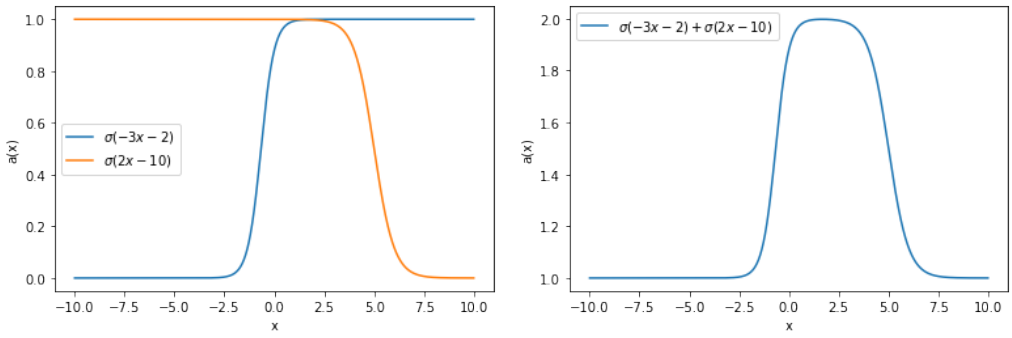

Figure 2.8: Combination of sigmoid functions.

Through time many secondary variations of these functions have appeared trying to fix some of their inconveniences. Two popular ones that we will consider are the Swish function [47], which also appear under the name of Sigmoid-Weighted Linear Unit [48], and the Softplus function [49]. The first offers a smooth continuous variation of the ReLU function using the sigmoid in its formula (thus has no vanishing higher order derivatives), and the second fixes the acute growth of the exponential function.

Name	**Value**	**First Derivative**
Swish:	$a(x) = x \cdot \sigma(x)$	$a'(x) = \sigma(x) + x\ \sigma(x)\ (1 - \sigma(x))$
Softplus	$a(x) = log\ (e^x + 1)$	$a'(x) = \dfrac{e^x}{e^x + 1}$

Table 2.2: List of secondary activation functions.

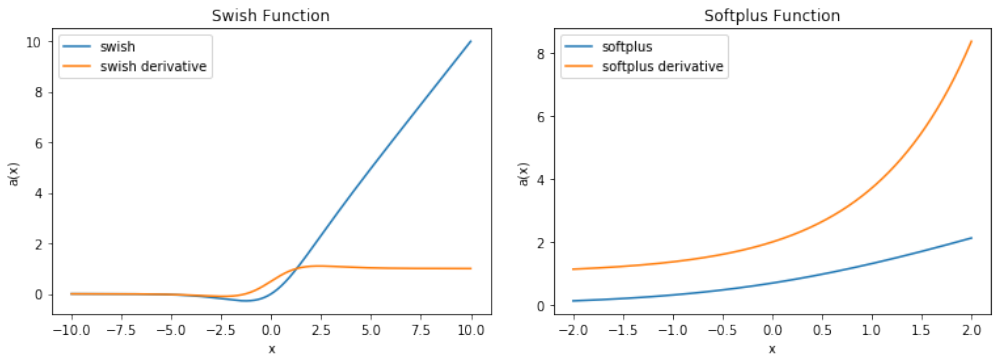

Figure 2.9: Secondary activation functions and their first order derivatives.

2.5.1 Parameter Initialization

An issue which depends directly on the activation function choice is the parameter initialization, necessary as the starting point to perform the iterative optimization methods we saw in the previous section. Actually, the optimization process is very sensitive to the initial point, and slight variations can even lead to very different results.

Each activation function has its own set of different initialization arrangements (many named after the authors that proposed them). However, all of them share the same the basic principle, which is to set the parameters so that, at least for the first inputs, the activation functions behave in their linear regime. For example, if we were to choose exponentials as activation functions, $a(x) = e^x \approx 1 + x + x^2/2 + ...$, we would like the parameters to be close to 0, so the inputs to each function are also close to 0 (or small) for each activation function behave like $1 + x$ in the first steps. The reason behind using this idea (with a few refinements that we will be covering in the next paragraph) is because, in practice, solving exactly the non-convex optimization problem is virtually impossible, and many times even finding a decent enough result becomes a matter of luck with trial and error (using different methods and initialization points). By starting in a point at a region where activation functions behave linearly, we guarantee that the slopes (the loss derivatives with respect to the parameters) will not vanish or explode abruptly from the start and convergence will be smooth (at least in the beginning).

There are two extra considerations that we have to add to the previous idea of starting in the linear regime.

- First we have to break the symmetry of the parameters. This is means that we cannot initialize all the parameters in a layer with the same values. Recall the derivatives computed in section 2.3.1. Giving the same values to the parameters in a layer implies that the derivatives with respect to the parameters for that layer are all the same, thus at every step of the optimization the parameters also increment by the same amount, and stay equal along the whole process. To avoid this symmetry which effectively impede the ability of the parameters of a layer to adapt to different traits of the dataset, we add some variance by drawing the parameters from a distribution, usually a uniform or a normal. For instance, in the exponential case, instead of initializing the parameters with 0 (which is actually the only value for which the parameters cannot be trained), we would draw them from something like $\mathcal{N}(0, 0.1)$ or $\mathcal{U}(-0.1, 0.1)$.

- Second, we have to balance the variance of the weights in every neuron, so that the neurons outputs all stay within a certain magnitude. The two conditions to impose are for the linear combination of the neurons during evaluation to have similar variance (2.48), and for each loss derivative with respect to the parameters of a neuron in during back-propagation to also be similar in variance (2.49).

$$\forall \ell, \ell', n, n', \quad Var(z^{[\ell]}{}_n) = Var(z^{[\ell']}{}_n), \tag{2.48}$$

$$\forall \ell, \ell', n, n', \quad Var\left(\frac{\partial L}{\partial w^{[\ell]}{}_n}\right) = Var\left(\frac{\partial L}{\partial w^{[\ell']}{}_n}\right). \tag{2.49}$$

  Assuming independence with respect to inputs and zero mean (2.48-2.49) become:

$$\forall \ell, \ell', n, n', \quad n_\ell \; Var(w^{[\ell]}{}_n) = n_{\ell'} \; Var(w^{[\ell']}{}_n), \tag{2.50}$$

$$\forall \ell, \ell', n, n', \quad n_{\ell+1} \; Var(w^{[\ell]}{}_n) = n_{\ell'+1} \; Var(w^{[\ell']}{}_n). \tag{2.51}$$

Averaging the previous conditions yield:

$$\forall \ell, n, \qquad Var(w^{[\ell]}{}_n) = \frac{2}{n_\ell + n_{\ell+1}}, \tag{2.52}$$

where n_ℓ is the number of inputs of the neuron (called the fan-in), and $n_{\ell+1}$ is the number of outputs to the neuron (called the fan-out), which in fully-connected artificial neural networks is equivalent to the number of neurons in the current and the next layer respectively. Note that in practice we are simply scaling by the size of the network.

Getting these three ideas together (linear regime, variation and fan-in/out scaling) lead to the Xavier/Glorot [50] and He [51] initializations, which are the main ones we will be using throughout this work. The former uses straight forward (2.52), and the latter makes a correction for linear units (which are activation functions which drop or almost drop their negative values, for example the ReLU) by dropping the fan-out term. These initializations in their uniform and normal distribution versions therefore are:

Xavier Uniform:
$$\mathcal{U}\left(-\sqrt{\frac{6}{n_\ell + n_{\ell+1}}}, \sqrt{\frac{6}{n_\ell + n_{\ell+1}}}\right), \tag{2.53}$$

Xavier Normal:
$$\mathcal{N}\left(0, \sqrt{\frac{2}{n_\ell + n_{\ell+1}}}\right), \tag{2.54}$$

He Uniform:
$$\mathcal{U}\left(-\sqrt{\frac{6}{n_\ell}}, \sqrt{\frac{6}{n_\ell}}\right), \tag{2.55}$$

He Normal:
$$\mathcal{N}\left(0, \sqrt{\frac{2}{n_\ell}}\right), \tag{2.56}$$

For the sigmoid and hyperbolic tangent activation functions we will use the Xavier/Glorot initialization, and for the ReLU, exponential, swish and softplus activation functions we will be using the He initialization.

2.6 Regularization

At this point we have all the tools necessary to build an artificial neural network and to apply an optimization method to train it. However, there is still the issue of selecting and tuning the hyper-parameters of our inverse problem (recall that these are problems in which we have sample data and we want to fit the parameters of certain model to fit the data), which is hard. The issue here is that, given any model with certain hyper-parameters defining it, in our case the number of layers and neurons, there are two opposite situations that may arise. Either the number of hyper-parameters are too few and the model is too simple to fit the data (underfitting), or the number of hyper-parameters are too many and the model has extra degrees of freedom that do not necessarily fit the data (overfitting).

Underfitting is a problem which is relatively simple to spot, it is impossible to accurately fit the model to the data for as much as we train the model, and it has its only solution in increasing the number of hyper-parameters or the complexity of the model. For overfitting, detecting and solving to the problem is not as easy. To mitigate overfitting, as well as sometimes fixing the ill-posedness of a problem, regularization techniques are commonly used.

An overfitted model can be harder to train, this is because in the optimization process, the gradient that updates the parameters contains contributions of the extra degrees of freedom of the model. This can make the optimization process much noisier and susceptible to over-represent outliers. Also, overfitted models tend to generalize data poorly, i.e. their ability to predict on new data decreases, as any data not strictly in the training set would not simply be the extrapolation of the relations among the training points, but would also contain the contributions of the degrees of freedom. Regularization techniques help reduce the number degrees of freedom of the models.

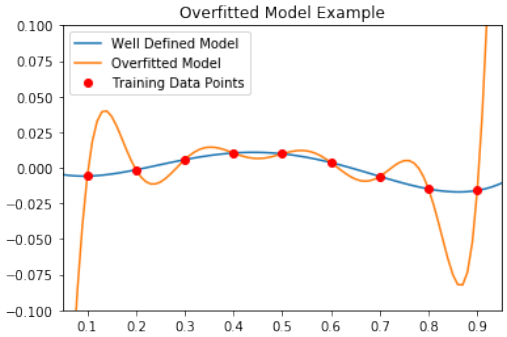

Figure 2.10: Example of overfitting of a model.

The previous Figure 2.10 show an example of an overfitted model. Both the blue line and orange line are polynomial models that fit the training data (red points). The orange model (which has more coefficients than the blue model) is overfitted though, since, most prominently at the extreme points, it generates a bumpy behaviour completely unrelated to that of the training data, related to the unnecessary extra parameters that we have added.

In practice finding the exact right number of hyper-parameters required in a model is an almost impossible task by the sheer amount of possibilities. Besides a trial and error strategy would impractical to compare models due to training being a computationally costly process sensitive to the optimizer and initial conditions. Hence effectively, since there is no general rule that can be followed, the hyper-parameters are often chosen within a reasonable rough margin (mostly based on the results of the first few steps of the optimization), guaranteeing some overfitting. Then regularization techniques are used to clamp down on the extra degrees of freedom of the model. This is much more viable approach than training an almost exponentially increasing amount of models with different numbers of hyper-parameters to narrow down the right number which does not underfit or overfit the data.

Each of the following subsections will be dedicated to a different regularization technique. We will improvise two categories to group these techniques based on the main general principles behind them, namely noise-based regularizations and restriction-based regularizations. In this work we will only be using restriction-based regularizations though.

2.6.1 Noise-based Regularizations

Behind the noise-based regularizations lies the idea that adding an stochastic component throughout the training process to add some (small) variance into model. To much variance can lead to a chaotic model (undesirable), but adding a small variance during training can be very beneficial as it can somewhat be seen as employing the extra degrees of freedom to account for the extra variability of the model.

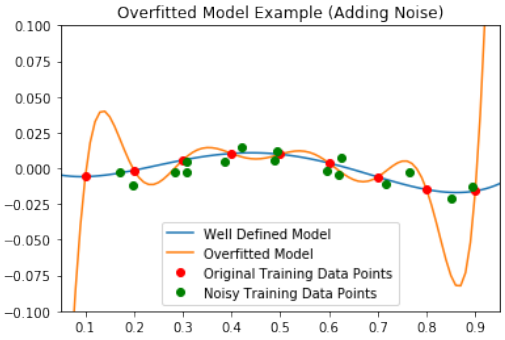

Figure 2.11: Example of a model adding noisy input.

Intuitive the concept in its most trivial form can be seen in Figure 2.11, which is nothing more than Figure 2.10 to which we have added some extra noisy points (the original points plus some extra noise). It now becomes apparent that when fitting the model the end result would be closer to the well-defined model (blue line) than to the overfitted model (orange line), since now the model has to also account for the green dots for which the blue line has a smaller error, specially at the extremes. Thus, if the variance is small the extra degrees of freedom are spent to ensure that small variations in the data do not yield overwhelmingly large changes in the model.

Now that we have explained how noise works, we will be looking at how to introduce it into the model for the training phase. The most obvious way is to introduce variance into the model is by using the **direct approach**, this is add noise to the input data, like for example, $x_i + \mathcal{N}(\mu, \sigma)$. A much smarter way to apply this concept is making use of some invariant to generate new data, in what is called **data augmentation**. This happens a lot in object recognition, whereby a car in a picture is car independently of the image being rotated 90^{o} or the car appearing in the center or a corner of the picture, thus we can rotate or shift the pictures to generate new inputs.

There are much more sophisticated way to introduce variance into the model. One of these is using **noisy neurons**, which implies that, only during the training phase, we add noise to the output of each neuron, this is ${y^{[\ell]}}_{n_\ell} + \mathcal{N}(\mu, \sigma)$. Another one is the **dropout** technique [52], which only during the training phase uses a probability to suppress the output of a given neuron. Hence, for example, for every neuron at each step we would draw a number from a uniform distribution $p \sim \mathcal{U}(0, 1)$, and if $0.9 < p < 1$ we would set its output to 0 (this would be a dropout of 10%). At last we recall that the annealing effect explained in section 2.4.1 for the vanilla stochastic gradient descent can also be considered some sort of noise-based regularization technique.

One of the main issues with noise is that, we want to introduce some small variance throughout the model during training, but we want this variance to be controlled and small along the process, for the end result not to be a chaotic model fitting only noise. By this we mean that, we do not want the effect of the variance introduced in the early layers (the ones closest to the input) to explode in the following layers. We want the variance contribution to remain small as it gets imputed into its next layers. The principal idea to assure this, as well as providing other very good properties, are the **normalization** techniques which has become a staple in many deep-learning, namely batch and layer [53] normalization. It consists in normalizing either input batch or the outputs of the neurons of every layer, respectively.

In general for small models (specially in their number of layers) with well initialized parameters do not require normalization because in those dimensions the noise will most likely not scale up. One inconvenience of normalizations is that it correlates the gradients. Recall from (2.4-2.5), that the loss function is a summation over each of the individual losses at the points of the collocation, which makes the derivatives of the parameters with respect to the loss a sum of uncorrelated derivatives. Normalization entangles these derivatives which makes the update gradient for the optimizer correlated with respect to collocation points, thus, the gradient computation at every step of the training becomes less sparse and more computationally expensive. Additionally, normalization does not work well with dropout.

The reason we have disregarded noise-based in favour of restriction-based regularizations, although both are mutually compatible, is because understanding the behaviour of noise in a context where we are considering the derivatives of the artificial neural network is very risky and becomes exponentially more complex with the order of the derivatives. Also, in the case of normalization (which we have tested for this work), where we do not actually introduce variance but limit its effects, the extra increase in computational cost, caused by the correlation of the gradient, builds up on the already unavoidable computation of higher order order derivatives of the neural network required in this work, making the training process many times slower and impractical. On the contrary, the techniques that we have categorized as restriction-based are mostly (soft or hard) binds on the parameters. Thus, their application do not interfere with the computation of L_1 and L_2, and so, their effect is applies afterwards.

2.6.2 Restriction-based Regularizations

On the other side of the spectrum lie what we have named as restriction-based regularization techniques. These are a set of soft or hard constrains on the parameters, added in the loss function or are applied independently. Hence, any extra degrees of freedom in the model may be invested in fulfilling these constrains.

The most common of these all are the **weight penalties** (actually it would be parameter penalties). This regularization techniques rely on an extra term, which is added to the loss function, and impose some preference in the parameters (this would correspond to the placeholder term R introduced in (2.1)). The most notable of these regularizations is the popular Tikhonov regularization which implies adding the following term:

$$R = \lambda \left(||w||_2^2 + ||b||_2^2 \right) := \lambda \sum_{\ell, n_\ell, m_{\ell-1}} \left(||w^{[\ell]}{}_{n_\ell}^{\ m_{\ell-1}}||_2^2 + ||b^{[\ell]}{}_{n_\ell}||_2^2 \right), \tag{2.57}$$

where λ is the regularization scale factor. Technically, this is same Tikhonov regularization as the one used in the least squares method for linear regression. One possible interpretation for it is that we impose a "minimum energy" model, i.e. we are looking for a model where the bias and specially the slopes are the smallest possible (yielding a "flatter model"), which happens naturally as we minimize the term R in the loss function. Another common reading is that the additive contribution of $\nabla_{(w,b)} R$ to the total update gradient introduces some sort of dampening force which penalizes the optimization method when moving in directions where the loss function, $L(w, b)$, is less smooth (which happen with larger values of w and b). Going back to Figure 2.10, the well-defined model would satisfy best the term (2.57).

Furthermore, the reason why these are called weight penalties and not parameter penalties, is because in most cases this regularization only affects the slopes (weights), and $||b||_2^2$, the bias terms in (2.57), are dropped. For this work though, we are considering the bias term as we believe it provides some sort of centring effect on the neuron outputs which (in the particular problem instances chosen for this work) help in speeding up the training. However, in a general context, this could prove tricky and is generally undesirable, specially in instances where the solution to the initial/boundary problem we want to approximate have many different localized features, i.e. the solution is very bumpy (which will not be case in this work). The reason for this, is that biases, although not strictly necessary in an artificial neural network (an only weights network is absolutely functional), when applied help to optimize the differentiation among neurons in the same layer. For example, in a neuron using sigmoid activation functions, given two inputs from two different regions and combining them with the weights, suppose we obtain values 0.5 and 1.5. Then, computing the activation, giving the neuron $b = 0$, yields a difference in output between the two inputs of ~ 0.19, and for $b = 2$, it yields a difference in output between the two inputs of ~ 0.05. This is apparent from Figure 2.7 as we see that the maximal slope is centred around zero. Hence, using a bias to shift the product of inputs and weights in a neuron can lead to an increase or decrease of the difference in outputs among values in different regions, an effect that applying a Tikhonov term which pushes $b \to 0$ can even negate, as contrary to the multiplicative contribution of the weights in the neuron, biases have an additive one requiring much larger values to have a significant effect. A second interpretation can be drawn by explanation given in Figure 2.8, whereby we argued that two increasing functions could be combined to form a sort of dissipative square pulse function. By reducing the biases, we reduce the amplitude of the plateaus (width of the windows) of these arrangements, which reduces some the specificity that can be achieved for certain regions in the model. Finally, we can still argue that the loss of some localized specialization in the neurons due to the term $||b||_2^2$ should still not pose a problem in the adaptability of the model, as it would simply make the contributions of a the neurons in a layer for a region more overlapping, why should this be a concern and undesirable? Although this last statement is true, a generally desired feature for a good artificial neural network is for it to be a *sparse artificial neural network*, i.e. that for any input given to the network almost all of the signal is contributed by just few neurons (or in other words any input only require passing through few relevant neurons neurons, and not all of them have to be active at the same time). Because of the objective of this work, which is proving it is possible to approximate solutions of initial/boundary problems by artificial neural networks, we would rather have the extra regularization effects of the term to obtaining a sparser network.

Another custom weight penalty that we have devised and seems to work rather well in this work is the following:

$$\begin{aligned} R &= \lambda \left(\left\| \frac{\partial \hat{u}(x)}{\partial w} - \frac{\partial}{\partial w}\frac{\partial \hat{u}(x)}{\partial x} \right\|_2^2 + \left\| \frac{\partial \hat{u}(x)}{\partial b} - \frac{\partial}{\partial b}\frac{\partial \hat{u}(x)}{\partial x} \right\|_2^2 \right) \\ &:= \lambda \sum_{\ell, n_\ell, m_{\ell-1}} \left(\left\| \frac{\partial \hat{u}(x)}{\partial w^{[\ell]}{}_{n_\ell}^{\ m_{\ell-1}}} - \frac{\partial}{\partial w^{[\ell]}{}_{n_\ell}^{\ m_{\ell-1}}}\frac{\partial \hat{u}(x)}{\partial x} \right\|_2^2 + \left\| \frac{\partial \hat{u}(x)}{\partial b^{[\ell]}{}_{n_\ell}} - \frac{\partial}{\partial b^{[\ell]}{}_{n_\ell}}\frac{\partial \hat{u}(x)}{\partial x} \right\|_2^2 \right), \end{aligned} \tag{2.58}$$

with its idea being, instead of using the extra degrees of freedom to obtain the solution with minimal slopes, to find a solution whose derivatives with respect to the parameters and with respect to parameters and inputs are similar in magnitude. This turns out to work quite well as we will see in the next section, and in fact, it requires no extra computations as all the derivatives involved in (2.58) are automatically calculated when computing $\nabla_{(w,b)} L_1$ (the loss of the differential operator).

A topic that we have not covered yet is the setting of the regularization coefficient λ. This has to be done manually and it has to be revised and updated at every step of the training process. We want the model to minimize its errors $L_1 + L_2$ (main objective) over fitting the term R (secondary objective). Typically, to stablish priority as with any other multi-objective function we use the coefficient λ, to limit the magnitude in which the term regularization R contributes the total loss without becoming irrelevant. One possible reasonable demand would be to ask for the regularization term magnitude to be between 10% and 20% of the main term magnitude, of in other words $0.1 \cdot (|L_1| + |L_2|) \lessapprox |R| \lessapprox 0.2 \cdot (|L_1| + |L_2|)$, and adjust λ every time the criterion is not met. A much logical approach at first sight would seem to be, adapt λ as a function of the magnitudes of the terms, for example in the Tikhonov case, we could always make $\lambda = 0.1(|L_1| + |L_2|)/|\left(||w||_2^2 + ||b||_2^2\right)|$, for R to always be 10% of the other two terms. However, bear in mind that every time we modify λ we are in fact changing the total loss function $L(w, b)$, which is detrimental for the robustness and convergence of the optimization process. Thus, it is best to use a threshold that allows the optimizer to train on a fixed hyper-surface for many steps until a correction has to be done, than to have the optimizer move on a hyper-surface that changes in every step, more so considering that all but vanilla SGD use some kind of memory from previous gradients, which becomes irrelevant if the hyper-surface has changed. Nevertheless, in section 3.3 in the next chapter, we will propose an alternative approach to deal with this issue as part of a larger framework to deal with multi-objective loss functions, which seems to perform much better than this classical threshold strategy and achieve faster convergence.

An alternative to weight penalties are the **weight constrains** (or parameter constrains), which are sets of inequalities that can be applied to the parameters, either by component value like $a < |w^{[\ell]}{}_{n_\ell}^{m_{\ell-1}}| < b$, or by node or layer norm $a < ||w^{[\ell]}{}_{n_\ell}||_2^2 < b$. The way to apply these inequalities is usually by *clipping*, this is for example, if we update a parameter in a given training step and surpass the upper bound b, then parameter is set to b. This effectively stalls the training of parameters that have become to large (usually very dominant effects) or prevents from vanishing parameters that have become too small (usually very negligible effects), forcing a more even distribution in the relevance of the parameters. A more common clipping practice is **gradient clipping**, which is applied on the gradients with respect to the parameters used to update the parameters on an upper bound. Hence, this limits the effect of any exploding derivatives case, as any extremely large derivative which would mean an extremely large update, whereby using gradient clipping would be instantly reduced to a maximum reasonable range. In the training of all the models in this work we have implemented a component upper bound weights and bias clipping 10^3, which is reasonable enough for the small scale of the models, and an upper bound gradient clipping by norm of all the parameters in the layer of 1.

Finally, we propose and will implement the following idea for a regularization, which can be drawn from the context of this work. Since want to train artificial neural networks to satisfy differential equations, and thus approximate their unique exact solutions, any conservation law satisfied by the exact solutions must also be approximately satisfied by the artificial neural network. Hence we can add **conservation laws** to the loss function the same way we did with the other weight penalty regularization, by simply replacing the conservation laws into the placeholder R (and optionally adding some regularization constant). Actually we could argue that, since the network must also satisfy the conservation law as closely as possible as the exact solution does, the term is not actually a regularization but a legitimate extra term which speeds up training, and not an extra condition which help select some specific model among the many that approximate the solution (essentially a regularization).

Not many differential equation have known conservation laws though, and thus, are hard to come by. Besides, in cases where conservation laws are known, usually adding an external force to the equation (something we will be doing in this work) invalidates such laws. Sometimes, although it is rare, this can be accounted for by deriving again the conservation law with the external force, and this can lead to the original law with some extra terms, like an the integral over the domain of the external force if the domain is bounded.

2.6.3 Other Regularizations

In this subsection we will look at two very common practices that might as well fall in the category of regularization. The first is known as **pre-training**, which consists of, instead of initializing a new artificial neural network to approximate an initial/boundary problem, we would use an already existing one as a starting point, with the hopes that this network is already closer to the desired outcome. All the attempts in this work to use pre-training with artificial neural network trained to only fit the initial/border data, to only fit the domain, or to fit the differential equation dropping any of its terms, have either had the same performance as using no pre-training, or worsened. The most plausible explanation might be that being the loss function multi-objective, it is best to keep a balanced agreement between the two parts from the start, rather than starting by fitting either L_1 or L_2, as the region in the parameter space we can fall in during these one term optimizations might be useless or even detrimental to the other term, thus making worse the combined optimization.

Second is **early stop**, which is not only always applicable, but useful in many ways. Data used to fit artificial neural networks (or any model by that), is usually split into two groups, the *training data* and the *validation data*. The training data is used to fit the model (it is the data imputed in the loss function during the optimization), and validation data is used as a control mechanism to prevent overfitting of the model. Therefore every certain number of iterations in the process (for example 1000 iteration), we evaluate the loss with respect to the validation set, and if this validation set loss has worsened with respect to its previous evaluation, then we stop the training (this is early stop). The principle here is that the training process is blind to the validation data (not used), however the model should still fit this data as part of its capabilities to generalize beyond the training points. Similarly to the workings of noise, if the validation loss gets worst, the behaviour beyond the training points becomes undesirable, and thus, the model is overfitting. When this happens we can simply stop the training completely, or it might be a sign that the learning rate of the optimizer is too large and we have to reduce it, we can try introduce some extra regularization or adapt the regularization coefficient λ to correct the model, or we can **generate a new batch** which is also a regularization technique, and resume the training. Hence, early stopping is very useful not only as a regularization technique, but it gives a cue to rectify the training of the model when the process is stalled. Throughout this work we use validation intervals (we check the validation loss) every 1000 steps of training.

As a final note, we want to address the reason why we do not check the validation loss at every step. The first motive is because first order optimizers are not always smooth, i.e. the loss function can be decreasing but in an oscillating (conjugate) manner (specially around valleys), thus in a very short span early stop could confuse one of these fluctuations where the is at local maximum with a stop criterion. Still, for second order methods which use line-search that guarantees that there is always a decrease in loss (or they simply stop), the reason is that it is computationally more expensive to evaluate an extra loss at each step, and a few more steps from the early stop criterion will not substantially change the model.

Chapter 3
Case Studies and Simulations

In this chapter we will finally be training artificial neural networks to approximate the solutions of some instances of initial/boundary, starting by the most simple case and building up to more complex operators.

The layout of this chapter will be fairly consistent. Besides the first three sections, dedicated to the general implementation related topics practical to this work: the coding framework, function approximation capabilities of artificial neural networks, and adaptation to multi-objective function training; each of the remaining sections follows the same structure of posing a problem instance, training, and result analysis, with different operators. All of the operators used in this chapter have already been detailed in Table 1.2, and as mentioned in the introduction, we will be using only Cauchy initial/boundary conditions. With regards to the external forces, we will be selecting them ad-hoc in every problem so that the solution is a simple known polynomial. This way we can benchmark the artificial neural network results against the exact solution with ease.

3.1 Coding Artificial Neural Networks

From an implementation standpoint, deep-learning model training require the computation of many operations, specially linear combinations (tensor operations). Recall that an artificial neural network neuron is composed of a linear combination of the outputs of the previous layer, and an application of a non-linear activation functions. In terms of activation functions, little can be done to improve performance, but the many sums and products of the linear combinations are susceptible to high parallelization, as they are mostly independent among neurons, low in computational cost and high in number. Therefore, in order to speed up learning, instead of using CPUs, which at the time of this work have up to 8/16 cores, i.e. processing units and maximum amount of operations that can be perform in parallel, we can make use of the already existing GPU hardware. GPUs are optimized for image processing, a process which rely heavily in matrix multiplication. Thus contrary to CPUs composed of a few powerful cores, GPU are built using a large number of lower end cores, which at the time of this work can be in average of 120 cores. By parallelizing the linear combination operations to the many cores of a GPU, we can reduce the training time of an artificial neural network manifold, especially in larger networks. Nowadays, a new piece of hardware specially designed for deep-learning training has irrupted called TPUs (Tensor Processing Units). This hardware contains an even larger number of cores and its architecture is ad-hoc designed to parallelize tensor operations, improving on the capabilities of GPUs.

In order to manage and distribute the flow of operations to make the most use of GPUs and TPUs, there are several developed software solutions. We will briefly give a basic understanding on the most prominent high/medium/low level options.

On the lowest level, almost exclusively, lies the APIs named CUDA, developed by GPU maker NVidia. This API allows for direct control and distribution of operations of the cores in a GPU/TPU. However, from a practical perspective, unless we want to really customize and micromanage the resources in our GPU/TPUs, this level of control is too much. Thus, there are several middle-level libraries used in deep-learning that automatically handle these tasks, the most popular ones being TensorFlow, developed by Google, and PyTorch, which is open source (both running on CUDA). The way this libraries work is by implementing their own class for multi-dimensional objects (like arrays), and every time tensor operations are performed on these objects, they use CUDA under the hood to distribute its operations into the GPUs and/or TPUs cores automatically. This simplifies the work by allows us to concentrate on programming the mathematical framework for the models without having to deal with the management of the parallelization tasks. Lastly, on top of these libraries, there are also higher level ones, such as Keras library which hinges on TensorFlow. These build on the multi-dimensional class to further implement classes for layers, models, optimizers, training, and more, with many options, creating a structure that allows to build and train a model in a very simple and encapsulated manner.

The code for this work has been written using Python's version of TensorFlow 2.3. Some of the principal classes of the Keras library that implement the layers, models and optimizers, have been imported but only serve as a structure, since they were not applicable to the special formulation of this work, they had to be completely overwritten. Moreover, the execution has been done through Jupyter Notebooks in the Google Colab cloud environment which offers a free Nvidia K80/T4 GPU. For the code, address to Appendix B.

3.2 Approximating a Function

Here we will be studying the approximating capabilities of an artificial neural network to model a function. This can be considered, in the context of this work, as simplest case of differential equation possible, the trivial case of the identity operator, whereby the artificial neural network should be adjusted to satisfy:

$$\mathcal{L}[\hat{u}(x)] = f(x) \;\Rightarrow\; \hat{u}(x) = f(x), \tag{3.1}$$

which is equivalent to simply having the artificial neural network model the external force function. Being this operator of order zero, initial/boundary conditions are irrelevant, and thus, the loss function to optimize is:

$$L(w,b) = L_1(w,b) + R = \frac{1}{N_\Omega} \sum_{i \in N_\Omega} \Big(\hat{u}(x_i; w, b) - f(x_i) \Big)^2 + R. \tag{3.2}$$

The (external force) function that we will be approximating in this section will be the polynomial $f(x) = x(x-1)$. Using this instance as an example, we will compare how well different optimizers and activation functions work at the task of modelling functions, as well as explain some of the behaviours of training. Here, the basic metric to assess performance is the relation loss function - iterations, this represents how well the model fits the solution at every step. As the loss function can have has very steep decreases in value, we will be using logarithmic scales for better representations. Moreover, for every model we will be showing a plot of the end result compared to the real solution, and in evaluation future evaluations, where it actually applies, we will also be decomposing the total loss into its components L_1 and L_2.

At last, we will be using an artificial neural network with an input layer with 1 neuron, two hidden layer with 3 and 4 neurons each, and an output layer with 1 neuron; which we will call a [3,4,1]-ANN, to approximate (3.1). First, we will start by comparing, how different activation functions work for the same network layout with different choices of activation functions. For this purpose we will use an Adam optimizer fixed to $\eta = 0.01$, $\beta_1 = 0.9$, and $\beta_1 = 0.999$, and we will see the performance for the first 10000 iterations without any adjustments. The only regularization applied will be a parameter upper bound of $10e^3$ and a gradient clipping by layer norm of 1, which as explained in the regularization section of this work, will be the standard. Initialization from here on are done as detailed in 2.5.1, using the normal distribution versions.

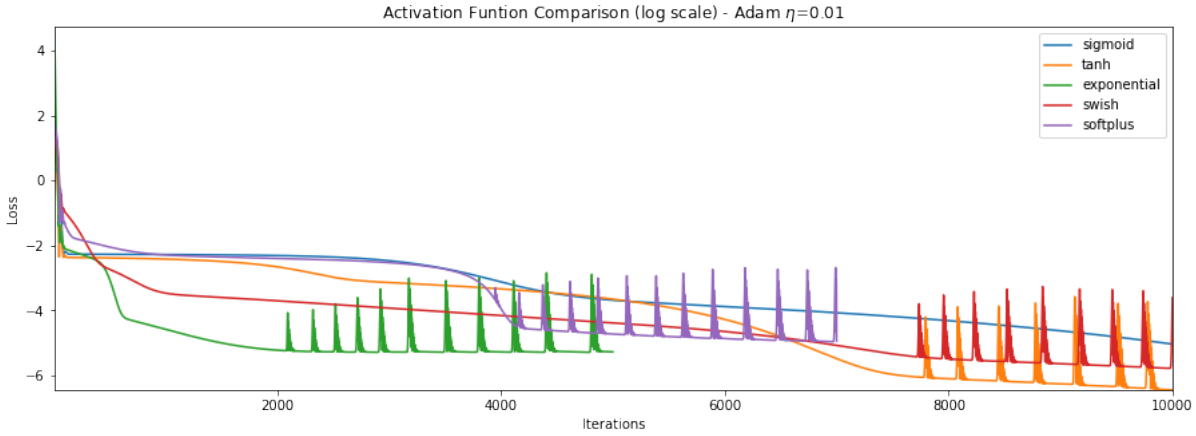

Figure 3.1: Comparison for different activation functions training performance for a [3,4,1]-ANN, with Adam $\eta = 0.01$, $\beta_1 = 0.9$, $\beta_1 = 0.999$. Log10 scale.

From Figure 3.1 we can observe that by the end of the training, for many of this activation functions, the loss value stagnates an oscillating behaviour starts to appear. This is in some sources called *saturation*, meaning that the model is unable to learn more. Fundamentally this is intrinsic to the model because we are approximating functions which may (and actually have) a very different analytic structure from the parametric model we are using. Hence, the same way it happens when we use a Taylor series expansion, where we have to truncate at some order to obtain a finite model obtaining an error, here we will also have an intrinsic minimal error to the model. However, being this a non-convex optimization problem, we do not know if these saturations correspond to reaching the intrinsic error of the model (global minimum), or if it corresponds to a local minimum or a valley. When this happens, if we have implemented early stop in the training loop, the process will stop (which happened for the exponential and softplus activations in Figure 3.1). Then, we can choose to strengthen the regularization (not very effective), or to reduce the learning rate in the optimizer or use a second order one in the hopes it is valley and we can scape it. If we use a second order method (in this work L-BFGS), and the method stops, we can be almost completely sure that at loss is in some minimum and we will not be able to scape it. This is because the stop criterion with line search is not finding any ratio in the gradient direction that can actually decrease the loss function (and line search looks for this ratio with exponential decay), thus almost guaranteeing we are in a minimum. Here is where luck of non-convex optimization comes into place, as a different initialization, or instance of the same initialization, or a simpler of more complex network layout, or an apparently worst performing optimizer can lead to a different optimization path through the loss hyper-surface, leading to a better fitting model.

In this benchmark we have used quite a minimal model to ensure it is not too overfitted (regularization can only fix some overfitting) and the loss function is quite smooth, and we have used a fairly robust optimizer. Therefore, we can assume with some confidence that the saturation corresponds at least to some minimum close to the global one. This lead us to extrapolate as a general criterion that, the activation functions that saturate the latest and at lower values, i.e. sigmoid, hyperbolic tangent and swish, are preferable to the exponential or softplus activation functions, and thus, we will prioritize the in the upcoming simulations (which does not mean that for some particular instance an exponential or softplus activation could outperform the others).

As a second part of this benchmarking, we will compare possibilities of the other crucial choice in training, the optimizers. We will be using the same set-up as before, but this time instead of fixing the optimizer, we will be fixing the activation function to be sigmoids.

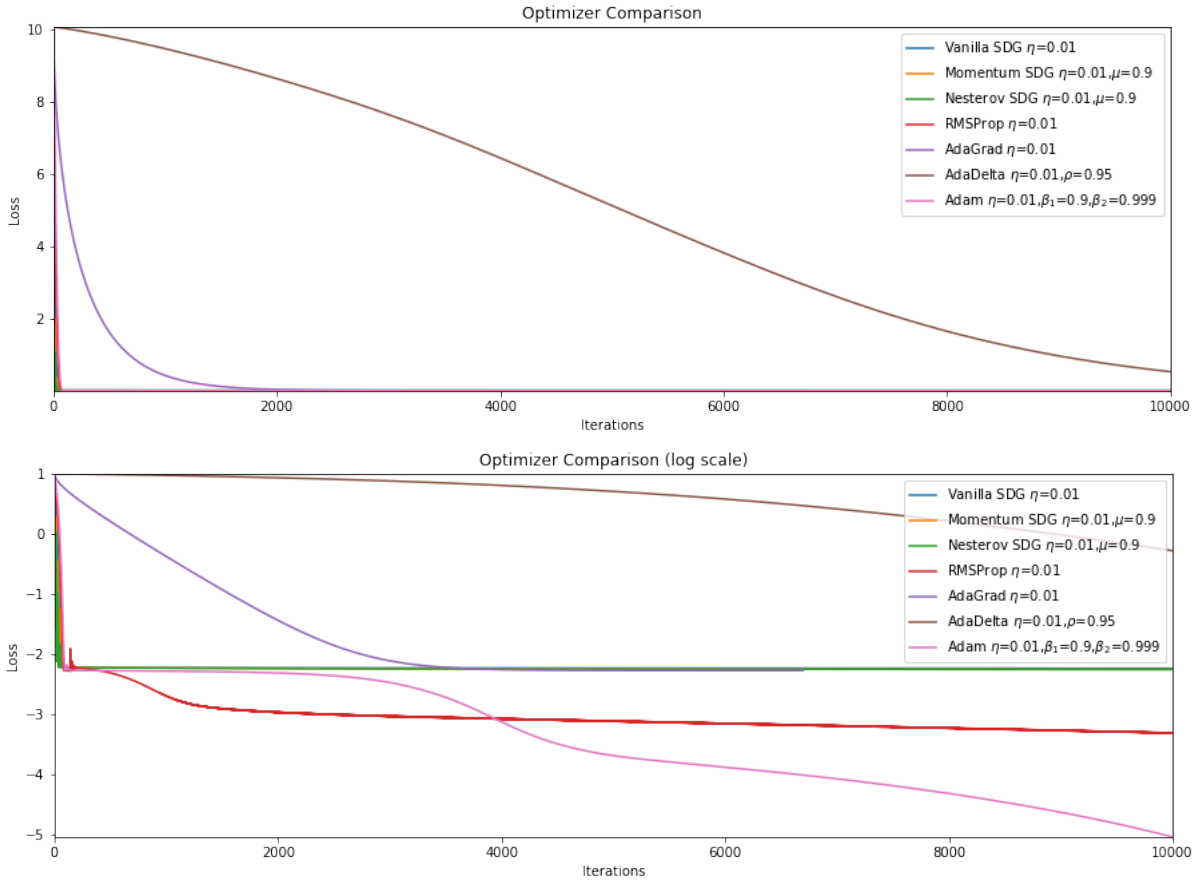

Figure 3.2: Comparison for different first order optimizers training performance for a [3,4,1]-ANN, with sigmoid activations. Lower image in log10 scale.

From these first 10000 iterations, all using a learning rate of $\eta = 0.01$ (the rest of the hyper-parameters in the optimizers can be drawn from the legend in Figure 3.2), we can see many of the behaviours expected from section 2.4.1. Vanilla, classic and Nesterov momentum SGD, all had a very similar performance reaching an almost-saturation around the same loss value, which means that at that point we should have reduced manually the learning rate. Although, it is hard to discern from Figure 3.2, Nesterov momentum had the fastest initial decrease until reaching the state of almost-saturation, followed by classic momentum and vanilla SGD, as expected.

As for the optimizers that adjust the learning component-wise, AdaGrad and AdaDelta performed poorly. In the AdaGrad case, which completely stopped learning in iteration 6696, it matches well with what we had explained in the previous chapter, the exponential accumulation of previous gradients in $G(t)$ from (2.30) vanished the update gradient. For the AdaDelta case, the method is steady but starts slow, which occurs because of the quotient of (2.34). Since the moving average of gradient and the increment $\Delta(w_t, b_t)$ are similar, the quotient that yield the actual increment ~ 1, making the starting convergence small.

On the other side of the component-wise adapting optimizers lay the RMSProp and Adam methods. These will be our preferred methods, especially Adam, since they yield very good results and have great autonomy (require little adjusting of the learning rate).

Concluding this discussion on optimizers, in the next Figure 3.4, we show the performance of the BFGS and L-BFGS, which are second order optimizers we use in this work. As it should be the behaviours are the exact same, since both, the limited memory version and the original methods are the same method, the only change being that the original method keeps the required matrices and vectors in memory, and the limited memory version stores only vectors by keeping the already multiplied matrix vector products. The line search will have 10 attempts to reduce the coefficient α. Comparing the performance of these methods to the first order ones, we see that the gains are much smaller (the figure does not use logarithmic scale) and each step takes much more time. While this is true, it is also true that the method is much more steady than the first order ones, and can further make gains in regions where first order methods get stuck. Hence, we will use these methods, especially L-BFGS, to unblock the training when first order methods stuck (mostly because of valleys).

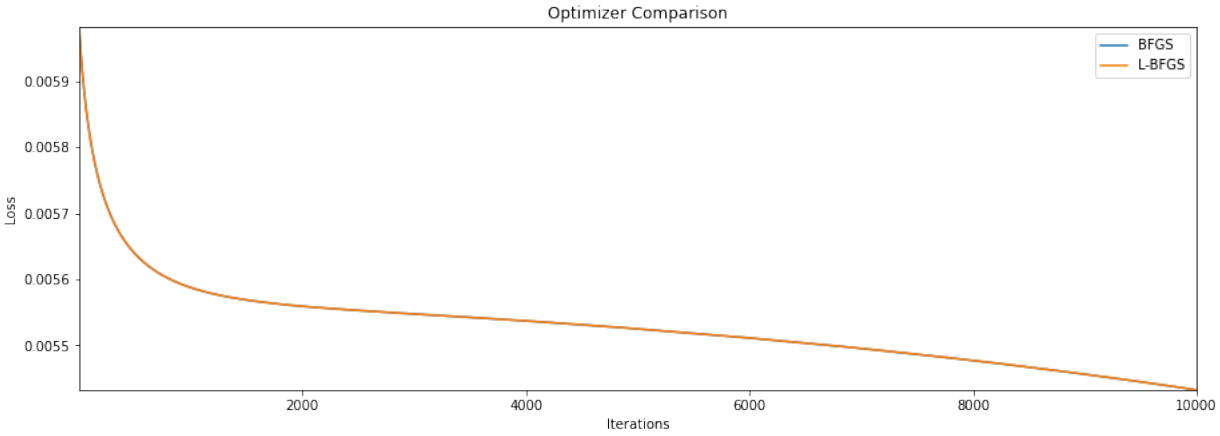

Figure 3.3: Training performance of a [3,4,1]-ANN with sigmoid activations, to fit (3.1), using BFGS and L-BFGS.

Finally we will show the end result of fitting the [3,4,1]-ANN model with sigmoids, using the Adam optimizer in the same conditions as before, for executions for 15000 iterations, with a training set of 10000 training points randomly collocated using a uniform distribution on the [0,1] segment. In the next figure will represent the model against the exact solution and the loss function for the iterations of training. We can see that model and the exact solution overlap very well, with a final total loss achieved of $L = L_{sol} = 4.331806e-07$.

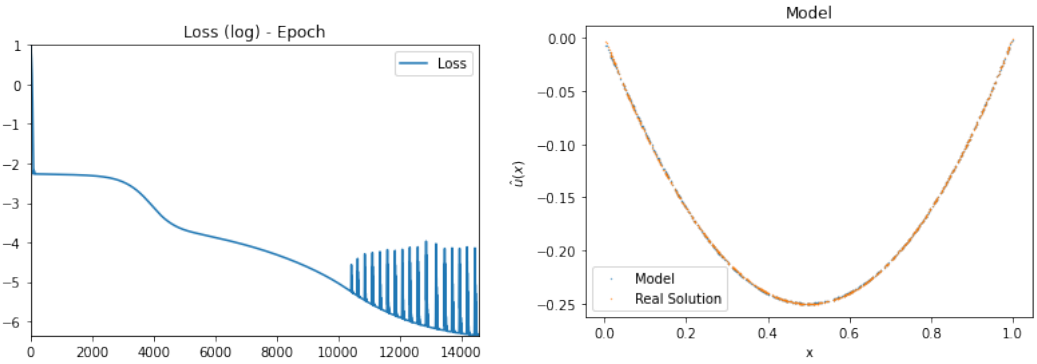

Figure 3.4: Training performance of a [3,4,1]-ANN with sigmoid activations, to fit (3.1), using Adam with $\eta = 0.01$.

3.3 Training with Multi-Objective Loss Functions

In this section we will be discussing some of the issues that arise when training multi-objective functions, and we will propose a method to adjust the gradients that will stabilize the training process.

One of the main inconveniences in this work is that our target loss function (2.1) is composed of two very different components: L_1 satisfying the differential operator and L_2 satisfying the initial/boundary conditions. However, in reality, we actually want to approximate the exact unique analytic solution to the initial/boundary problem, $u(x)$. Therefore, this means that ideally, we would want to be minimizing:

$$L_{sol}(w,b) \approx \frac{1}{N_\Omega + N_\Gamma} \sum_{x_i \in N_\Omega, N_\Gamma} \Big(u(x_i) - \hat{u}(x_i;w,b)\Big)^2. \tag{3.3}$$

Since the exact solution, $u(x)$, is generally unknown, as this is the whole purpose of using a numerical integration method, $L_{sol}(w,b)$ cannot be computed. The question thus is, how well does the target loss $L(w,b)$ (2.1) and its terms correlate with the ideal loss $L_{sol}(w,b)$ (3.3)?

Throughout this work we have introduced ad-hoc initial/boundary conditions, as well as external forces, to modify the models so that the unique solution is known and has a simple form, which means that $L_{sol}(w,b)$ could always be computed. Observing how the models in the next sections behaves, we can conclude that the relation between the target and ideal losses is not strictly correlational. Indeed, although $L(w,b)$ and $L_{sol}(w,b)$ have their only global minimum at 0, which corresponds to the model being the exact solution $\hat{u} = u$ (the desired end result and the one we would surely obtain optimizing both if the problem was purely convex), both losses describe completely different hyper-surfaces. In fact, we could argue that $L(w,b)$ is a poor choice of a loss function, since $L_1(w,b) = 0$ for any solution of the differential operator satisfying any other set of initial/boundary conditions, and $L_2(w,b) = 0$ for any function that satisfies the initial/boundary conditions. This implies that there is an infinite number of potential local minima where the optimization can fall in, or at least by the standards of this work as we expect some continuity in with respect to the operators and conditions, extremely flat valleys.

Imagine, we would want to approximate the solution of $du/dx = 2x - 1$ with $u(0) = 0$, which is nothing more than $u(x) = x^2 - x$, with a model of the type $\hat{u} = ax^2 + bx + c$. The losses corresponding to the parameter values $a = 1.5$, $b = -2$ and $c = 0.05$, for the point 0.5 of the domain take values $L_1 = 0.25$ and $L_{sol} = 0.105625$, and for the second point 0.1 of the domain take values $L_1 = 0.81$ and $L_{sol} = 0.002025$. Through this basic example we have seen that, not even in this simplified model, the loss functions at two different points are positively correlated, as in the first case $L_1 \downarrow L_{sol} \uparrow$ and in the second case $L_1 \uparrow L_{sol} \downarrow$. On top of this, we see that the magnitudes of both losses are different, and furthermore this proportion does not remain constant during training. This means that the scenario where in the first stages of training, a decrease from 10^{-1} to 10^{-4} in L_1 represents a 10^{-1} to 10^{-2} decrease in L_{sol} for the domain points, and the later on in training, a decrease from 10^{-4} to 10^{-6} in L_1 represents a 10^{-1} to 10^{-5} decrease in L_{sol} for the domain points, is very plausible. This poses one of main problems of this method for integrating differential equations compared to others, such as FEM, which is: in a general situation where the exact solution is unknown, it is not possible to know the real error of the approximated artificial neural network solution, $\hat{u}$. (The same arguments are valid with respect between L_2 and L_{sol}, and L_1 and L_2).

The fact that the terms L_1 and L_2 behave so differently between them, and during training, makes simultaneous training of these two objectives quite hard. For this work, we always chose the domain in which we want to approximate the solutions to be $\Omega = [0, 1]$ in the one dimensional case, and $\Omega = [0, 1] \times [0, 1]$ in the two dimensional case. Under this regimes, the two main behaviours of the losses after the first steps of training, when the optimization has stabilized, are that $L << L_{sol}$, and $\nabla_{(w,b)} L_2(w, b) >> \nabla_{(w,b)} L_1(w, b)$ always (at least in this work). Both are expected characteristics of the training: the first is related to the explanations of the previous paragraph as having similar information about the derivatives is not the same as being the same function; and the second occurs due to the initial/boundary conditions always having information of one derivative less than the differential operator, thus the vanishing gradient problem (2.21) with respect to input derivatives explained in 2.3.1.

Having $\nabla_{(w,b)} L_2(w, b) >> \nabla_{(w,b)} L_1(w, b)$ is a real problem. In fact if we were to optimize the model with what tools we have discussed until now, we would not be able to model anything past the divergence operator, as the artificial neural networks would fit the initial/border conditions with great precision and nothing else. At this point the optimization gets stuck in a very flat valley where the gradient is basically too weak and noisy that not even L-BFGS can find a minimum loss improvement in that direction. We will illustrate this problem with the following example shown in Figure 3.5.

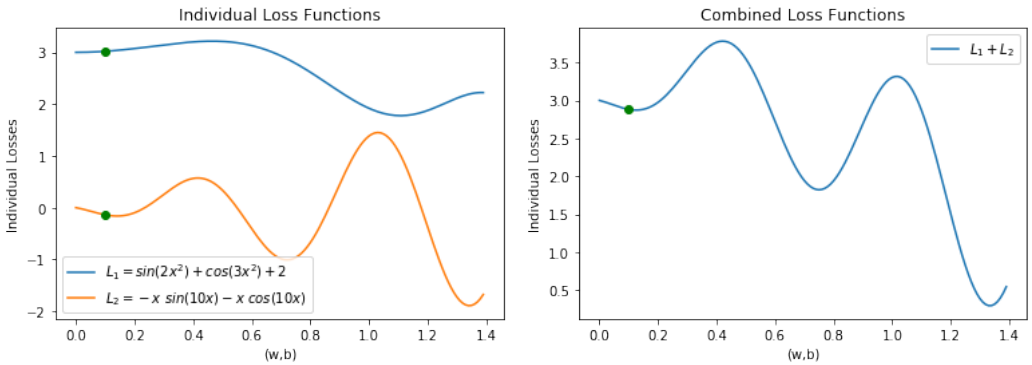

Figure 3.5: Example of possible multi-objective functions. Component and total representation.

Suppose that we wanted to find the simultaneously the minimum of the two individual loss functions, L_1 and L_2 shown in Figure 3.5 (left), and thus, we consider the global loss function composed by their addition represented in Figure 3.5 (right). The actual expressions of the functions can be drawn from the figure legends, but are irrelevant as are using them for conceptual reasons. Observe that the term L_1 is much flatter (has smaller derivatives) than the term L_2, hence, when adding both functions together, the topology of the end result becomes mostly that of L_2. When it comes to the term L_1, it acts as though it is little more than a constant moving up or down the global loss, and the whole variation is dominated by L_2. Therefore, minimizing the global loss L effectively becomes minimizing the L_2 with complete disregard to L_1. For example, if we were to optimize the global loss starting at the green point of the figure (around 0.1), we would be moving to the right and most likely get stuck in the local minimum of the second term (around 0.7), which is the same that would happen optimizing only L_2. This might be acceptable if we were only minimizing L_2, since we might accept this local minimum as a good enough result. However, at around 0.7 the term L_1 still has a very clear slope that points to the right, and quite strong for L_1 standards, although it is many orders of magnitude weaker to its counterpart in L_2. Hence, this slope for L_1 is completely ignored, meaning that we are not minimizing L_1 at all. Finally, the question is, how to balance the influence of both terms, L_1 and L_2?

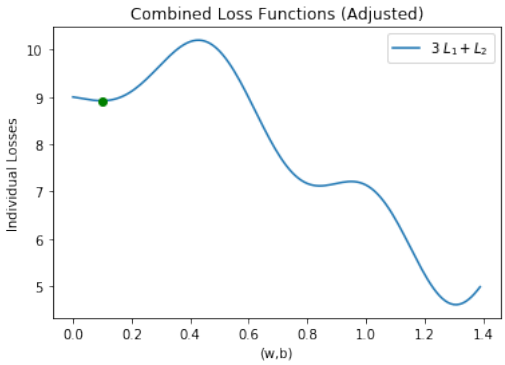

Figure 3.6: Example of possible multi-objective functions. Adjusted factors.

In Figure 3.6 we show what happens when we scale a term relatively to the other (here by a factor of 3). Note that, when multiplying a term by a factor, we also multiply their slopes and the variance it contributes to the global loss by that same factor. Hence, the final global loss function becomes a much more balanced agreement between the two terms. Here, minimizing the global loss starting at the green point and reaching the region at 0.7 does not represent such a strong sink for a gradient based approach to overcome. The region is still a good local minimum for L_2, but the slope of L_1 which disagrees showing a strong direction of decrease for L_1 to the right, now has enough influence over the global loss to have an effect. Balancing the terms makes it so that the minimum found for global loss is at least a minimum (or close enough) for each of the individual terms L_1 and L_2.

Going back to the problem of $\nabla_{(w,b)}L_2(w,b) >> \nabla_{(w,b)}L_1(w,b)$ in the regime we are dealing with, i.e. the differential operator loss L_1 is much flatter that the initial/border loss L_2, we would like to find a consistent way to balance the loss terms for any instance. The approach that we will be taking in this work to deal with this issue does not use a scale factor, as the example in Figures 3.5 and 3.6, but make use of the exact same underlying ideas.

We will still be computing the global loss as $L(w,b) = L_1(w,b) + L_2(w,b) + R(w,b)$ (2.1), and we will still be computing the each individual gradients with respect to the components $\nabla_{(w,b)}L_1(w,b)$, $\nabla_{(w,b)}L_2(w,b)$, $\nabla_{(w,b)}R(w,b)$, which is done separately since the global loss is additive. However, the difference here will be that, the gradient of the global loss will not be calculated as $\nabla_{(w,b)}L(w,b) = \nabla_{(w,b)}L_1(w,b) + \nabla_{(w,b)}L_2(w,b) + \nabla_{(w,b)}R(w,b)$ but using:

$$
\begin{aligned}
&\nabla_{(w^{[\ell]},b^{[\ell]})}L(w^{[\ell]},b^{[\ell]}) \rightarrow \\
&\quad \frac{\nabla_{(w^{[\ell]},b^{[\ell]})}L_1(w^{[\ell]},b^{[\ell]})}{||\nabla_{(w^{[\ell]},b^{[\ell]})}L_1(w^{[\ell]},b^{[\ell]})||_2^2} \cdot \min(||\nabla_{(w^{[\ell]},b^{[\ell]})}L_1(w^{[\ell]},b^{[\ell]})||_2^2, ||\nabla_{(w^{[\ell]},b^{[\ell]})}L_2(w^{[\ell]},b^{[\ell]})||_2^2) \\
&\quad + \frac{\nabla_{(w^{[\ell]},b^{[\ell]})}L_2(w^{[\ell]},b^{[\ell]})}{||\nabla_{(w^{[\ell]},b^{[\ell]})}L_2(w^{[\ell]},b^{[\ell]})||_2^2} \cdot \min(||\nabla_{(w^{[\ell]},b^{[\ell]})}L_1(w^{[\ell]},b^{[\ell]})||_2^2, ||\nabla_{(w^{[\ell]},b^{[\ell]})}L_2(w^{[\ell]},b^{[\ell]})||_2^2) \\
&\quad + \lambda \cdot \frac{\nabla_{(w^{[\ell]},b^{[\ell]})}R(w^{[\ell]},b^{[\ell]})}{||\nabla_{(w^{[\ell]},b^{[\ell]})}R(w^{[\ell]},b^{[\ell]})||_2^2} \cdot \min(||\nabla_{(w^{[\ell]},b^{[\ell]})}L_1(w^{[\ell]},b^{[\ell]})||_2^2, ||\nabla_{(w^{[\ell]},b^{[\ell]})}L_2(w^{[\ell]},b^{[\ell]})||_2^2),
\end{aligned}
\tag{3.4}
$$

which means that, before adding up the individual losses (and regularization) gradients, we renormalise the sub-gradient of each layer to the minimum between L_1 and L_2. Geometrically, we are adjusting the slopes corresponding to one layer of L_1, L_2 and R (corrected by λ), to be of the same magnitude, and thus, make each of the terms equally influential to the global loss (the regularization corrected by λ), which was the original intention.

This fixes really well the vanishing gradient problem with respect to the derivatives of the inputs, and the reason is, in some sense, analogous to why the first order methods which we called *component learning rate adapting* (Adam et al.) deal so well with the classic vanishing gradient problem. In those methods the gradients are adjusted component-wise, i.e. each parameter derivative is modified individually based on the changes for the previous vales of that same parameter. Hence, although the gradient gets deformed (we are no longer moving in maximal slope direction), it is much preferable as it allows for a much more equilibrated learning where all the parameters learn at similar rates, increasing naturally smaller parameter derivatives, when the slope in that direction is steady. Similarly, what component learning rate adapting optimizers do for parameters, (3.4) does among the global loss terms, L_1, L_2 and R, allowing to rescale in a balanced way naturally smaller sub-gradients, which otherwise would have no contribution.

Correcting the gradient before introducing it the optimizer is much convenient than altering the global loss function $L(w,b)$ terms with coefficients (recalled we already gave some arguments when discussing weight penalties in section 2.6.2). First, we only affect the step we take during the optimization, thus $L(w,b)$ and their terms remain unaltered and consistent metric of the performance of the model throughout the whole training. Second, it is really the gradient (slope of the hyper-surface) and not the actual value of the loss (which can be naturally high or low) that conditions the optimization process.

Finally, comment there is a second multi-objective behaviours due to the loss terms effectively being the sum over a collocation of points in the domain and initial/boundary conditions. A possible fix to this is to increase the power of the norm, $|\cdot| \rightarrow ||\cdot||_2^2 \rightarrow ||\cdot||_n^n$, which is known to make the problem more sensible to outliers. Since we are not using noise, and the collocation points expected outputs are deterministic, as a result of satisfying the initial/boundary value problem, making the model more sensible to fit the regions where there are more discrepancies between the model and the expected outputs can be very beneficial.

3.4 Model Simulation

At last, in this section we will be numerically integrating initial/boundary value problem with the operators already introduced in Table 1.2 (except for the identity operator which already got covered in section 3.2). As stated before we will add external forces and initial/boundary conditions, so that the exact solution, $u(x)$, has a simple polynomial form and the exact loss L_{sol} (3.3) can be computed.

The general layout for all instance will consist on several distributions of layers and neurons, activation functions, optimization methods, and regularizations, that we will vary and compare among each other. In terms of the sampling, we will always approximate the solutions in the domain $\Omega = [0, 1]$ for 1D, and $\Omega = [0, 1] \times [0, 1]$ for 2D, hence, we will always draw samples of 10000 and 1000 points, for the domain and initial/boundary training sets, and validation sets, respectively, from $\sim \mathcal{U}(0, 1)$ distributions. While a very small collocation sample may lead to very poorly fitted model, there is a point where increasing the size of the sample does not improve the result of the model, which has to be located with trial and error. For most of the instances in this work it is probably around 1000 points, thus we have increased it 10 fold for safety reasons. Also implicitly applied through all the instances, there will be upper bound parameter constrains of 10^3, gradient clipping by layer with norm 1, early stop evaluation every 500 training steps (or epochs), and the gradient adjustment (3.4).

Each of the instances will be increasing in complexity. Furthermore each instance may contain optional comparisons for some of the aspects that we have already discussed for benchmarking reasons.

3.4.1 Model 1: The 1D Divergence Operator

For the first proper initial/boundary problem we will be approximating the 1D divergence operator (derivative equals to an external force), which is one of the most simple models that we can generate. The exact instance of this ODE will be:

$$\begin{aligned} \nabla_{(x)} \cdot u(x) = \frac{\partial u(x)}{\partial x} &= 2x - 1, \\ u(0) &= 0, \end{aligned} \tag{3.5}$$

which has exact solution $u(x) = x^2 - x$. The solution can be easily obtain through the separation method by integrating both sides:

$$\begin{aligned} \int_0^x \frac{\partial u(x)}{\partial x}\, dx &= \int_0^x 2x - 1\, dx, \\ u(x) \,\big|_0^x &= x^2 - x \,\big|_0^x, \\ u(x) - u(0) &= (x^2 - x) - (0^2 - 0), \\ u(x) &= (x^2 - x). \end{aligned} \tag{3.6}$$

Here the loss function we are optimizing is:

$$L(w, b) = \frac{1}{N_\Omega} \sum_{1 \le i \le N_\Omega} \left(\frac{\partial \hat{u}(x_i; w, b)}{\partial x} - (2x_i - 1) \right)^2 + \left(\hat{u}(0; w, b) - 0 \right)^2 + R(w, b). \tag{3.7}$$

For this instance we will focus on the effect in the models of varying the activation functions. We will be training 3 model with sigmoid, hyperbolic tangent and swish activation functions respectively. All model will be trained on 3000 iterations (epochs), using a [1,5,5,1]-ANN, with no regularization, and using Adam with $\eta = 0.01$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Table 3.1 shows the end result losses, and Figure 3.7 plots the performance of the training (the losses for the figure have been broken into each of its components).

Activation	L	L_{sol}
Sigmoid	$1.66 \cdot 10^{-4}$	$1.71 \cdot 10^{-6}$
Tanh	$1.20 \cdot 10^{-4}$	$4.02 \cdot 10^{-7}$
Swish	$6.41 \cdot 10^{-5}$	$1.75 \cdot 10^{-6}$

Table 3.1: Results of 3 models trained for a [1,5,5,1]-ANN scheme, with no regularization, using Adam with $\eta = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, on 3000 epochs. (3.5)

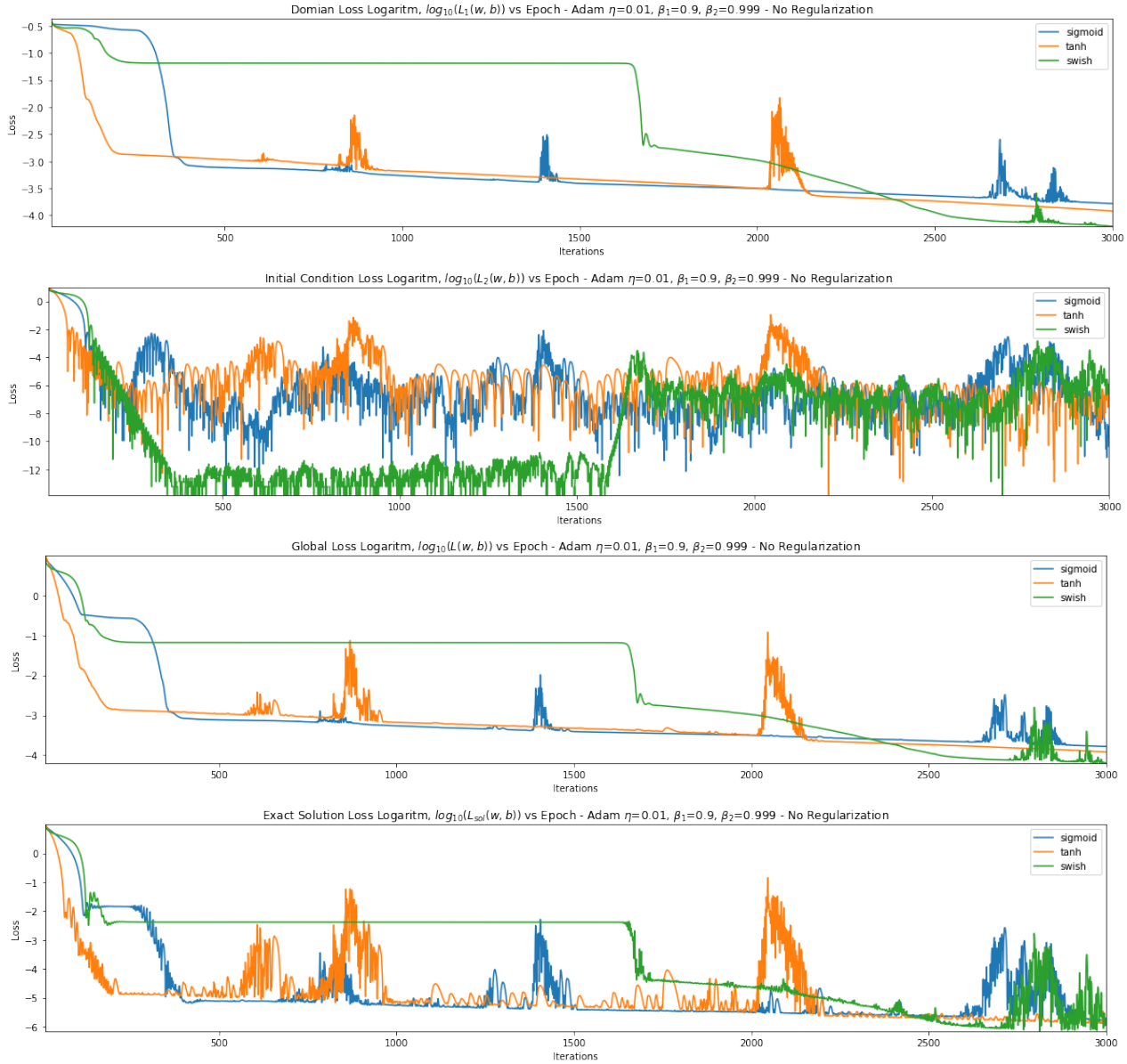

Figure 3.7: Training performance of 3 models trained for a [1,5,5,1]-ANN scheme, with no regularization, using Adam with $\eta = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, on 3000 epochs. (3.5)

From the previous results we can see some interesting behaviours. First of all, the worst performing activation (the swish) with regards to the solution loss L_{sol}, was actually the best in terms of the objective loss L; and the best performing activation (the tanh) with regards to the solution loss L_{sol}, was not the best in terms of the objective loss L. Also, we see that there is correlation between L (3rd plot of Figure 3.7) and L_{sol} (4th plot of Figure 3.7); and that L_1 dominates over L_2, meaning L_2 is natively much smaller that L_1. A well as this, we observe that the swish model had a much later initial decay than the other two, but the three of them start saturating at the same time. All these are expected behaviours that we have explained before.

Finally, in the next figure we plot the output of the best performin model (the one with tanh activations), against the exact solution. Note that just in 3000 epochs (2min) the match is almost perfect.

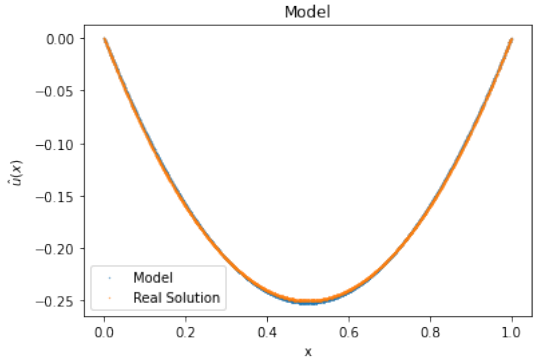

Figure 3.8: Final results. Best performing trained model (tanh) for (3.7) against the exact solution.

3.4.2 Model 2: The 2D Divergence Operator

Here we will take the previous model to the next level adding a dimension, and in doing so we will consider our first PDE. Still, this will be a very simple problem. The instance we will be considering first is:

$$\begin{aligned} \nabla_{(x,y)} \cdot u(x,y) \cdot \mathbf{1} = \frac{\partial u(x,y)}{\partial x} + \frac{\partial u(x,y)}{\partial y} = (2x-1)\cdot(y^2-y) + (x^2-x)\cdot(2y-1), \\ u(x,0) = 0, \qquad x \in (-\infty, \infty), \end{aligned} \tag{3.8}$$

which has exact solution $u(x,y) = (x^2-x)\cdot(y^2-y)$. The loss function for (3.8) would be:

$$\begin{aligned} L(w,b) = \frac{1}{N_\Omega} \sum_{1\le i\le N_\Omega} \Bigg(\frac{\partial \hat{u}(x_i,y_i;w,b)}{\partial x} + \frac{\partial \hat{u}(x_i,y_i;w,b)}{\partial y} - (2x_i-1)(y_i^2-y_i) \\ -(x_i^2-x_i)(2y_i-1)\Bigg)^2 + \frac{1}{N_\Gamma} \sum_{1\le i\le N_\Gamma} \left(\hat{u}(x_i,0;w,b) - 0\right)^2 + R(w,b). \end{aligned} \tag{3.9}$$

However, there is an issue when using the (3.9) loss function. The border conditions are described by a curve for $x \in (-\infty, \infty)$, but effectively, we cannot draw samples from such a wide range. Since, we are limiting ourselves to approximating the solutions in the domain to $\Omega = [0,1] \times [0,1]$ for practical reasons, we will sample x from $(-10, 10)$ for the L_2 term.

The following Figure 3.9 shows the results of training a model under the previous assumptions (specifics in the caption). Observe that the left plot shows that, the solution approximated by the model has two separate regions, one approximating really well the exact solution, and another one that does not by a large margin. If we turn to the right plot we see that the MSE of the individual points with respect to the differential operator/external force is very even, meaning every point is equally well fitted.

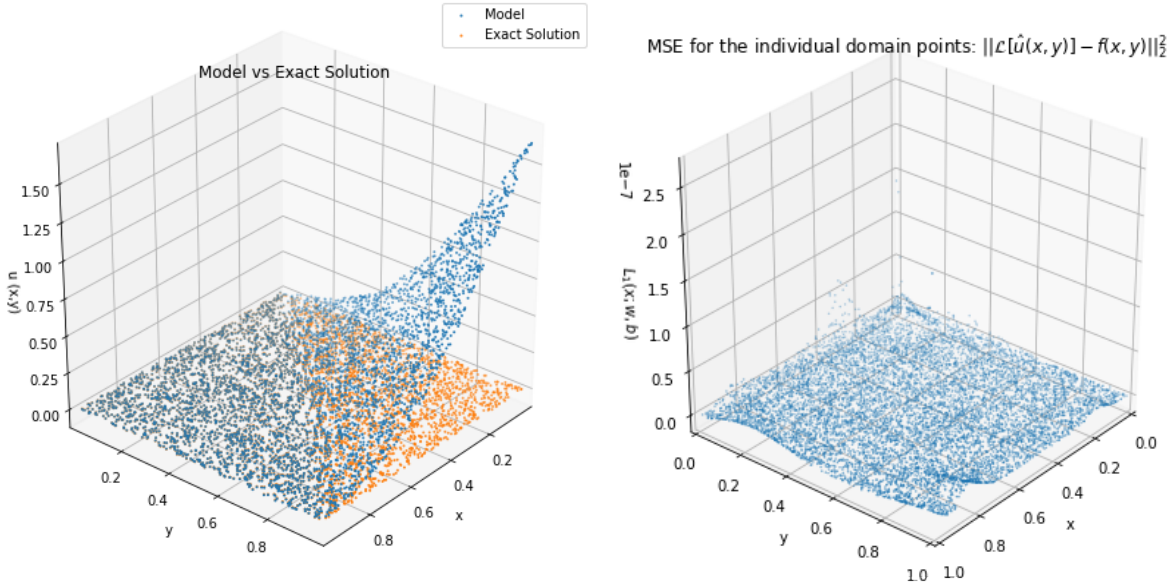

Figure 3.9: Result of a [1,10,10,1]-ANN model and tanh activations, trained with no regularization, using Adam with $\eta = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, on 12000 epochs. Left plot: model against exact solution. Right plot MSE error of the model, for each point in the domain.

This occurs because, in practice, when we draw a sample points for the border conditions, we are limiting ourselves to $x \in (-10, 10)$. Hence, for all purposes we are solving (3.8) with boundary conditions $u(x, 0) = 0,\ x \in (-10, 10)$, which are no longer Cauchy conditions and do not guarantee uniqueness. The solution we want to find is also a solution of the problem we are fitting in practice, but there are many more. In fact, what we see in Figure 3.9 is the artificial neural network overlapping two different solutions of the problem (the one closest to $y = 0$ is the one we would want). Thus, this a good example of what happens when integrating a problem which is not well-posed.

In order to fix this issue we will change the Cauchy "border" conditions, which for infinite domains would be simply an open curve, to its finite domain version, which requires the information over the border. This means, for $\Omega = [0, 1] \times [0, 1]$, changing (3.8) to:

$$\begin{aligned} \nabla_{(x,y)} \cdot u(x,y) \cdot \mathbf{1} = \frac{\partial u(x,y)}{\partial x} + \frac{\partial u(x,y)}{\partial y} &= (2x-1)\cdot(y^2-y) + (x^2-x)\cdot(2y-1), \\ u(x,0) = 0, \quad u(x,1) = 0, \quad & x \in (0,1), \\ u(0,y) = 0, \quad u(1,y) = 0, \quad & y \in (0,1), \end{aligned} \tag{3.10}$$

with its respective change in the loss function (3.9). The solution for this problem is the same as before.

In the actual experiments for (3.10) we will take more interesting features to compare than the simple activation functions of the previous instance. Here we will analyse the effects of the size of the artificial neural network and the regularization.

When choosing an artificial neural network architecture, the general rule is that deeper neural networks are able to learn more complex functions, although at a greater cost of training [54]. Furthermore, papers such as [55], focused on learning polynomials with artificial neural networks, suggest that a fully-connected network with a single hidden layer with a number of nodes equals to the degree of the polynomial, would be enough to learn a polynomial (this is an rough and imprecise extraction of what [55] states, but holds for the most part). In this work though, we have been using two hidden layers so far (an will keep using them), and a much larger number of neurons than the theoretic minimal suggests for the underlying solutions we want to approximate. The reason for doing this is to better account for the information of the derivatives during training and make use of regularization techniques, to obtain better minima.

For this instance (3.10) we will be training 6 models, all using hyperbolic tangent activations and are trained on Adam with $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, on 8000 epochs. The models will either have a [1,10,10,1]-ANN structure or a [1,40,40,1]-ANN structure; and be trained using no regularization, the custom regularization (2.58) with $\lambda = 1$, or a Tikhonov regularization with $\lambda = 1$; which make for a total of 6 combinations. Moreover, all the models with the same architecture have been initialized with exactly the same parameters. This has been done to root out the possible effect of luck for starting at a slightly better point for the optimization, and ensure the difference is training are caused by the regularization.

In Table 3.2 we show the final results of the models, and in Figures 3.10 and 3.11 we show the performance of the training. First, we observe for these kind of problems Tikhonov regularizations do not work well and their training incurs in early stopping. For the (2.58) custom regularization we see that, in the smaller [1,10,10,1]-ANN model, the training is actually hindered and yields awful results, but used the larger [1,40,40,1]-ANN model it outperforms any other set-up. This, is due to what we have already explained in section 2.6, that regularizations clamp down on the extra degree of freedom overfitting the model. Hence, for the smaller model which is adequately parametrized, it becomes an extra condition drawing resources form the model, while for the larger model it narrows the parameters to the fit the model. Furthermore, not only the larger model with regularization outperforms the smaller one without, but if we compare their performances from Figures 3.10 and 3.11, we note that by the end of the training, the smaller model has saturated (stagnated), while the larger is still steadily decreasing (thus, have more room for improvement). This shows that is much preferable to have a larger model with regularization than simply a well adjusted one.

Architecture - Regularization Technique	L	L_{sol}
[1,10,10,1]-ANN - No Regularization	$1.04 \cdot 10^{-4}$	$7.52 \cdot 10^{-6}$
[1,10,10,1]-ANN - (2.58) Regularization with $\lambda = 0.1$	$7.18 \cdot 10^{-3}$	$2.84 \cdot 10^{-4}$
[1,10,10,1]-ANN - Tikhonov Regularization with $\lambda = 0.1$	$6.33 \cdot 10^{-4}$	$4.19 \cdot 10^{-5}$
[1,40,40,1]-ANN - No Regularization	$6.36 \cdot 10^{-4}$	$1.95 \cdot 10^{-5}$
[1,40,40,1]-ANN - (2.58) Regularization with $\lambda = 0.1$	$2.93 \cdot 10^{-4}$	$2.32 \cdot 10^{-6}$
[1,40,40,1]-ANN - Tikhonov Regularization with $\lambda = 0.1$	$3.88 \cdot 10^{-3}$	$7.91 \cdot 10^{-5}$

Table 3.2: Results of 6 models with different architectures, trained for (3.10), using Adam with $\eta = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, on 8000 epochs and different regularization techniques.

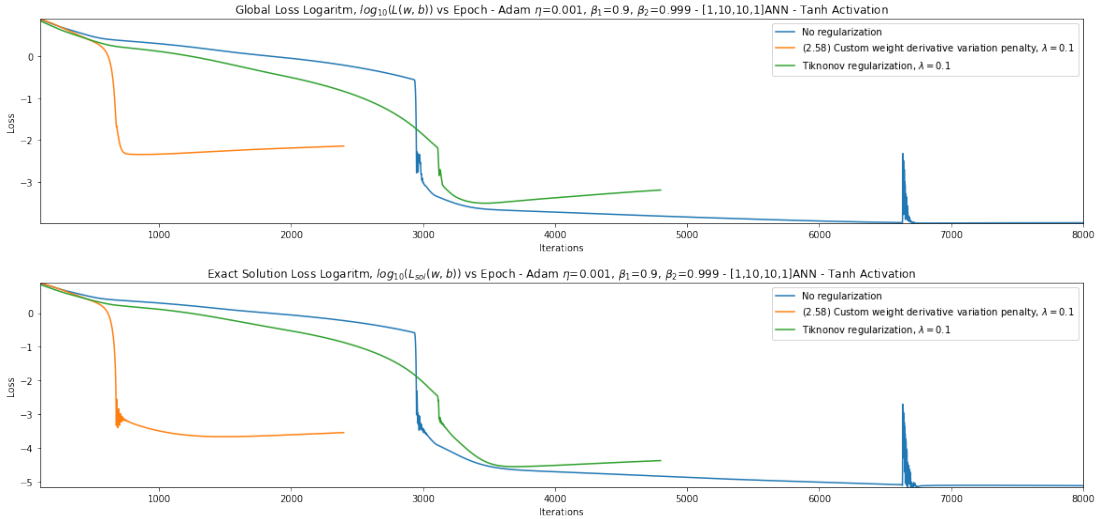

Figure 3.10: Comparison of different regularization techniques in training performance of 3 models trained for a [1,10,10,1]-ANN scheme, using Adam with $\eta = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, on 8000 epochs. (3.10)

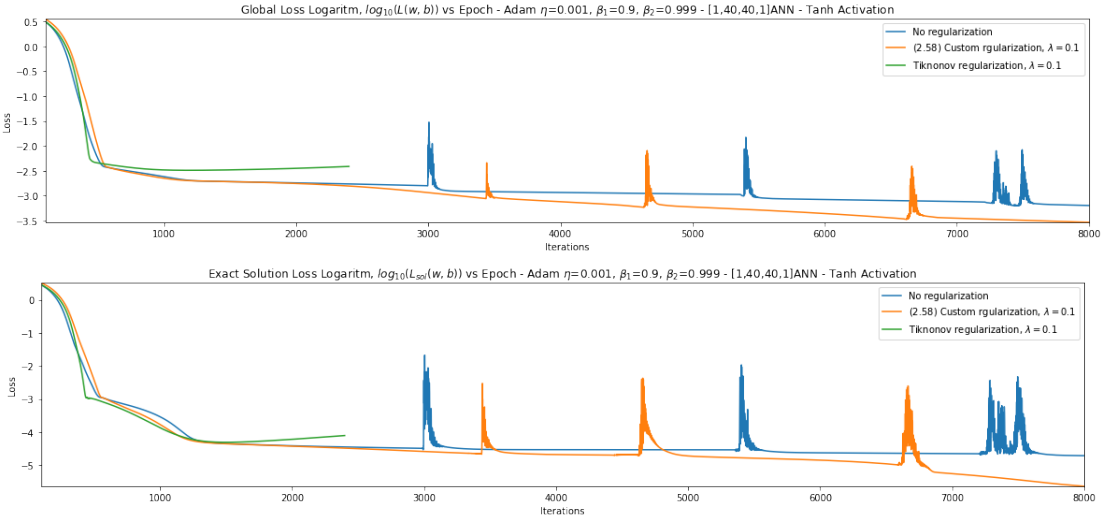

Figure 3.11: Comparison of different regularization techniques in training performance of 3 models trained for a [1,40,40,1]-ANN scheme, using Adam with $\eta = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, on 8000 epochs. (3.10)

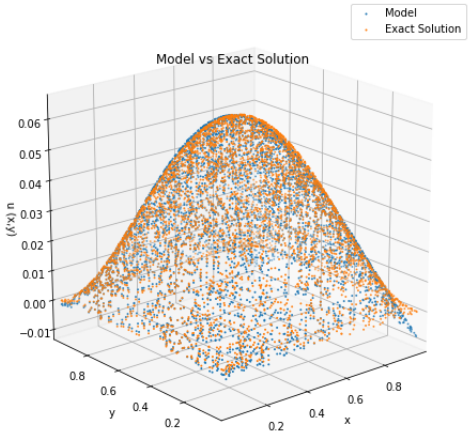

Figure 3.12: Final results of the best performing trained model ([1,40,40,1]-ANN, trained with the custom regularization (2.58)) for (3.7) against the exact solution.

3.4.3 Model 3: The 2D Laplacian Operator

At this point we will complicate a bit more the differential operator by considering second order derivatives. Thus, we will consider the following boundary value problem for the Laplacian operator in 2 dimensions:

$$\Delta u(x,y) = \frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 2\cdot(y^2-y) + 2\cdot(x^2-x),$$

$$u(\Gamma) = g_1(\Gamma): \begin{cases} u(x,0)=0,\ u(x,1)=0, & x\in(0,1), \\ u(0,y)=0,\ u(1,y)=0, & y\in(0,1), \end{cases}$$

$$\frac{\partial u(\Gamma)}{\partial(x,y)}\cdot n(\Gamma) = g_2(\Gamma): \begin{cases} \dfrac{\partial u(x,0)}{\partial y} = -(x^2-x),\ \dfrac{\partial u(x,1)}{\partial y} = (x^2-x), & x\in(0,1), \\ \dfrac{\partial u(0,y)}{\partial x} = -(y^2-y),\ \dfrac{\partial u(1,y)}{\partial x} = (y^2-y), & y\in(0,1), \end{cases} \tag{3.11}$$

which has exact solution $u(x,y) = (x^2-x)\cdot(y^2-y)$, as with the previous problem. The form of problem (3.11) in its general form, for any dimension and external force, constitutes what is called the Poisson equation, which is important throughout physics, as it is the interpretation of Gauss Law in terms of potentials.

Before training an artificial neural network to fit this model, we would like to make a brief note regarding the coding of higher order derivatives in TensorFlow. Looking at the official documentation of TensorFlow, the method given to obtain higher order derivatives in one variable is by nesting auto-differentiations calls. However, note that, TensorFlow is used in a context of training artificial neural networks, thus when auto-differentiating twice we obtain:

$$\begin{aligned} \nabla_{(x)} f(x_1,...,x_n) &= \left(\frac{\partial f}{\partial x_1},...,\frac{\partial f}{\partial x_n}\right), \\ \nabla^2_{(x)} f(x_1,...,x_n) &= \left(\frac{\partial}{\partial x_1}\sum_{m=1}^{n}\frac{\partial f}{\partial x_m},...,\frac{\partial}{\partial x_n}\sum_{m=1}^{n}\frac{\partial f}{\partial x_m}\right), \end{aligned} \tag{3.12}$$

which is not the Laplacian. There are two ways to overcome this issue: either use the *unstack* and *stack* functions to decouple the inputs and compute the gradients tracking only an individual variable (the option we have used in the code); or to use the *hessian* function to compute the Hessian matrix and then compute the trace, which is highly inefficient as we only require the elements in the diagonal. Without [56] where this observation is pointed out, we would not have been able to carry out this simulation.

At this point we have already experimented on all the principal options and hyper-parameter choices covered in this work, and we have studied their performance. So, from now on, we will be dropping the comparisons and limit ourselves to simply solve the next models with the best possible set-up best on what we have discussed.

The artificial neural network model trained for (3.11) has achieved a final global loss of $L = 1.23\cdot10^{-3}$ and final loss with respect to the solution of $L_{sol} = 4.25\cdot10^{-6}$. This model consisted of a [1,40,40,1]-ANN with tanh activations, trained for 6000 epochs (when early stop triggered), using Adam with $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and the custom regularization (2.58) with $\lambda = 0.1$. The results can be seen in the following Figure 3.13.

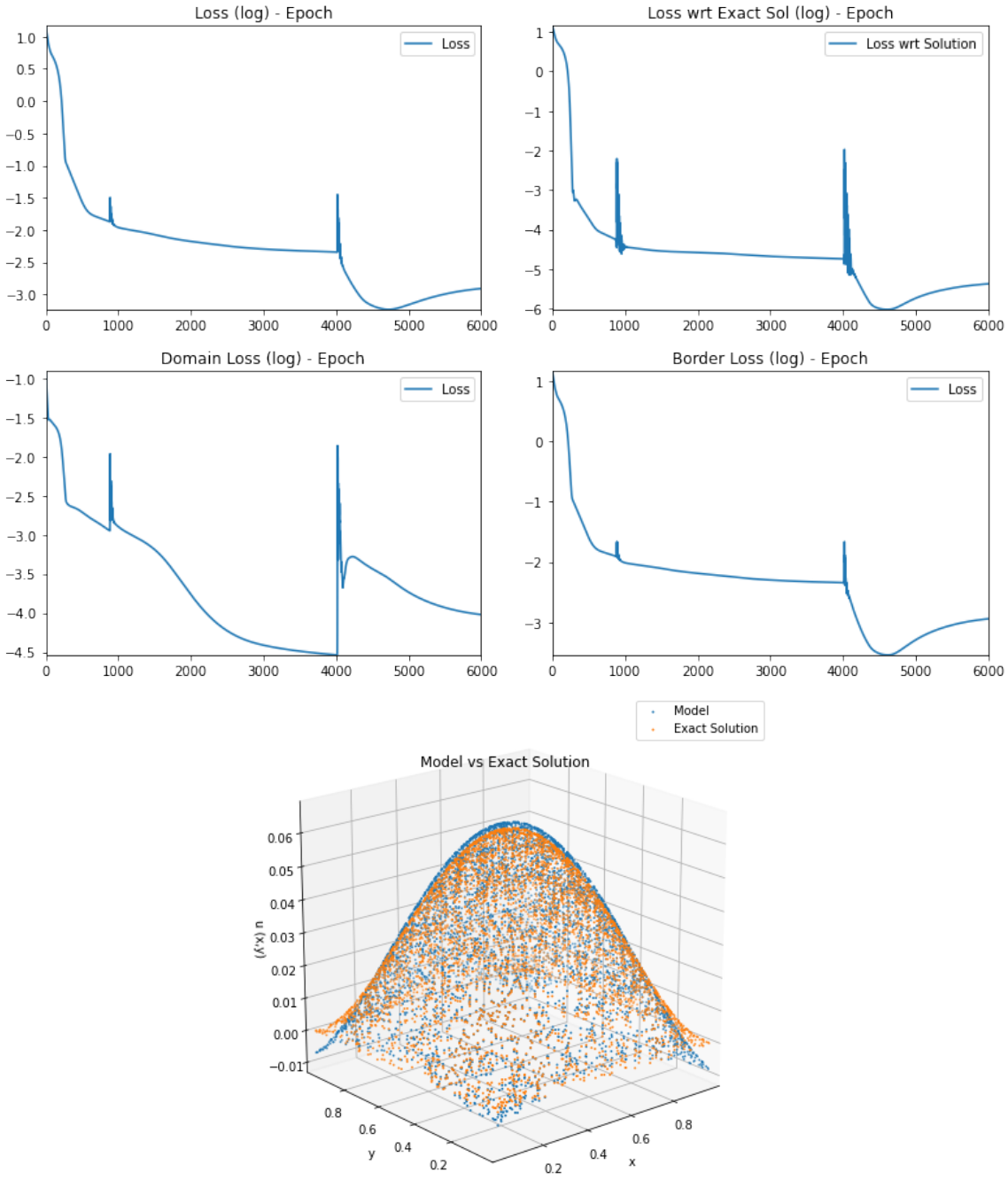

Figure 3.13: Results and performance of the model trained for (3.11).

3.4.4 Model 4: The 1D Advection Operator

For this simulation we step down from the 2D PDE cases, to go back to an ODE. The reason for this downgrade is to explain a certain issue occuring for this operator. This issue is one that happens for the final case of this section, the 2D Burgers operator, and since the advection operator we are proposing coincides with the Burgers operator in 1D, we see this as a much simpler example to introduce a discussion.

The initial value problem we want to consider is:

$$\begin{aligned} u(x) \cdot \nabla_{(x)} \cdot u(x) = u(x) \cdot \frac{\partial u(x)}{\partial x} &= 2x^3 - 3x^2 + x, \\ u(0) &= 0, \end{aligned} \tag{3.13}$$

which has exact solution $u(x) = x^2 - x$, same as the 1D divergence case. This problem look like falling under the Cauchy-Kovalevskaya conditions, so existence and uniqueness should be guaranteed. However, there is a subtlety hidden here. If we write the equation in its canonical form (isolating the higher derivative), which is required to apply the Cauchy-Kovalevskaya theorem,

$$\frac{\partial u(x)}{\partial x} = \frac{2x^3 - 3x^2 + x}{u(x)}, \tag{3.14}$$

we note that the equation is quasi-linear and its terms are analytic everywhere except for the zeroes of $u(x)$. Hence we have local existence and uniqueness almost everywhere, but since it can fail in some points, we cannot build a unique global solution using the theorem. This can be verified easily in this case, as the differential equation is separable and can be solved easily by separations of variables method:

$$\begin{aligned} \int_0^x u(x) \frac{\partial u(x)}{\partial x}\, dx &= \int_0^x 2x^3 - 3x^2 + x\, dx, \\ \frac{1}{2}(u(x))^2 \Big|_0^x &= \frac{1}{2}x^4 - x^3 + \frac{1}{2}x^2 \Big|_0^x, \\ \frac{1}{2}(u(x))^2 - 0 &= \frac{1}{2}x^4 - x^3 + \frac{1}{2}x^2 - 0, \\ u(x) &= \pm\sqrt{x^4 - 2x^3 + x^2} = \pm(x^2 - x). \end{aligned} \tag{3.15}$$

Looking at Figure 3.14 we observe that the solutions intersect (hence, are not unique) at the roots of $u(x)$.

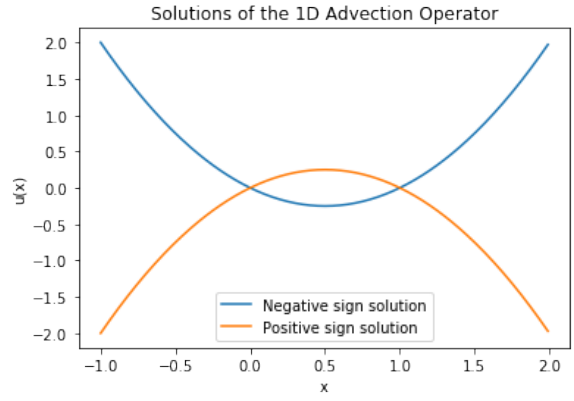

Figure 3.14: Positive and negative sign solutions of 3.13.

To fix this issue and fix a solution, it is enough to provide information about an extra derivative of one more order than the required by the Cauchy conditions. Therefore, the well-posed problem that we will consider will be:

$$\begin{aligned} u(x) \cdot \nabla_{(x)} \cdot u(x) = u(x) \cdot \frac{\partial u(x)}{\partial x} &= 2x^3 - 3x^2 + x, \\ u(0) = 0, \qquad u'(0) &= -1. \end{aligned} \tag{3.16}$$

The artificial neural network model trained for (3.16) has achieved a final global loss of $L = 1.05 \cdot 10^{-3}$ and final loss with respect to the solution of $L_{sol} = 2.74 \cdot 10^{-7}$. This model consisted of a [1,20,20,1]-ANN with sigmoid activations, trained for 3000 epochs using Adam with $\eta = 0.01$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and the custom regularization (2.58) with $\lambda = 0.1$. The results can be seen in the following Figure 3.16.

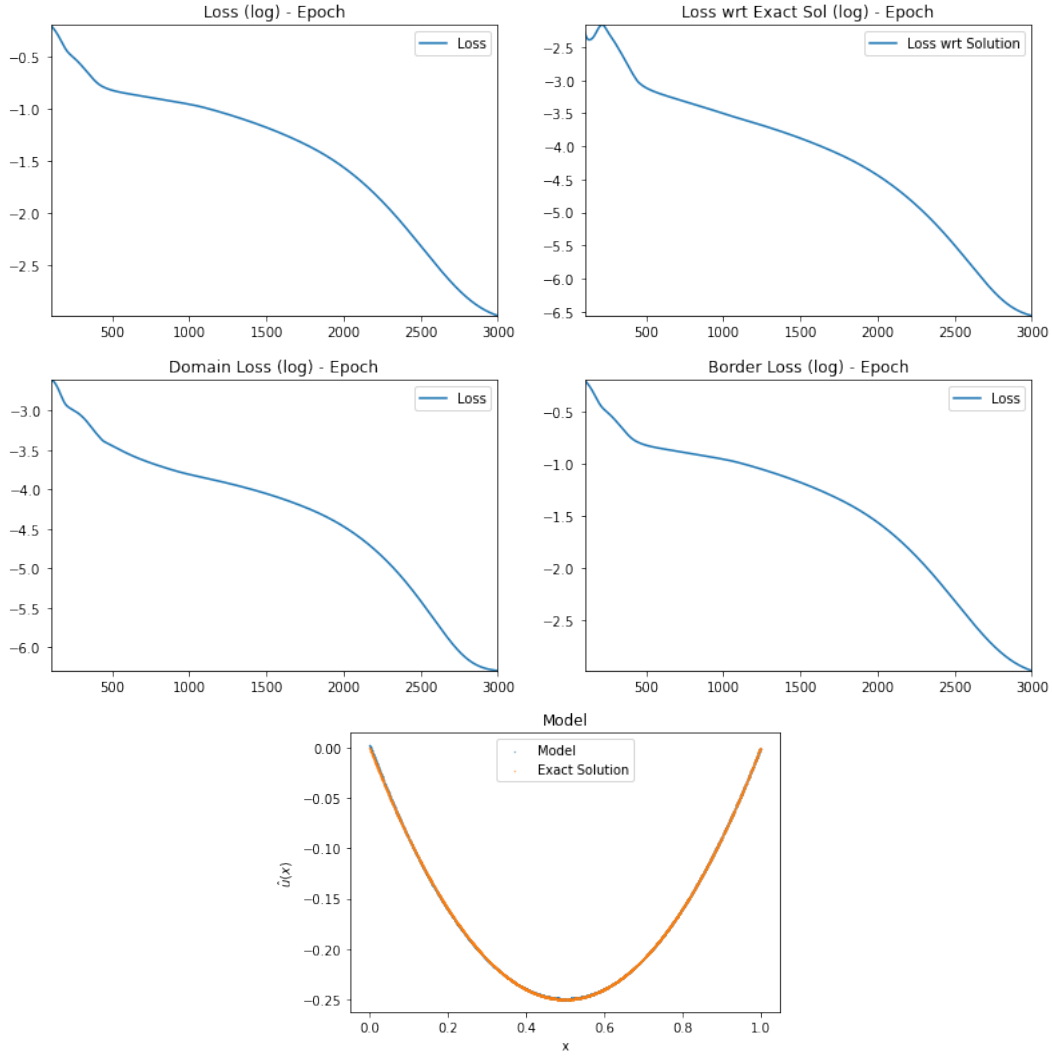

Figure 3.15: Results and performance of the model trained for (3.16).

3.4.5 Model 5: The 2D Clairaut Operator

The Clairaut operator can be seen as an upgrade to the 2D Advection case. It may not be much more complicated than what we have seen before, but it is the first PDE with non-constant coefficients that we integrate in this work. We pose its boundary problem as:

$$\begin{aligned}(x,y) \cdot \nabla_{(x,y)} u(x,y) = \; & x \cdot \frac{\partial u(x,y)}{\partial x} + y \cdot \frac{\partial u(x,y)}{\partial y} \\ & = x \cdot (2x-1) \cdot (y^2 - y) + (x^2 - x) \cdot y \cdot (2y-1), \\ u(x,0) = 0, \quad & u(x,1) = 0, \qquad x \in (0,1), \\ u(0,y) = 0, \quad & u(1,y) = 0, \qquad y \in (0,1),\end{aligned} \tag{3.17}$$

with solution $u(x,y) = (x^2 - x) \cdot (y^2 - y)$, as always.

The artificial neural network model trained for (3.17) has achieved a final global loss of $L = 3.04 \cdot 10^{-6}$ and final loss with respect to the solution of $L_{sol} = 3.96 \cdot 10^{-6}$. This model consisted of a [1,40,40,1]-ANN with tanh activations, trained for 8000 epochs, using Adam with $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and the custom regularization (2.58) with $\lambda = 0.1$. The results can be seen in the following Figure 3.16.

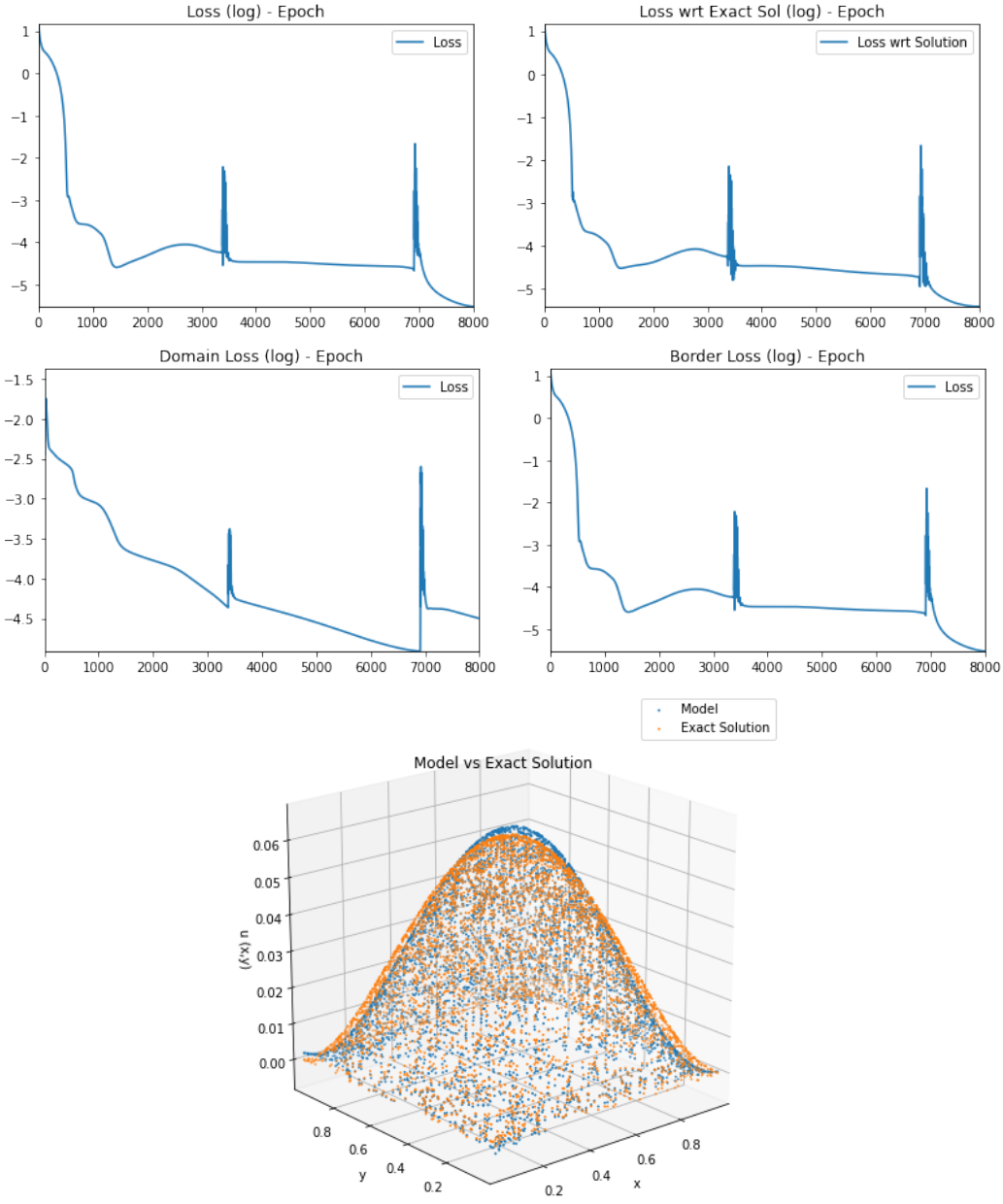

Figure 3.16: Results and performance of the model trained for (3.17).

3.4.6 Model 6: The 2D Burgers Operator

Finally, we will numerically integrate the last, and most complex, boundary problem of this work. This would be the 2D Burgers operator, and it can be regarded as the multi-dimensional case of the advection operator. While the advection operator is applied on scalar functions, the Burgers operator is applied on vector fields.

The boundary value problem requires, as explained in the 1D advection case, information on one more derivative than in the Cauchy conditions, and thus will be defined by:

$$\begin{aligned}\mathbf{u}(x,y)\cdot\nabla_{(x,y)}\mathbf{u}(x,y) &= (u_x(x,y),u_y(x,y))\begin{pmatrix}\dfrac{\partial u_x(x,y)}{\partial x} & \dfrac{\partial u_y(x,y)}{\partial x}\\ \dfrac{\partial u_x(x,y)}{\partial y} & \dfrac{\partial u_y(x,y)}{\partial y}\end{pmatrix}\\ &= \begin{pmatrix}(x^2-x)\cdot(y^2-y)[(2x-1)\cdot(y^2-y)+(x^2-x)\cdot(2y-1)]\\ (x^2-x)\cdot(y^2-y)[(2x-1)\cdot(y^2-y)+(x^2-x)\cdot(2y-1)]\end{pmatrix}\end{aligned}$$

$$\mathbf{u}(\Gamma)=\mathbf{g_1}(\Gamma):\begin{cases}u_x(x,0)=u_y(x,0)=0,\ u_x(x,1)=u_y(x,1)=0, & x\in(0,1),\\ u_x(0,y)=u_y(0,y)=0,\ u_x(1,y)=u_y(1,y)=0, & y\in(0,1),\end{cases}$$

$$\frac{\partial\mathbf{u}(\Gamma)}{\partial(x,y)}\cdot\mathbf{n}(\Gamma)=\mathbf{g_2}(\Gamma):\begin{cases}\dfrac{\partial u_x(x,0)}{\partial y}=-(x^2-x),\ \dfrac{\partial u_x(x,1)}{\partial y}=(x^2-x), & x\in(0,1),\\ \dfrac{\partial u_x(0,y)}{\partial x}=-(y^2-y),\ \dfrac{\partial u_x(1,y)}{\partial x}=(y^2-y), & y\in(0,1),\\ \dfrac{\partial u_y(x,0)}{\partial y}=-(x^2-x),\ \dfrac{\partial u_y(x,1)}{\partial y}=(x^2-x), & x\in(0,1),\\ \dfrac{\partial u_y(0,y)}{\partial x}=-(y^2-y),\ \dfrac{\partial u_y(1,y)}{\partial x}=(y^2-y), & y\in(0,1),\end{cases}\tag{3.18}$$

with the vector field $\mathbf{u}(x)=((x^2-x)\cdot(y^2-y),(x^2-x)\cdot(y^2-y))$ as its solution.

We have called the differential operator in (3.18) as the Burgers operator (although it is not the real burgers operator) because the most prominent place we can find this operator is in the inviscid Burgers equation, where it appears along an extra $\partial/\partial t$. Hence, we can think of (3.18) as the system that solves for the steady-state solutions $u_t=0$ of an inviscid Burgers equation.

All the attempts to obtain the two components of the solution as an artificial neural network with 2 outputs resulted in failure. The models always seemed to get stuck in very bad local minima and the process was slow. We assume that the reasons for this lies in the fully-connected nature of the architecture we are using, since we were trying to adjust 2 functions on the same parameters (despite being the same in this case). Hence, we have taken the much efficient approach of approximating each component of the solution with a separate neural network, and obtained much better results. We will only show the results for the first component since the operator and external force are symmetric, therefore the behaviours for both components are the almost same.

The artificial neural network model trained for (3.18) has achieved a final global loss of $L=6.21\cdot10^{-6}$ and final loss with respect to the solution of $L_{sol}=8.75\cdot10^{-5}$. This model consisted of a [1,40,40,1]-ANN with tanh activations, trained for 8000 epochs, using Adam with $\eta=0.001$, $\beta_1=0.9$, $\beta_2=0.999$, and the custom regularization (2.58) with $\lambda=0.1$. The results can be seen in the following Figure 3.17.

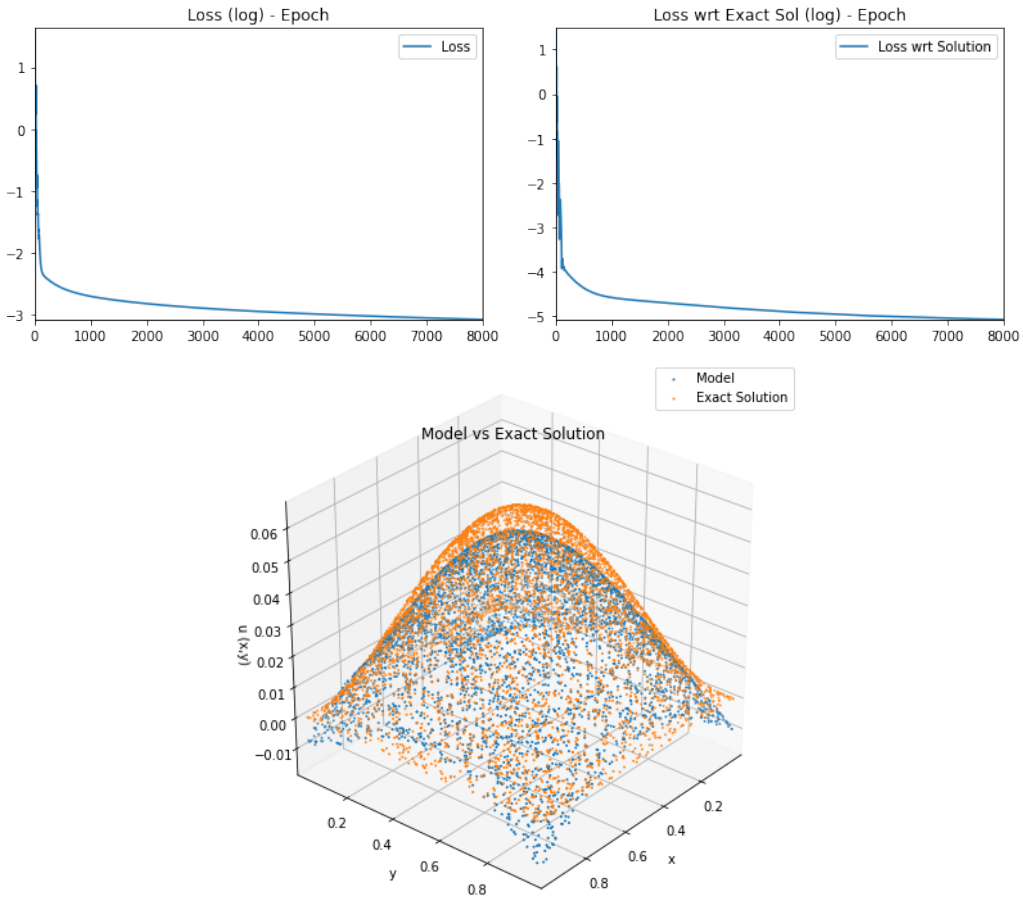

Figure 3.17: Results and performance of the model trained for (3.18).

Chapter 4
Conclusions

Regarding the method, in this work we have covered, theoretically and in practice, all the elements necessary to build a functional artificial neural network method to solve initial/boundary problem. Based on what we have seen as a whole we can conclude that this approach offers a solid option for integrating PDEs. Summarizing all of the pros and cons discussed so far:

- Pros:
  - Mesh-free algorithm: we do not need a mesh, a less costly random collocation of points approach is enough.
  - Higher dimensions of the same operator do not increase much the number of iterations necessary.
  - Good interpolation capacities for points outside of the collocation.
- Cons:
  - Unknown error behaviour: we do not known the precision we are getting and cannot use an objective stop criterion.
  - Unknown stability: we are subject to solving a non-convex optimization, which means that we incur in many potential traps such as local minima and valleys, yielding bad results.
  - Iterative method: we are required to evaluate the derivatives of the artificial neural network at each step of the training, thus the cost of an increase in complexity in the equations by having many different derivatives, accumulates exponentially through the training.

The first and the last cons can be somewhat mitigated. For the last we can try to use synthetic derivatives (i.e. draw the from a distribution or approximation). This idea is not foreign, as we have already seen in this work how the second order quasi-Newton methods (BFGS and L-BFGS) and truncated Newton methods (the Hessian-free), use approximations of the Hessian to avoid computing the whole matrix.

In the case of the error, looking back at the simulations we can observe that in the cases where we took the same operator in different dimensions, like with the divergence operator, the relation between the objective loss L and the exact solution loss L_{sol} behaved in a very similar way. This means that perhaps we could roughly use the results of a smaller problem in 1D to approximate the relation between L and L_{sol}.

To sum up, we would recommend this kind of method to deal with simple differential equations (not many different types of derivatives) in high dimensions and irregular domains. The strengths and weaknesses are close to that of Monte Carlo methods, so with further research it could became an alternative to those methods.

4.1 Author's Final Thoughts

This work contains a wide coverage of the topics general to the subject of deep learning, auto-differentiation, activation functions, optimizers, regularizations, problems like the vanishing and exploding gradient..., all of them applicable to contexts other than this project. The perspective of artificial neural networks through the view of tensors has also been very helpful.

As well as this, there has been an important part of programming involved using TensorFlow, and many bits of practical knowledge were gained through the experience of the many simulations. By experimenting on the cases we were able to identify many problems like the vanishing and exploding gradient with respect to inputs, and offer solutions such as the custom regularization (2.58) proposed in this job, and mechanism (3.4) to balance the gradients.

Finally, from a personal standpoint, I am really satisfied with this work since I have learned many things about differential equations and artificial neural networks, and I believe I have gained much expertise practical to understand and work confidently with those fields.

4.2 Further Work

As contents for further work it would be interesting to study the effect of different architectures and the relations between the objective loss and the solution loss to obtain a measure of the real error. An interesting architecture for PDEs could be the convolutional one, since it is known to preserve account for the spacial relations of its inputs.

Appendix A
Linear Algebra Formulation of 2.3.1

Here we will be describe the elements and derivatives of the example model in Figure 2.6 using linear algebra notation for clarity purposes. We will limit ourselves to tensors in which the linear algebra representation is practical, i.e. up to (1,1)-tensors, or vector and matrices.

First, we will start by writing the variables and parameters (2.11) of the artificial neural network in linear algebra form:

$$
\begin{gathered}
{y^{[\ell]}}_{n_\ell} \equiv \begin{pmatrix} {y^{[1]}}_1 \\ {y^{[1]}}_2 \\ {y^{[3]}}_3 \end{pmatrix}, \quad \begin{pmatrix} {y^{[2]}}_1 \\ {y^{[2]}}_2 \\ {y^{[2]}}_3 \\ {y^{[2]}}_4 \end{pmatrix}, \quad \begin{pmatrix} {y^{[3]}}_1 \\ {y^{[3]}}_2 \end{pmatrix}, \\
{z^{[\ell]}}_{n_\ell} \equiv \begin{pmatrix} {z^{[1]}}_1 \\ {z^{[1]}}_2 \\ {z^{[3]}}_3 \end{pmatrix}, \quad \begin{pmatrix} {z^{[2]}}_1 \\ {z^{[2]}}_2 \\ {z^{[2]}}_3 \\ {z^{[2]}}_4 \end{pmatrix}, \quad \begin{pmatrix} {z^{[3]}}_1 \\ {z^{[3]}}_2 \end{pmatrix}, \\
{w^{[\ell]}}_{n_\ell}^{m_\ell} \equiv \begin{pmatrix} {w^{[1]}}_1^1 & {w^{[1]}}_1^2 \\ {w^{[1]}}_2^1 & {w^{[1]}}_2^2 \\ {w^{[1]}}_3^1 & {w^{[1]}}_3^2 \end{pmatrix}, \quad \begin{pmatrix} {w^{[2]}}_1^1 & {w^{[2]}}_1^2 & {w^{[2]}}_1^3 \\ {w^{[2]}}_2^1 & {w^{[2]}}_2^2 & {w^{[2]}}_2^3 \\ {w^{[2]}}_3^1 & {w^{[2]}}_3^2 & {w^{[2]}}_3^3 \\ {w^{[2]}}_4^1 & {w^{[2]}}_4^2 & {w^{[2]}}_4^3 \end{pmatrix}, \quad \begin{pmatrix} {w^{[3]}}_1^1 & {w^{[3]}}_1^2 & {w^{[3]}}_1^3 & {w^{[3]}}_1^4 \\ {w^{[3]}}_2^1 & {w^{[3]}}_2^2 & {w^{[3]}}_2^3 & {w^{[3]}}_2^4 \end{pmatrix}, \\
{b^{[\ell]}}_{n_\ell} \equiv \begin{pmatrix} {b^{[1]}}_1 \\ {b^{[1]}}_2 \\ {b^{[3]}}_3 \end{pmatrix}, \quad \begin{pmatrix} {b^{[2]}}_1 \\ {b^{[2]}}_2 \\ {b^{[2]}}_3 \\ {b^{[2]}}_4 \end{pmatrix}, \quad \begin{pmatrix} {b^{[3]}}_1 \\ {b^{[3]}}_2 \end{pmatrix},
\end{gathered}
\tag{A.1}
$$

The operations in each neuron, without Einstein Notation are: (2.12):

$$
\begin{aligned}
{z^{[1]}}_{n_1} &= {b^{[1]}}_{n_1} + \sum_{k=1}^{2} {w^{[1]}}_{n_1}^{k} \cdot x_k \,, \qquad {y^{[1]}}_{n_1} = {a^{[1]}}_{n_1}\left({z^{[1]}}_{n_1}\right) \\
{z^{[2]}}_{n_2} &= {b^{[2]}}_{n_2} + \sum_{k=1}^{3} {w^{[2]}}_{n_2}^{k} \cdot {y^{[2]}}_k \,, \qquad {y^{[2]}}_{n_2} = {a^{[2]}}_{n_2}\left({z^{[2]}}_{n_2}\right) \\
{z^{[3]}}_{n_3} &= {b^{[3]}}_{n_3} + \sum_{k=1}^{4} {w^{[3]}}_{n_3}^{k} \cdot {y^{[3]}}_k \,, \qquad u_{n_3} = {y^{[3]}}_{n_3} = {z^{[3]}}_{n_3}
\end{aligned}
\tag{A.2}
$$

The total composed output (2.14) of the artificial neural network, without Einstein Notation is:

$$u_{n_3} = b^{[3]}{}_{n_3} + \sum_{k_1=1}^{4} w^{[3]}{}^{k_1}_{n_3} \cdot a^{[2]}{}_{k_1} \left(b^{[2]}{}_{k_1} + \sum_{k_2=1}^{3} w^{[2]}{}^{k_2}_{k_1} \cdot a^{[1]}{}_{k_2} \left(b^{[1]}{}_{k_2} + \sum_{k_3=1}^{2} w^{[1]}{}^{k_3}_{k_2} \cdot x_{k_3} \right) \right) \tag{A.3}$$

An example of the chain rule would be the following (where underneath we have specified the dimensions of the resulting matrix):

$$\frac{\partial u_{n_3'}}{\partial x_{n_0'}} = \frac{\partial y^3{}_{n_3'}}{\partial z^0{}_{n_0'}} = \underbrace{\frac{\partial y^3{}_{n_3'}}{\partial z^3{}_{n_3}}}_{2\times 2} \cdot \underbrace{\frac{\partial z^3{}_{n_3}}{\partial y^2{}_{n_2}}}_{2\times 4} \cdot \underbrace{\frac{\partial y^2{}_{n_2}}{\partial z^2{}_{n_2}}}_{4\times 4} \cdot \underbrace{\frac{\partial z^2{}_{n_2}}{\partial y^1{}_{n_1}}}_{4\times 3} \cdot \underbrace{\frac{\partial y^1{}_{n_1}}{\partial z^1{}_{n_1}}}_{3\times 3} \cdot \underbrace{\frac{\partial z^1{}_{n_1}}{\partial y^0{}_{n_0}}}_{3\times 2} \cdot \underbrace{\frac{\partial y^0{}_{n_0}}{\partial z^0{}_{n_0'}}}_{2\times 2} \tag{A.4}$$

Finally, the partial derivatives used in the chain rules for (2.15-2.18), in their matrix form look like:

$$\frac{\partial y^{[3]}_{n_3'}}{\partial z^{[3]}_{n_3}} = \begin{pmatrix} \frac{\partial y^{[3]}_1}{\partial z^{[3]}_1} & \frac{\partial y^{[3]}_1}{\partial z^{[3]}_2} \\ \frac{\partial y^{[3]}_2}{\partial z^{[3]}_1} & \frac{\partial y^{[3]}_2}{\partial z^3{}_2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{A.5}$$

$$\frac{\partial y^{[2]}_{n_2}}{\partial z^{[2]}_{n_2}} = \begin{pmatrix} \frac{\partial y^{[2]}_1}{\partial z^{[2]}_1} & \frac{\partial y^{[2]}_1}{\partial z^{[2]}_2} & \frac{\partial y^{[2]}_1}{\partial z^{[2]}_3} & \frac{\partial y^{[2]}_1}{\partial z^{[2]}_4} \\ \frac{\partial y^{[2]}_2}{\partial z^{[2]}_1} & \frac{\partial y^{[2]}_2}{\partial z^{[2]}_2} & \frac{\partial y^{[2]}_2}{\partial z^{[2]}_3} & \frac{\partial y^{[2]}_2}{\partial z^{[2]}_4} \\ \frac{\partial y^{[2]}_3}{\partial z^{[2]}_1} & \frac{\partial y^{[2]}_3}{\partial z^{[2]}_2} & \frac{\partial y^{[2]}_3}{\partial z^{[2]}_3} & \frac{\partial y^{[2]}_3}{\partial z^{[2]}_4} \\ \frac{\partial y^{[2]}_4}{\partial z^{[2]}_1} & \frac{\partial y^{[2]}_4}{\partial z^{[2]}_2} & \frac{\partial y^{[2]}_4}{\partial z^{[2]}_3} & \frac{\partial y^{[2]}_4}{\partial z^{[2]}_4} \end{pmatrix} = \begin{pmatrix} \frac{\partial a^{[2]}_1(z^2{}_1)}{\partial z^2{}_1} & 0 & 0 & 0 \\ 0 & \frac{\partial a^{[2]}_2(z^2{}_2)}{\partial z^2{}_2} & 0 & 0 \\ 0 & 0 & \frac{\partial a^{[2]}_3(z^2{}_3)}{\partial z^2{}_3} & 0 \\ 0 & 0 & 0 & \frac{\partial a^{[2]}_4(z^2{}_4)}{\partial z^2{}_4} \end{pmatrix} \tag{A.6}$$

$$\frac{\partial y^{[1]}_{n_1}}{\partial z^{[1]}_{n_1}} = \begin{pmatrix} \frac{\partial y^{[1]}_1}{\partial z^{[1]}_1} & \frac{\partial y^{[1]}_1}{\partial z^{[1]}_2} & \frac{\partial y^{[1]}_1}{\partial z^{[1]}_3} \\ \frac{\partial y^{[1]}_2}{\partial z^{[1]}_1} & \frac{\partial y^{[1]}_2}{\partial z^{[1]}_2} & \frac{\partial y^{[1]}_2}{\partial z^{[1]}_3} \\ \frac{\partial y^{[1]}_3}{\partial z^{[1]}_1} & \frac{\partial y^{[1]}_3}{\partial z^{[1]}_2} & \frac{\partial y^{[1]}_3}{\partial z^{[1]}_3} \end{pmatrix} = \begin{pmatrix} \frac{\partial a^{[1]}_1(z^{[1]}_1)}{\partial z^{[1]}_1} & 0 & 0 \\ 0 & \frac{\partial a^{[1]}_2(z^{[1]}_2)}{\partial z^{[1]}_2} & 0 \\ 0 & 0 & \frac{\partial a^{[1]}_3(z^{[1]}_3)}{\partial z^{[1]}_3} \end{pmatrix} \tag{A.7}$$

$$\frac{\partial y^{[0]}_{n_0}}{\partial z^{[0]}_{n_0'}} = \begin{pmatrix} \frac{\partial y^{[0]}_1}{\partial z^{[0]}_1} & \frac{\partial y^{[0]}_1}{\partial z^{[0]}_2} \\ \frac{\partial y^0{}_2}{\partial z^{[0]}_1} & \frac{\partial y^{[0]}_2}{\partial z^{[0]}_2} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{A.8}$$

The derivatives of $z^{[\ell]}{}_{n_\ell}$ with respect to $y^{[\ell]}{}_{n_\ell}$ are simply the weight matrices in (A.1).

Appendix B
The Code

As already introduced in section 3.1, the code has been implemented using Python's version TensorFlow 2.3. The code was implemented a Google Colab notebook, hence each class was encapsulated in a cell. Next, there is a brief simplified description on what each cell/class contains:

- **imports Cell:** Imports the main libraries, which includes TensorFlow for tensor manipulation, Time to get the time stamp, Pickle to save the models, MatPlotLib to plot the models, among many others. It also suppresses Warnings.
- **auxiliryPlotting Class:** Encapsulates the methods for plotting results. It contains functions to: plot the distribution of dataset collocations points, plot the output of the model along the exact solution, plot the loss vs epoch graph of the training of the model, plot the errors of individual points in the training set, and plot the loss vs epoch of multiple models in the same graph.
- **myDataSets Class:** Used to create instances of myDataSets. Each of this instances mainly generate for different options, and contain, the collocation of points for the training and validation sets.
- **problemInstance Class:** Encapsulates the methods for the specifics of the diverse instances of the initial/boundary problems. It contains functions that given the training or validation set, and the artificial neural network output and derivatives, return the values of: the differential operator, the external force, the initial/boundary conditions lhs and rhs, and the exact solution of the problem.
- **secondOrderOptimizers Class:** Used to create instances of secondOrderOptimizers implementing the BFGS and L-BFGS optimizers. Since Keras only contains first order optimizers, this custom class uses the implementation in tensorflow_probability library, which is generic, and adapts it to input artificial neural network models.
- **myLayer Class:** Overrides the keras.Layer class and it is used to create object instances of myLayer. These objects contain the parameters and composition of the neurons in an artificial neuron layer, and the *feed* method which process an input to obtain the corresponding layer output.
- **myModel Class:** Overrides the keras.Model class to create instances of myModel, which implements the artificial neural network models. These objects are based on collections of myLayer instances, and contain either, a first order optimizer instance from Keras, or second order optimizer instance from secondOrderOptimizers, which can be accessed and changed at any moment. Through the methods in these objects and given a myDataSets instance one can: obtain the model output, or train the model for a problem set-up which calls on problemInstance for its specifics. Historical information about the loss performance during training is stored in the object. Also, there are methods to save and load models in *.pickle files, for later use.

– **execution Cell:** These are the snippets of code that calls on to the previous classes to perform the experiments. One of these calls usually consist on: a call to instance a myDataSets and myModel, with some options; a call to the fit method in myModel, to train the artificial neural network; a call to one of the auxiliryPlotting methods to plot the results; and optionally, saving the model. B.8 has an example showing in comments all of the variations that can be used.

There are more functionalities implemented throughout these classes. For more details, read the comments through the code. (The code font has been reduced to preserve indentation).

B.1 imports Cell

```
"""
@author: Alberto Garcia Molina
@latest_update: 12/10/2020
"""

import math
from math import log
import numpy as np

import time
import matplotlib.pyplot as plt
from pylab import rcParams
from mpl_toolkits.mplot3d import Axes3D

import pickle
from google.colab import files # Only for the colab environment.

import tensorflow as tf
import tensorflow_probability as tfp
from tensorflow import keras
from tensorflow.keras import layers

import logging, os

# Supress Warnings.

logging.disable(logging.WARNING)
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"

# Optional code to check if there is a GPU available.

#%tensorflow_version 2.x
#device_name = tf.test.gpu_device_name()
#if device_name != '/device:GPU:0':
#  raise SystemError('GPU device not found')
#print('Found GPU at: {}'.format(device_name))
```

B.2 auxiliryPlotting Class

```
"""
@author: Alberto Garcia Molina
@latest_update: 12/10/2020
"""

class auxiliryPlotting:

    ####################
    # Plots the generated sets (Only 2D).
    ####################
    def plot_datasets (data_set):

        %matplotlib inline
        training_set, border_training_set, validation_set = data_set.get_sets()
        _ , _, _, input_dim, output_dim = data_set.get_set_dimensions()

        if (input_dim == 2):
            plt.scatter(training_set[:,0], training_set[:,1], s=0.1)
            plt.title('Training Set')
            plt.show()

            plt.scatter(border_training_set[0][:,0], border_training_set[0][:,1], s=0.1)
            plt.title('Border Training Set')
            plt.show()

            plt.scatter(validation_set[:,0], validation_set[:,1], s=0.1)
            plt.title('Validation Set')
            plt.show()
        else:
            print('Invalid dimensions for plot.')
```

```
31
32      ####################
33      # Plot loss (Only for training set).
34      ####################
35      def plot_loss_function (model,
36                              init_range = 0,
37                              end_range = -1,
38                              subdivide_losses = False,
39                              use_log_scale = False):
40
41          %matplotlib inline
42
43          #Sets the x range of the plot.
44          plot_real_sol_loss = True
45          if (end_range < 0):
46              end_range = len(model._losses)
47
48          # Plots the real loss.
49          min_loss = min(model._losses[init_range:end_range])
50          max_loss = max(model._losses[init_range:end_range])
51
52          if (use_log_scale == True):
53              min_loss = min(loss for loss in model._losses[init_range:end_range] if loss > 0)
54              plt.plot(range(init_range, end_range),
55                       [log(y,10) if y !=0 else None
56                                  for y in model._losses[init_range:end_range]],
57                        label='Loss')
58              plt.ylim(log(min_loss,10), log(max_loss,10))
59              plt.title('Loss (log) - Epoch')
60          else:
61              plt.plot(range(init_range, end_range),
62                       model._losses[init_range:end_range],
63                       label='Loss')
64              plt.ylim(min_loss, max_loss)
65              plt.title('Loss - Epoch')
66
67          plt.xlim(init_range, end_range)
68          plt.legend()
69          plt.show()
70          print('Minimum Loss at:', str(min_loss))
71
72          # Plots the loss wrt the real solution.
73          min_loss_wrt_solution = min(model._losses_solution[init_range:end_range])
74          max_loss_wrt_solution = max(model._losses_solution[init_range:end_range])
75
76          if (use_log_scale == True):
77              min_loss_wrt_solution = min(loss for loss in model._losses_solution[init_range:end_range] if loss > 0)
78              plt.plot(range(init_range, end_range),
79                       [log(y,10) if y !=0 else None
80                                  for y in model._losses_solution[init_range:end_range]],
81                        label='Loss wrt Solution')
82              plt.ylim(log(min_loss_wrt_solution,10), log(max_loss_wrt_solution,10))
83              plt.title('Loss wrt Exact Sol (log) - Epoch')
84          else:
85              plt.plot(range(init_range, end_range),
86                       model._losses_solution[init_range:end_range],
87                       label='Loss')
88              plt.ylim(min_loss_wrt_solution, max_loss_wrt_solution)
89              plt.title('Loss wrt Exact Sol - Epoch')
90          plt.xlim(init_range, end_range)
91          plt.legend()
92          plt.show()
93          print('Minimum Loss wrt Solution at:', str(min_loss_wrt_solution))
94
95          # Plots the subdivision of the loss by its components.
96          if (subdivide_losses == True):
97
98              # Domain Component
99              min_domain_loss = min(model._losses_domain[init_range:end_range])
100             max_domain_loss = max(model._losses_domain[init_range:end_range])
101
102             if (use_log_scale == True):
103                 min_domain_loss = min(loss for loss in model._losses_domain[init_range:end_range] if loss > 0)
104                 plt.plot(range(init_range, end_range),
105                          [log(y,10) if y !=0 else None
106                                     for y in model._losses_domain[init_range:end_range]],
107                           label='Loss')
108                 plt.ylim(log(min_domain_loss,10), log(max_domain_loss,10))
109                 plt.title('Domain Loss (log) - Epoch')
110             else:
111                 plt.plot(range(init_range, end_range),
112                          model._losses_domain[init_range:end_range],
113                          label='Loss')
114                 plt.ylim(min_domain_loss, max_domain_loss)
115                 plt.title('Domain Loss - Epoch')
116
117             plt.xlim(init_range, end_range)
118             plt.legend()
119             plt.show()
120             print('Minimum Domain Loss at:', str(min_domain_loss))
121
122             # Border Component
123             min_border_loss = min(model._losses_border[init_range:end_range])
124             max_border_loss = max(model._losses_border[init_range:end_range])
125
126             if (use_log_scale == True):
127                 min_border_loss = min(loss for loss in model._losses_border[init_range:end_range] if loss > 0)
128                 plt.plot(range(init_range, end_range),
129                          [log(y,10) if y !=0 else None
130                                     for y in model._losses_border[init_range:end_range]],
131                           label='Loss')
132                 plt.ylim(log(min_border_loss,10), log(max_border_loss,10))
```

```
                plt.title('Border Loss (log) - Epoch')
            else:
                plt.plot(range(init_range, end_range),
                         model._losses_border[init_range:end_range],
                         label='Loss')
                plt.ylim(min_border_loss, max_border_loss)
                plt.title('Border Loss - Epoch')

            plt.xlim(init_range, end_range)
            plt.legend()
            plt.show()
            print('Minimum Border Loss at:', str(min_border_loss))

            # Regularization Component
            if (model._regularization != None):
                min_reg_loss = min(model._losses_regularization[init_range:end_range])
                max_reg_loss = max(model._losses_regularization[init_range:end_range])

                if (use_log_scale == True):
                    min_reg_loss = min(loss for loss in model._losses_regularization[init_range:end_range] if loss > 0)
                    plt.plot(range(init_range, end_range),
                            [log(y,10) if y !=0 else None
                                       for y in model._losses_regularization[init_range:end_range]],
                              label='Loss')
                    plt.ylim(log(min_reg_loss,10), log(max_reg_loss,10))
                    plt.title('Regularization Loss (log) - Epoch')
                else:
                    plt.plot(range(init_range, end_range),
                            model._losses_regularization[init_range:end_range],
                            label='Loss')
                    plt.ylim(min_reg_loss, max_reg_loss)
                    plt.title('Regularization Loss - Epoch')
                plt.xlim(init_range, end_range)
                plt.legend()
                plt.show()
                print('Minimum Regularization Loss at:', str(min_border_loss))

    ####################
    # Plots the model (Uses the validation set).
    ####################
    def plot_model (data_set,
                    model,
                    plot_real_sol = False):

        if (model._input_dim == 1):
            if (model._output_dim == 1):
                outputs, _ = model.predict(data_set._validation_set)
                exact_sol = problemInstance.exact_solution(inputs = data_set._validation_set,
                                                           exact_solution = model._exact_solution,
                                                           input_dim = 1,
                                                           output_dim = 1)
                plt.scatter(data_set._validation_set, outputs, s=0.1, label='Model')
                plt.scatter(data_set._validation_set, exact_sol, s=0.1, label='Exact Solution')
                plt.xlabel('x')
                plt.ylabel(r'$\hat{u}(x)$')
                plt.legend()
                plt.title('Model')
            else:
                print('Invalid dimensions for plot.')

        elif (model._input_dim == 2):
            x = data_set._validation_set[:,0]
            y = data_set._validation_set[:,1]

            if (model._output_dim == 1):
                outputs, _ = model.predict(data_set._validation_set)
                exact_sol = problemInstance.exact_solution(inputs = data_set._validation_set,
                                                           exact_solution = model._exact_solution,
                                                           input_dim = 2,
                                                           output_dim = 1)
                # Modificar para los limites reales.
                plt.rcParams['figure.figsize'] = [8,8]
                fig = plt.figure()
                ax = plt.axes(projection='3d')
                ax.set_title('Model')
                ax.set_xlim(tf.math.reduce_min(x), tf.math.reduce_max(x))
                ax.set_ylim(tf.math.reduce_min(y), tf.math.reduce_max(y))
                ax.set_xlabel('x')
                ax.set_ylabel('y')
                ax.set_zlabel('u (x,y)')
                ax.scatter3D(x, y, outputs, cmap='Greens', s=1, label='Model')
                fig = plt.figure()

                ax.scatter3D(x, y, exact_sol, cmap='Greens', s=1, label='Exact Solution')
                fig = plt.figure()
                ax.set_title('Model vs Exact Solution')
                ax.legend()
                #Optional
                ax.view_init(20, 230)

            elif (model._output_dim == 2):
                outputs, _ = model.predict(data_set._validation_set)
                exact_sol = problemInstance.exact_solution(inputs = data_set._validation_set,
                                                           exact_solution = model._exact_solution,
                                                           input_dim = 2,
                                                           output_dim = 2)
                # Modificar para los limites reales.
                plt.rcParams['figure.figsize'] = [7,7]
                fig = plt.figure()
                ax = plt.axes(projection='3d')
                ax.set_title('Model')
                ax.set_xlim(tf.math.reduce_min(x), tf.math.reduce_max(x))
```

```
                ax.set_ylim(tf.math.reduce_min(y), tf.math.reduce_max(y))
                ax.set_xlabel('x')
                ax.set_ylabel('y')
                ax.set_zlabel('u_x(x,y)')
                ax.scatter3D(x, y, outputs[:,0], cmap='Greens', s=0.2)
                fig = plt.figure()

                fig = plt.figure()
                ax = plt.axes(projection='3d')
                ax.set_title('Exact Solution')
                ax.set_xlim(tf.math.reduce_min(x), tf.math.reduce_max(x))
                ax.set_ylim(tf.math.reduce_min(y), tf.math.reduce_max(y))
                ax.set_xlabel('x')
                ax.set_ylabel('y')
                ax.set_zlabel('u_x(x,y)')
                ax.scatter3D(x, y, exact_sol[:,0], cmap='Greens', s=0.2)
                fig = plt.figure()
                plt.legend()

                # Modificar para los limites reales.
                plt.rcParams['figure.figsize'] = [7,7]
                fig = plt.figure()
                ax = plt.axes(projection='3d')
                ax.set_title('Model')
                ax.set_xlim(tf.math.reduce_min(x), tf.math.reduce_max(x))
                ax.set_ylim(tf.math.reduce_min(y), tf.math.reduce_max(y))
                ax.set_xlabel('x')
                ax.set_ylabel('y')
                ax.set_zlabel('u_y(x,y)')
                ax.scatter3D(x, y, outputs[:,1], cmap='Greens', s=0.2)
                fig = plt.figure()

                fig = plt.figure()
                ax = plt.axes(projection='3d')
                ax.set_title('Exact Solution')
                ax.set_xlim(tf.math.reduce_min(x), tf.math.reduce_max(x))
                ax.set_ylim(tf.math.reduce_min(y), tf.math.reduce_max(y))
                ax.set_xlabel('x')
                ax.set_ylabel('y')
                ax.set_zlabel('u_y(x,y)')
                ax.scatter3D(x, y, exact_sol[:,1], cmap='Greens', s=0.2)
                fig = plt.figure()
                plt.legend()

        else:
            print('Invalid dimensions for plot.')

    ####################
    # Plots the squared error (Prototype).
    ####################
    def plot_error (data_set,
                    model):

        if (model._input_dim == 1):
            if (model._output_dim == 1):
                outputs, _ = model.predict(data_set._validation_set)
                exact_sol = problemInstance.exact_solution(inputs = data_set._validation_set,
                                                           exact_solution = model._exact_solution,
                                                           input_dim = 1,
                                                           output_dim = 1)
                plt.scatter(data_set._validation_set, tf.square(outputs-exact_sol), s=0.1, label='Square Error')
                plt.legend()
                plt.title('Model')
            else:
                print('Invalid dimensions for plot.')

        elif (model._input_dim == 2):
            x = data_set._validation_set[:,0]
            y = data_set._validation_set[:,1]

            if (model._output_dim == 1):
                # Loss wrt to the operator and force
                domain_ind_loss = tf.reduce_sum(
                                    tf.square(
                                        problemInstance.differential_operator(
                                            inputs = data_set._training_set,
                                            outputs = model.predict(data_set._training_set,
                                                                    model._required_derivative_order)[0],
                                            outputs_derivatives = model.predict(data_set._training_set,
                                                                                model._required_derivative_order)[1],
                                            differential_operator = model._differential_operator,
                                            input_dim = 2,
                                            output_dim = 1)
                                        - problemInstance.external_force(inputs = data_set._training_set,
                                                                         external_force = model._external_force,
                                                                         input_dim = 2,
                                                                         output_dim = 1)),
                                    axis = 1,
                                    keepdims =True)

                #border_ind_loss = tf.reduce_sum(
                #                    tf.square(data_set._border_training_set[1]
                #                              - model.predict(data_set._border_training_set[0])[0],
                #                              model._required_derivative_order),
                #                    axis = 1,
                #                    keepdims = True)

                # Modificar para los limites reales.
                plt.rcParams['figure.figsize'] = [7,7]
                fig = plt.figure()
                ax = plt.axes(projection='3d')
                ax.set_xlim(0, 1)
```

```
                ax.set_ylim(0, 1)
                ax.set_xlabel('x')
                ax.set_ylabel('y')
                ax.set_zlabel(r'$L_{1}(x;w,b)$')
                ax.scatter3D(#tf.concat([data_set._training_set[:,0], data_set._border_training_set[0][:,0]], axis=0),
                             #tf.concat([data_set._training_set[:,1], data_set._border_training_set[0][:,1]], axis=0),
                             #tf.square(tf.concat([domain_ind_loss, border_ind_loss], axis=0)),
                             tf.concat([data_set._training_set[:,0]], axis=0),
                             tf.concat([data_set._training_set[:,1]], axis=0),
                             tf.square(tf.concat([domain_ind_loss], axis=0)),
                             cmap='Greens',
                             s=0.2)
                fig = plt.figure()
                ax.set_title(r'MSE for the individual domain points: $||\mathcal{L}[\hat{u}(x,y)]-f(x,y)||^{2}_{2}$')
                # Optional
                ax.view_init(30, 40)

                # Loss wrt to the real sol.
                #input_set = tf.concat([data_set._training_set[:], data_set._border_training_set[0][:]], axis=0)
                input_set = tf.concat([data_set._training_set[:]], axis=0)
                outputs, _ = model.predict(input_set)
                exact_sol = problemInstance.exact_solution(inputs = input_set,
                                                           exact_solution = model._exact_solution,
                                                           input_dim = 2,
                                                           output_dim = 1)
                # Modificar para los limites reales.
                plt.rcParams['figure.figsize'] = [7,7]
                fig = plt.figure()
                ax = plt.axes(projection='3d')
                ax.set_title('Model')
                ax.set_xlim(0, 1)
                ax.set_ylim(0, 1)
                ax.set_xlabel('x')
                ax.set_ylabel('y')
                ax.set_zlabel('Square Error')
                ax.scatter3D(input_set[:,0],
                             input_set[:,1],
                             tf.square(outputs-exact_sol),
                             cmap='Greens',
                             s=0.2)
                fig = plt.figure()
                ax.set_title('Square Error of the Real Sol')

            if (model._output_dim == 2):
                # Loss wrt to the operator and force
                domain_ind_loss = tf.reduce_sum(
                                      tf.square(
                                          problemInstance.differential_operator(
                                              inputs = data_set._training_set,
                                              outputs = model.predict(data_set._training_set,
                                                                      model._required_derivative_order)[0],
                                              outputs_derivatives = model.predict(data_set._training_set,
                                                                                  model._required_derivative_order)[1],
                                              differential_operator = model._differential_operator,
                                              input_dim = 2,
                                              output_dim = 2)
                                          - problemInstance.external_force(inputs = data_set._training_set,
                                                                           external_force = model._external_force,
                                                                           input_dim = 2,
                                                                           output_dim = 2)),
                                       axis = 1,
                                       keepdims =True)

                border_ind_loss = tf.reduce_sum(
                                     tf.square(data_set._border_training_set[1]
                                               - model.predict(data_set._border_training_set[0])[0],
                                               model._required_derivative_order),
                                    axis = 1,
                                    keepdims = True)

                # Modificar para los limites reales.
                plt.rcParams['figure.figsize'] = [7,7]
                fig = plt.figure()
                ax = plt.axes(projection='3d')
                ax.set_title('Model')
                ax.set_xlim(0, 1)
                ax.set_ylim(0, 1)
                ax.set_xlabel('x')
                ax.set_ylabel('y')
                ax.set_zlabel('Square Error')
                ax.scatter3D(tf.concat([data_set._training_set[:,0], data_set._border_training_set[0][:,0]], axis=0),
                             tf.concat([data_set._training_set[:,1], data_set._border_training_set[0][:,1]], axis=0),
                             tf.square(tf.concat([domain_ind_loss, border_ind_loss], axis=0)),
                             cmap='Greens',
                             s=0.2)
                fig = plt.figure()
                ax.set_title('Square Error of the Loss Formula')

            else:
                print('Invalid dimensions for plot.')

        def plot_loss_comparison (models,
                                  names,
                                  title,
                                  init_range = 0,
                                  end_range = -1,
                                  subdivide_losses = False,
                                  use_log_scale = False):

            %matplotlib inline
            #rcParams['figure.figsize'] = 15, 5
            rcParams['figure.figsize'] = 20, 4
```

```
        #Sets the x range of the plot.
        if (end_range < 0):
            end_range = 0
            for model in models:
                end_range_var = len(model._losses)
                if (end_range < end_range_var):
                    end_range = end_range_var

        # Plots the global loss.
        min_loss = 1e30
        max_loss = 0
        for model in models:
            min_loss_var = min(loss for loss in model._losses[init_range:end_range] if loss > 0)
            max_loss_var = max(model._losses[init_range:end_range])
            if (min_loss_var < min_loss):
                    min_loss = min_loss_var
            if (max_loss_var > max_loss):
                    max_loss = max_loss_var

        if (use_log_scale == True):
            for model_ind in range(len(models)):
                plt.plot(range(init_range, len(models[model_ind]._losses)),
                        [log(y,10) if y !=0 else None
                                for y in models[model_ind]._losses[init_range:len(models[model_ind]._losses)]],
                        label = names[model_ind])
            plt.ylim(log(min_loss,10), log(max_loss,10))
            plt.title(r'Global Loss Logaritm, $log_{10}(L(w,b))$ vs Epoch - ' + title)
        else:
            for model_ind in range(len(models)):
                plt.plot(range(init_range, len(models[model_ind]._losses)),
                        models[model_ind]._losses[init_range:len(models[model_ind]._losses)],
                        label = names[model_ind])
            plt.ylim(min_loss, max_loss)
            plt.title(r'Global Loss, L(w,b) vs Epoch - ' + title)


        plt.xlim(init_range, end_range)
        plt.xlabel('Iterations')
        plt.ylabel('Loss')
        plt.legend()
        plt.show()

        # Plots the loss wrt the real solution.
        min_loss = 1e30
        max_loss = 0
        for model in models:
            min_loss_var = min(loss for loss in model._losses_solution[init_range:end_range] if loss > 0)
            max_loss_var = max(model._losses_solution[init_range:end_range])
            if (min_loss_var < min_loss):
                    min_loss = min_loss_var
            if (max_loss_var > max_loss):
                    max_loss = max_loss_var

        if (use_log_scale == True):
            for model_ind in range(len(models)):
                plt.plot(range(init_range, len(models[model_ind]._losses_solution)),
                        [log(y,10) if y !=0 else None
                                for y in models[model_ind]._losses_solution[init_range:len(models[model_ind]._losses_solution)
                                        ]],
                        label = names[model_ind])
            plt.ylim(log(min_loss,10), log(max_loss,10))
            plt.title(r'Exact Solution Loss Logaritm, $log_{10}(L_{sol}(w,b))$ vs Epoch - ' + title)
        else:
            for model_ind in range(len(models)):
                plt.plot(range(init_range, len(models[model_ind]._losses_solution)),
                        models[model_ind]._losses_solution[init_range:len(models[model_ind]._losses_solution)],
                        label = names[model_ind])
            plt.ylim(min_loss, max_loss)
            plt.title(r'Exact Solution Loss, $L_{sol}(w,b)$ vs Epoch - ' + title)

        plt.xlim(init_range, end_range)
        plt.xlabel('Iterations')
        plt.ylabel('Loss')
        plt.legend()
        plt.show()

        # Plots the border loss.
        min_loss = 1e30
        max_loss = 0
        for model in models:
            min_loss_var = min(loss for loss in model._losses_border[init_range:end_range] if loss > 0)
            max_loss_var = max(model._losses_border[init_range:end_range])
            if (min_loss_var < min_loss):
                    min_loss = min_loss_var
            if (max_loss_var > max_loss):
                    max_loss = max_loss_var

        if (use_log_scale == True):
            for model_ind in range(len(models)):
                plt.plot(range(init_range, len(models[model_ind]._losses_border)),
                        [log(y,10) if y !=0 else None
                                for y in models[model_ind]._losses_border[init_range:len(models[model_ind]._losses_border)]],
                        label = names[model_ind])
            plt.ylim(log(min_loss,10), log(max_loss,10))
            plt.title(r'Initial Condition Loss Logaritm, $log_{10}(L_{2}(w,b))$ vs Epoch - ' + title)
        else:
            for model_ind in range(len(models)):
                plt.plot(range(init_range, len(models[model_ind]._losses_border)),
                        models[model_ind]._losses_border[init_range:len(models[model_ind]._losses_border)],
                        label = names[model_ind])
            plt.ylim(min_loss, max_loss)
```

```
            plt.title(r'Initial Condition Loss, $L_{2}(w,b)$ vs Epoch - ' + title)

        plt.xlim(init_range, end_range)
        plt.xlabel('Iterations')
        plt.ylabel('Loss')
        plt.legend()
        plt.show()

        # Plots the domain loss.
        min_loss = 1e30
        max_loss = 0
        for model in models:
            min_loss_var = min(loss for loss in model._losses_domain[init_range:end_range] if loss > 0)
            max_loss_var = max(model._losses_domain[init_range:end_range])
            if (min_loss_var < min_loss):
                    min_loss = min_loss_var
            if (max_loss_var > max_loss):
                    max_loss = max_loss_var

        if (use_log_scale == True):
            for model_ind in range(len(models)):
                plt.plot(range(init_range, len(models[model_ind]._losses_domain)),
                         [log(y,10) if y !=0 else None
                                    for y in models[model_ind]._losses_domain[init_range:len(models[model_ind]._losses_domain)]],
                         label = names[model_ind])
            plt.ylim(log(min_loss,10), log(max_loss,10))
            plt.title(r'Domian Loss Logaritm, $log_{10}(L_{1}(w,b))$ vs Epoch - ' + title)
        else:
            for model_ind in range(len(models)):
                plt.plot(range(init_range, len(models[model_ind]._losses_domain)),
                         models[model_ind]._losses_domain[init_range:len(models[model_ind]._losses_domain)],
                         label = names[model_ind])
            plt.ylim(min_loss, max_loss)
            plt.title(r'Domian Loss, $L_{1}(w,b)$ vs Epoch - ' + title)

        plt.xlim(init_range, end_range)
        plt.xlabel('Iterations')
        plt.ylabel('Loss')
        plt.legend()
        plt.show()
```

B.3 myDataSets Class

```
"""
@author: Alberto Garcia Molina
@latest_update: 12/10/2020
"""

class myDataSets:

    # Initialize myDataSets object.
    def __init__ (self,
                  training_batch_size = 2000,
                  border_training_batch_size = 20,
                  validation_batch_size = 1000,
                  input_dim = 1,
                  method = 'uniform-hit-collocation',
                  domain = 'hypercube-0-1',
                  border = 'side-x_1-y_0',
                  seed = None):

        self._training_batch_size = training_batch_size
        self._border_training_batch_size = border_training_batch_size
        self._validation_batch_size = validation_batch_size
        self._input_dim = input_dim

        self.method = method
        self.domain = domain
        self.border = border

        seed_1 = None
        seed_2 = None
        seed_3 = None
        if (seed != None):
            seed_1 = seed
            seed_2 = 2*seed
            seed_3 = 3*seed

        self._training_set = self.generate_domain_set (training_batch_size, input_dim,
                                                       method, domain, seed_1)

        self._border_training_set = self.generate_border_set (border_training_batch_size,
                                                              input_dim, method, border, seed_2)

        self._validation_set = self.generate_domain_set(validation_batch_size, input_dim,
                                                        method, domain, seed_3)

    # Generates a distribution of points inside the solving domain.
    def generate_domain_set (self,
                             batch_size = 2000,
                             input_dim = 1,
                             method = 'uniform-hit-collocation',
                             domain = 'hypercube-0-1',
                             seed = None):

        if (seed != None):
            tf.random.set_seed(seed)
```

```
55
56          if (method == 'uniform-hit-collocation'):
57              if (domain == 'hypercube-0-1'):
58                  domain_set = tf.random.uniform(shape=[batch_size, input_dim],
59                                                 minval=0., maxval=1., dtype=tf.float32)
60              elif (domain == 'quarter-hypercube-0-1'):
61                  domain_set = tf.random.uniform(shape=[batch_size, input_dim],
62                                                 minval=0., maxval=0.5, dtype=tf.float32)
63              elif (domain == 'hypercube-0-2'):
64                  domain_set = tf.random.uniform(shape=[batch_size, input_dim],
65                                                 minval=0., maxval=2., dtype=tf.float32)
66
67          return domain_set
68
69      # Generates a distribution of points on the border of the solving domain.
70      def generate_border_set (self,
71                               batch_size = 2,
72                               input_dim = 1,
73                               method = 'uniform-hit-collocation',
74                               border = 'hypercube-0-1',
75                               seed = None):
76
77          if (seed != None):
78              tf.random.set_seed(seed)
79
80          if (method == 'uniform-hit-collocation'):
81              if (border == 'hypercube-0-1'):
82                  if (input_dim == 1):
83                      x1 = tf.constant(0., shape=[1, input_dim], dtype=tf.float32)
84                      x2 = tf.constant(1., shape=[1, input_dim], dtype=tf.float32)
85                      border_set = tf.concat([x1, x2], axis=0)
86
87                  elif (input_dim == 2):
88                      x1 = tf.random.uniform(shape=[batch_size//4],
89                                             minval=0.,
90                                             maxval=1.,
91                                             dtype=tf.float32)
92                      y1 = tf.constant(0.,
93                                       shape=[batch_size//4],
94                                       dtype=tf.float32)
95                      border_set_1 = tf.stack([x1, y1], axis=1) # y=0
96
97                      x2 = tf.random.uniform(shape=[batch_size//4],
98                                             minval=0.,
99                                             maxval=1.,
100                                            dtype=tf.float32)
101                     y2 = tf.constant(1.,
102                                      shape=[batch_size//4],
103                                      dtype=tf.float32)
104                     border_set_2 = tf.stack([x2, y2], axis=1) # y=1
105
106                     x3 = tf.constant(0.,
107                                      shape=[batch_size//4],
108                                      dtype=tf.float32)
109                     y3 = tf.random.uniform(shape=[batch_size//4],
110                                            minval=0.,
111                                            maxval=1.,
112                                            dtype=tf.float32)
113                     border_set_3 = tf.stack([x3, y3], axis=1) # x=0
114
115                     x4 = tf.constant(1.,
116                                      shape=[batch_size//4],
117                                      dtype=tf.float32)
118                     y4 = tf.random.uniform(shape=[batch_size//4],
119                                            minval=0.,
120                                            maxval=1.,
121                                            dtype=tf.float32)
122                     border_set_4 = tf.stack([x4, y4], axis=1) # x=1
123
124                     border_set = tf.concat([border_set_1, border_set_2, border_set_3, border_set_4],
125                                            axis=0)
126
127             elif (border == 'side-x_1-y_0'):
128                 if (input_dim == 1):
129                     border_set = tf.constant(0., shape=[1, input_dim], dtype=tf.float32)
130                 elif (input_dim == 2):
131                     x1 = tf.random.uniform(shape=[batch_size],
132                                            minval=-1.,
133                                            maxval=2.,
134                                            dtype=tf.float32)
135                     y1 = tf.constant(0.,
136                                      shape=[batch_size],
137                                      dtype=tf.float32)
138                     border_set = tf.stack([x1, y1], axis=1)
139
140             elif (border == 'side-x_1-y_0_expanded'):
141                 if (input_dim == 1):
142                     border_set = tf.constant(0., shape=[1, input_dim], dtype=tf.float32)
143                 elif (input_dim == 2):
144                     x1 = tf.random.uniform(shape=[batch_size],
145                                            minval=-1.,
146                                            maxval=2.,
147                                            dtype=tf.float32)
148                     y1 = tf.constant(0.,
149                                      shape=[batch_size],
150                                      dtype=tf.float32)
151                     border_set = tf.stack([x1, y1], axis=1)
152
153             elif (border == 'two_sides-x_0-y_0'):
154                 if (input_dim == 1):
155                     border_set = tf.constant(0., shape=[1, input_dim], dtype=tf.float32)
156                 elif (input_dim == 2):
```

```
                    x1 = tf.random.uniform(shape=[batch_size//2],
                                           minval=0.,
                                           maxval=1.,
                                           dtype=tf.float32)
                    y1 = tf.constant(0.,
                                     shape=[batch_size//2],
                                     dtype=tf.float32)

                    x2 = tf.constant(0.,
                                     shape=[batch_size//2],
                                     dtype=tf.float32)
                    y2 = tf.random.uniform(shape=[batch_size//2],
                                           minval=0.,
                                           maxval=1.,
                                           dtype=tf.float32)

                    border_set_1 = tf.stack([x1, y1], axis=1) # y=0
                    border_set_2 = tf.stack([x2, y2], axis=1) # x=0
                    border_set = tf.concat([border_set_1, border_set_2],
                                           axis=0)

        return border_set

    # Returns the sets stored in this object.
    def get_sets(self):
        return self._training_set, self._border_training_set, self._validation_set

    # Returns the metadata of the sets stored in this object.
    def get_set_metadata(self):
        return  self._training_batch_size, self._border_training_batch_size, self._validation_batch_size,\
                self._input_dim, self.method, self.domain, self.border

    # Drops the values which have negative loss.
    def drop_negative_loss (data_set,
                            model):

        # TBD
        domain_ind_dif = tf.reduce_sum(
                            problemInstance.differential_operator(
                                inputs = data_set._training_set,
                                outputs = model.predict(data_set._training_set,
                                                        model._required_derivative_order)[0],
                                outputs_derivatives = model.predict(data_set._training_set,
                                                                    model._required_derivative_order)[1],
                                differential_operator = model._differential_operator,
                                input_dim = model._input_dim,
                                output_dim = model._output_dim)
                            - problemInstance.external_force(inputs = data_set._training_set,
                                                             external_force = model._external_force,
                                                             input_dim = model._input_dim,
                                                             output_dim = model._output_dim),
                           axis = 1,
                           keepdims = False)

        # TBD
        border_ind_dif = tf.reduce_sum(
                               data_set._border_training_set[1]
                               - model.predict(data_set._border_training_set[0],0)[0],
                            axis = 1,
                            keepdims = False)

        # Mask
        filtered_training_set = tf.boolean_mask(tensor = data_set._training_set,
                                                mask = domain_ind_dif > 0,
                                                axis = 0)
        filtered_border_training_set_0 = tf.boolean_mask(tensor = data_set._border_training_set[0],
                                                mask = border_ind_dif > 0,
                                                axis = 0)
        filtered_border_training_set_1 = tf.boolean_mask(tensor = data_set._border_training_set[1],
                                                mask = border_ind_dif > 0,
                                                axis = 0)

        # Replace the Dataset
        if (filtered_training_set.shape[0] != 0):
            data_set._training_set = filtered_training_set
            data_set._training_batch_size = filtered_training_set.shape[0]
        if (filtered_border_training_set_0.shape[0] != 0):
            data_set._border_training_set[0] = filtered_border_training_set_0
            data_set._border_training_set[1] = filtered_border_training_set_1
            data_set._border_training_batch_size = filtered_border_training_set_0.shape[0]

    # Drops the values which have negative loss.
    def drop_best_loss (data_set,
                            model):

        # TBD
        domain_ind_dif = tf.reduce_sum(tf.square(
                            problemInstance.differential_operator(
                                inputs = data_set._training_set,
                                outputs = model.predict(data_set._training_set,
                                                        model._required_derivative_order)[0],
                                outputs_derivatives = model.predict(data_set._training_set,
                                                                    model._required_derivative_order)[1],
                                differential_operator = model._differential_operator,
                                input_dim = model._input_dim,
                                output_dim = model._output_dim)
                            - problemInstance.external_force(inputs = data_set._training_set,
                                                             external_force = model._external_force,
                                                             input_dim = model._input_dim,
                                                             output_dim = model._output_dim)),
                           axis = 1,
                           keepdims = False)
```

```
259
260         # TBD
261         border_ind_dif = tf.reduce_sum(tf.square(
262                               data_set._border_training_set[1]
263                               - model.predict(data_set._border_training_set[0],0)[0]),
264                           axis = 1,
265                           keepdims = False)
266
267         # TBD
268         domain_best = tf.math.reduce_mean(
269             domain_ind_dif)
270
271         border_best = tf.math.reduce_mean(
272             border_ind_dif)
273
274         # Mask
275         filtered_training_set = tf.boolean_mask(tensor = data_set._training_set,
276                                                 mask = domain_ind_dif > domain_best,
277                                                 axis = 0)
278         filtered_border_training_set_0 = tf.boolean_mask(tensor = data_set._border_training_set[0],
279                                                 mask = border_ind_dif > border_ind_dif,
280                                                 axis = 0)
281         filtered_border_training_set_1 = tf.boolean_mask(tensor = data_set._border_training_set[1],
282                                                 mask = border_ind_dif > border_ind_dif,
283                                                 axis = 0)
284
285         # Replace the Dataset
286         if (filtered_training_set.shape[0] != 0):
287             data_set._training_set = filtered_training_set
288             data_set._training_batch_size = filtered_training_set.shape[0]
289         if (filtered_border_training_set_0.shape[0] != 0):
290             data_set._border_training_set[0] = filtered_border_training_set_0
291             data_set._border_training_set[1] = filtered_border_training_set_1
292             data_set._border_training_batch_size = filtered_border_training_set_0.shape[0]
```

B.4 problemInstance Class

```
1  """
2  @author: Alberto Garcia Molina
3  @latest_update: 12/10/2020
4  """
5
6
7  class problemInstance():
8
9      ####################
10     # Validates if the problem instance of a model is implemented.
11     ####################
12     def instance_exists (differential_operator,
13                          external_force,
14                          exact_solution):
15
16         if (differential_operator == 'Constant'):
17             required_derivative_order = 0
18         elif (differential_operator == 'Divergence'):
19             required_derivative_order = 1
20         elif (differential_operator == 'Advection'):
21             required_derivative_order = 1
22         elif (differential_operator == 'Laplacian'):
23             required_derivative_order = 2
24         elif (differential_operator == 'Clairaut'):
25             required_derivative_order = 1
26         elif (differential_operator == 'Korteweg-deVries'):
27             required_derivative_order = 3
28         else:
29               raise Exception("Invalid differential operator option. Please recompile with a valid name.")
30
31         if (external_force == 'Force_Constant'
32             or external_force == 'Force_Divergence'
33             or external_force == 'Force_Laplacian'
34             or external_force == 'Force_Advection'
35             or external_force == 'Force_Clairaut'
36             or external_force == 'Force_Korteweg-deVries'):
37               pass
38         else:
39             raise Exception("Invalid external force option. Please recompile with a valid name.")
40
41         if (exact_solution == 'Sol_Polynomial_2Deg_1D_1D'
42             or exact_solution == 'Sol_Polynomial_2Deg_2D_1D'
43             or exact_solution == 'Sol_Polynomial_2Deg_2D_2D'
44             or exact_solution == 'Sol_Polynomial_4Deg_1D_1D'):
45               pass
46         else:
47             raise Exception("Invalid exact solution option. Please recompile with a valid name.")
48
49         return required_derivative_order
50
51     ####################
52     # Computes the differential operator value.
53     ####################
54     #@tf.function(experimental_relax_shapes = True)
55     def differential_operator (inputs,
56                                outputs,
57                                outputs_derivatives,
58                                differential_operator,
59                                input_dim,
60                                output_dim):
```

```
61
62          #batch_size = inputs.shape[0]
63          # tf.print(batch_size)
64
65          if (differential_operator == 'Constant'):
66              if (input_dim == 1):
67                  dif_operator_output = outputs
68              elif (input_dim == 2):
69                  dif_operator_output = outputs
70              else:
71                  raise Exception("Incompatible dimension (Diff Operator).")
72
73          if (differential_operator == 'Divergence'):
74              if (input_dim == 1):
75                  dif_operator_output = outputs_derivatives[0]
76              elif (input_dim == 2):
77                  dif_operator_output = tf.reduce_sum(outputs_derivatives[0], axis=1, keepdims=True)
78              else:
79                  raise Exception("Incompatible dimension (Diff Operator).")
80
81          if (differential_operator == 'Laplacian'):
82              if (input_dim == 2):
83                  dif_operator_output = tf.reduce_sum(outputs_derivatives[1],
84                                                      axis=1,
85                                                      keepdims=True)
86              else:
87                  raise Exception("Incompatible dimension (Diff Operator).")
88
89          if (differential_operator == 'Advection'):
90              if (input_dim == 1):
91                  dif_operator_output = outputs*outputs_derivatives[0]
92              elif (input_dim == 2):
93                  dif_operator_output = tf.linalg.matvec(outputs_derivatives[0], outputs) # transpose_a = True
94                  #outputs = tf.reshape(outputs,[batch_size,1,output_dim])
95                  #outputs_derivatives[0] = tf.transpose(outputs_derivatives[0], perm=[0, 2, 1])
96                  #dif_operator_output = tf.matmul(outputs, outputs_derivatives[0])
97                  #dif_operator_output = tf.reshape(dif_operator_output,
98                  #                                 [batch_size, output_dim])
99              else:
100                 raise Exception("Incompatible dimension (Diff Operator).")
101
102         if (differential_operator == 'Clairaut'):
103             if (input_dim == 2):
104                 dif_operator_output = tf.reduce_sum(inputs*outputs_derivatives[0],
105                                                     axis = 1,
106                                                     keepdims = True)
107             else:
108                 raise Exception("Incompatible dimension (Diff Operator).")
109
110         if (differential_operator == 'Korteweg-deVries'):
111             if (input_dim == 1):
112                 dif_operator_output = outputs_derivatives[2]-outputs_derivatives[0]
113             else:
114                 raise Exception("Incompatible dimension (Diff Operator).")
115
116         return dif_operator_output
117
118     ####################
119     # Computes the external force value.
120     ####################
121     #@tf.function(experimental_relax_shapes = True)
122     def external_force(inputs,
123                        external_force,
124                        input_dim,
125                        output_dim):
126
127         batch_size = inputs.shape[0]
128         # tf.print(batch_size)
129
130         if (external_force == 'Force_Constant'):
131             if (input_dim == 1 and output_dim == 1):
132                 x = inputs
133                 ext_force_output = x*(x-1)
134             else:
135                 raise Exception("Incompatible dimension (External force).")
136
137         if (external_force == 'Force_Divergence'):
138             if (input_dim == 1 and output_dim == 1):
139                 x = inputs
140                 ext_force_output = 2*x-1
141             elif (input_dim == 2 and output_dim == 1):
142                 x = inputs[:,0]
143                 y = inputs[:,1]
144                 ext_force = (2*x-1)*y*(y-1)+x*(x-1)*(2*y-1)
145                 ext_force_output = tf.reshape(ext_force,
146                                               [batch_size, output_dim])
147             else:
148                 raise Exception("Incompatible dimension (External force).")
149
150         if (external_force == 'Force_Laplacian'):
151             if (input_dim == 2 and output_dim == 1):
152                 x = inputs[:,0]
153                 y = inputs[:,1]
154                 ext_force = 2*y*(y-1)+2*x*(x-1)
155                 ext_force_output = tf.reshape(ext_force,
156                                               [batch_size, output_dim])
157             else:
158                 raise Exception("Incompatible dimension (External force).")
159
160         if (external_force == 'Force_Advection'):
161             if (input_dim == 1 and output_dim == 1):
162                 x = inputs
```

```
                ext_force_output = x*(x-1)*(2*x-1)
            elif (input_dim == 2 and output_dim == 2):
                x = inputs[:,0]
                y = inputs[:,1]
                x_ext_force = x*(x-1)*y*(y-1)*((2*x-1)*y*(y-1)+x*(x-1)*(2*y-1))
                y_ext_force = x*(x-1)*y*(y-1)*((2*x-1)*y*(y-1)+x*(x-1)*(2*y-1))
                ext_force_output = tf.stack([x_ext_force, y_ext_force], axis=1)
            else:
                raise Exception("Incompatible dimension (External force).")

        if (external_force == 'Force_Clairaut'):
            if (input_dim == 2 and output_dim == 1):
                x = inputs[:,0]
                y = inputs[:,1]
                ext_force = x*(2*x-1)*y*(y-1)+y*x*(x-1)*(2*y-1)
                ext_force_output = tf.reshape(ext_force,
                                              [batch_size, output_dim])
            else:
                raise Exception("Incompatible dimension (External force).")

        if (external_force == 'Force_Korteweg-deVries'):
            if (input_dim == 1 and output_dim == 1):
                x = inputs
                ext_force_output = -4*x*x*x+3*x*x+32*x-10
            else:
                raise Exception("Incompatible dimension (External force).")

        return ext_force_output

    ####################
    # Computes the rhs border functions of the boundary conditions.
    # Returns list with [border_inputs, rhs_function_1, rhs_funtion_2, ...]
    ####################
    def border_data_prep(border_inputs,
                         border_type,
                         border_batch_size,
                         external_force,
                         required_derivative_order,
                         input_dim,
                         output_dim):

        border_data = None
        if (required_derivative_order != 0):
            border_data = [border_inputs]

            if (required_derivative_order >= 1):

                # Border is always zero in the perimeter.
                if (input_dim == 1):
                    x1 = border_inputs[:,0]

                    if (output_dim == 1):
                        g1 = 0.*x1
                        border_set_outputs = tf.stack([g1], axis=1)

                    elif (output_dim == 2):
                        g1 = 0.*x1
                        g2 = 0.*x1
                        border_set_outputs = tf.stack([g1, g2], axis=1)

                elif (input_dim == 2):
                    x1 = border_inputs[:,0]
                    y1 = border_inputs[:,1]

                    if (output_dim == 1):
                        g1 = 0.*x1*y1
                        border_set_outputs = tf.stack([g1], axis=1)

                    elif (output_dim == 2):
                        g1 = 0.*x1*y1
                        g2 = 0.*x1*y1
                        border_set_outputs = tf.stack([g1, g2], axis=1)

                border_data.append(border_set_outputs)

            if (required_derivative_order >= 2 or external_force == 'Force_Advection'):

                if (external_force == 'Force_Laplacian' or external_force == 'Force_Advection'):
                    if (border_type == 'side-x_1-y_0'
                        or border_type == 'side-x_1-y_0_expanded'):

                        if (input_dim == 1):
                            x1 = border_inputs[:,0]

                            if (output_dim == 1):
                                g1 = 2.*x1-1
                                border_set_outputs = tf.stack([g1], axis=1)

                        elif (input_dim == 2):
                            x1 = border_inputs[:,0]
                            y1 = border_inputs[:,1]

                            if (output_dim == 1):
                                g1 = (2.*y1-1)*x1*(x1-1)
                                border_set_outputs = tf.stack([g1], axis=1)

                    elif (border_type == 'two_sides-x_0-y_0'):
                        if (input_dim == 1):
                            x1 = border_inputs[:,0]

                            if (output_dim == 1):
                                g1 = 2.*x1-1
```

```
                                border_set_outputs = tf.stack([g1], axis=1)

                        elif (input_dim == 2):
                            sub_batch_size = border_batch_size//2
                            x1 = border_inputs[0*sub_batch_size:1*sub_batch_size,0]
                            y1 = border_inputs[0*sub_batch_size:1*sub_batch_size,1]
                            x2 = border_inputs[1*sub_batch_size:2*sub_batch_size,0]
                            y2 = border_inputs[1*sub_batch_size:2*sub_batch_size,1]

                            if (output_dim == 1):
                                g1 = x1*(x1-1)*(2.*y1-1) # First y=0
                                g2 = (2.*x2-1)*y2*(y2-1) # Second x=0
                                border_set_outputs = tf.stack([g1,g2], axis=1)

                    elif (border_type == 'hypercube-0-1'):
                        if (input_dim == 1):
                            x1 = border_inputs[:,0]

                            if (output_dim == 1):
                                g1 = 2.*x1-1
                                border_set_outputs = tf.stack([g1], axis=1)

                        elif (input_dim == 2):
                            sub_batch_size = border_batch_size//4
                            x1 = border_inputs[0*sub_batch_size:1*sub_batch_size,0]
                            y1 = border_inputs[0*sub_batch_size:1*sub_batch_size,1]
                            x2 = border_inputs[1*sub_batch_size:2*sub_batch_size,0]
                            y2 = border_inputs[1*sub_batch_size:2*sub_batch_size,1]
                            x3 = border_inputs[2*sub_batch_size:3*sub_batch_size,0]
                            y3 = border_inputs[2*sub_batch_size:3*sub_batch_size,1]
                            x4 = border_inputs[3*sub_batch_size:4*sub_batch_size,0]
                            y4 = border_inputs[3*sub_batch_size:4*sub_batch_size,1]

                            if (output_dim == 1):
                                g1 = x1*(x1-1)*(2.*y1-1) # First y=0
                                g2 = x2*(x2-1)*(2.*y2-1) # First y=1
                                g3 = (2.*x3-1)*y3*(y3-1) # Second x=0
                                g4 = (2.*x4-1)*y4*(y4-1) # Second x=1

                                border_set_outputs = tf.stack([g1,g2,g3,g4], axis=1)

                border_data.append(border_set_outputs)

        return border_data

    ####################
    # Computes the boundary conditions (lhs).
    ####################
    def lhs_boundary_condtions (inputs,
                                outputs,
                                outputs_derivatives,
                                border_type,
                                border_batch_size,
                                external_force,
                                required_derivative_order,
                                input_dim ,
                                output_dim):

        boundary_cond = []

        if (required_derivative_order >= 1):
            boundary_cond.append(outputs)

        if (required_derivative_order >= 2 or external_force == 'Force_Advection'):
            if (external_force == 'Force_Laplacian' or external_force == 'Force_Advection'):
                if (border_type == 'side-x_1-y_0'
                    or border_type == 'side-x_1-y_0_expanded'):
                      if (input_dim == 1):
                          if (output_dim == 1):
                              boundary_cond.append(outputs_derivatives[0])

                      elif (input_dim == 2):
                          if (output_dim == 1):
                              boundary_cond_val = outputs_derivatives[0][:,1]
                              boundary_cond_val_resized = tf.stack([boundary_cond_val], axis=1)
                              boundary_cond.append(boundary_cond_val_resized)

                elif (border_type == 'two_sides-x_0-y_0'):
                    if (input_dim == 1):
                        if (output_dim == 1):
                            boundary_cond.append(outputs_derivatives[0])

                    elif (input_dim == 2):
                        if (output_dim == 1):
                            sub_batch_size = border_batch_size//2
                            boundary_cond_val_1 = outputs_derivatives[0][0*sub_batch_size:1*sub_batch_size,1] # First y=0
                            boundary_cond_val_2 = outputs_derivatives[0][1*sub_batch_size:2*sub_batch_size,0] # Second x=0
                            boundary_cond_val_comb = tf.stack([boundary_cond_val_1,boundary_cond_val_2], axis=1)
                            boundary_cond.append(boundary_cond_val_comb)

                elif (border_type == 'hypercube-0-1'):
                    if (input_dim == 1):
                        if (output_dim == 1):
                            boundary_cond.append(outputs_derivatives[0])

                    elif (input_dim == 2):
                        if (output_dim == 1):
                            sub_batch_size = border_batch_size//4
                            boundary_cond_val_1 = outputs_derivatives[0][0*sub_batch_size:1*sub_batch_size,1] # y=0
                            boundary_cond_val_2 = outputs_derivatives[0][1*sub_batch_size:2*sub_batch_size,1] # y=1
                            boundary_cond_val_3 = outputs_derivatives[0][2*sub_batch_size:3*sub_batch_size,0] # x=0
                            boundary_cond_val_4 = outputs_derivatives[0][3*sub_batch_size:4*sub_batch_size,0] # x=1
```

```
                            boundary_cond_val_comb = tf.stack([boundary_cond_val_1,boundary_cond_val_2,boundary_cond_val_3,
                                 boundary_cond_val_4], axis=1)
                            boundary_cond.append(boundary_cond_val_comb)

        return boundary_cond

    ####################
    # Computes the exact solution value.
    ####################
    #@tf.function(experimental_relax_shapes = True)
    def exact_solution (inputs,
                        exact_solution,
                        input_dim,
                        output_dim):

        batch_size = inputs.shape[0]
        # tf.print(batch_size)

        if (exact_solution == 'Sol_Polynomial_2Deg_1D_1D'):
            if (input_dim == 1 and output_dim == 1):
                x = inputs
                exact_sol_output = x*(x-1)
            else:
                raise Exception("Incompatible dimension (External force).")

        if (exact_solution == 'Sol_Polynomial_4Deg_1D_1D'):
            if (input_dim == 1 and output_dim == 1):
                x = inputs
                exact_sol_output = x*(x-1)*(x*x-4)
            else:
                raise Exception("Incompatible dimension (External force).")

        if (exact_solution == 'Sol_Polynomial_2Deg_2D_1D'):
            if (input_dim == 2 and output_dim == 1):
                x = inputs[:,0]
                y = inputs[:,1]
                exact_sol = x*(x-1)*y*(y-1)
                exact_sol_output = tf.reshape(exact_sol,
                                              [batch_size, output_dim])
            else:
                raise Exception("Incompatible dimension (External force).")

        if (exact_solution == 'Sol_Polynomial_2Deg_2D_2D'):
            if (input_dim == 2 and output_dim == 2):
                x = inputs[:,0]
                y = inputs[:,1]
                x_exact_sol = x*(x-1)*y*(y-1)
                y_exact_sol = x*(x-1)*y*(y-1)
                exact_sol_output = tf.stack([x_exact_sol, y_exact_sol],
                                            axis=1)
            else:
                raise Exception("Incompatible dimension (External force).")

        return exact_sol_output
```

B.5 secondOrderOptimizers Class

```
"""
@author: Alberto Garcia Molina
@latest_update: 12/10/2020
"""

class secondOrderOptimizers():

    def __init__ (self,
                  name,
                  model):

        self._name = name
        self._model = model

    def optimizer_train_data (self,
                              inputs,
                              border_data,
                              split_gradient = False,
                              display_gradient_norm = False,
                              normalize_gradient = False,
                              train_only_domain = False,
                              train_only_border = False):

        self._inputs = inputs
        self._border_data = border_data
        self._split_gradient = split_gradient
        self._display_gradient_norm = display_gradient_norm
        self._normalize_gradient = normalize_gradient
        self._train_only_domain = train_only_domain
        self._train_only_border = train_only_border

    def flatten_tensor_list (self,
                             tensor_list):

        lineraized_params = tf.constant(0., shape=[1,])
        for param_ind in range(0, len(tensor_list)//2):
            for param_sub_ind in range(0, tensor_list[2*param_ind].shape[0]):
                param = tensor_list[2*param_ind][param_sub_ind]
                lineraized_params = tf.concat([lineraized_params, param], axis=0)
            param = tensor_list[2*param_ind+1]
```

```
            lineraized_params = tf.concat([lineraized_params, param], axis=0)
        lineraized_params = tf.reshape(lineraized_params, [1,-1])
        return lineraized_params

    def update_parameters (self,
                           lineraized_params):

        init_pos = 1
        final_pos = 1
        for param_ind in range(0, len(self._model._trainable_weights)//2):
            for param_sub_ind in range(0, self._model._trainable_weights[2*param_ind].shape[0]):
                final_pos = init_pos + self._model._trainable_weights[2*param_ind][param_sub_ind].shape[0]
                self._model._trainable_weights[2*param_ind][param_sub_ind].assign(lineraized_params[0,init_pos:final_pos])
                init_pos = final_pos
            final_pos = init_pos + self._model._trainable_weights[2*param_ind+1].shape[0]
            self._model._trainable_weights[2*param_ind+1].assign(lineraized_params[0,init_pos:final_pos])
            init_pos = final_pos

    def parametric_model (self,
                          new_parameters):

        self.update_parameters(new_parameters)

        loss, _, _, _ = self._model.loss_function (inputs_domain = self._inputs,
                                                   border_data = self._border_data,
                                                   is_training = False,
                                                   use_only_domain = self._train_only_domain,
                                                   use_only_border = self._train_only_border)

        gradient_update = self._model.back_propagation(inputs = self._inputs,
                                                       border_data = self._border_data,
                                                       is_training = False,
                                                       split_gradient = True,
                                                       display_gradient_norm = False,
                                                       normalize_gradient = self._normalize_gradient,
                                                       train_only_domain = self._train_only_domain,
                                                       train_only_border = self._train_only_border)

        gradient_update = self.flatten_tensor_list(gradient_update)
        loss = tf.reshape(loss, shape=[1])
        return loss, gradient_update

    def apply_gradients (self):

        initial_params = self.flatten_tensor_list(self._model._trainable_weights)
        if (self._name == 'L-BFGS'):
            optimization_results = tfp.optimizer.lbfgs_minimize(self.parametric_model,
                                                                initial_position = initial_params,
                                                                num_correction_pairs = 10,
                                                                tolerance = 1e-32,
                                                                max_iterations=1,
                                                                parallel_iterations=1,
                                                                stopping_condition=None,
                                                                max_line_search_iterations = 10)
        elif (self._name == 'BFGS'):
            optimization_results = tfp.optimizer.bfgs_minimize(self.parametric_model,
                                                               initial_position = initial_params,
                                                               tolerance = 1e-32,
                                                               max_iterations=1,
                                                               parallel_iterations=1,
                                                               stopping_condition=None,
                                                               max_line_search_iterations = 10)
        optimized_params = optimization_results.position
        self.update_parameters(optimized_params)

        _ = self._model.back_propagation(inputs = self._inputs,
                                         border_data = self._border_data,
                                         is_training = True,
                                         split_gradient = self._split_gradient,
                                         display_gradient_norm = self._display_gradient_norm,
                                         normalize_gradient = self._normalize_gradient,
                                         train_only_domain = self._train_only_domain,
                                         train_only_border = self._train_only_border)
```

B.6 myLayer Class

```
"""
@author: Alberto Garcia Molina
@latest_update: 12/10/2020
"""

class myLayer(keras.layers.Layer):

    ####################
    # Initialize the Layer object.
    ####################
    def __init__ (self,
                  name = 'newLayer'):

        super(myLayer, self).__init__()
        self._name = name

    ####################
    # Constructs the Layer
    ####################
    def build (self,
               input_dim = 2,
```

```
                output_dim = 2,
                activation = 'sigmoid',
                weight_initializer = 'xavier',
                bias_initializer = 'xavier',
                seed = None,
                batch_normalization = False,
                supress_bias = False,
                epsilon = 1e-12):

        self._input_dim = input_dim
        self._output_dim = output_dim
        self._activation = activation
        self._weight_initializer = weight_initializer
        self._bias_initializer = bias_initializer
        self._batch_normalization = batch_normalization
        self._has_bias = not supress_bias
        self._epsilon = epsilon

        if (weight_initializer == 'zeros'):
            wInit = tf.keras.initializers.Zeros()
        elif (weight_initializer == 'ones'):
            wInit = tf.keras.initializers.Ones()
        elif (weight_initializer == 'normal_0_1'):
            wInit = RandomNormal(mean=0., stddev=1., seed=seed)
        elif (weight_initializer == 'uniform_-1_1'):
            wInit = tf.keras.initializers.RandomUniform(minval=-1., maxval=1., seed=seed)
        elif (weight_initializer == 'xavier'):
            wInit = tf.keras.initializers.GlorotNormal(seed=seed)
        elif (weight_initializer == 'he'):
            wInit = tf.keras.initializers.he_normal(seed=seed)

        if (bias_initializer == 'zeros'):
            bInit = tf.keras.initializers.Zeros()
        elif (bias_initializer == 'ones'):
            bInit = tf.keras.initializers.Ones()
        elif (bias_initializer == 'normal_0_1'):
            bInit = RandomNormal(mean=0.,stddev=1.,seed=seed)
        elif (bias_initializer == 'uniform_-1_1'):
            bInit = tf.keras.initializers.RandomUniform(minval=-1., maxval=1., seed=seed)
        elif (bias_initializer == 'xavier'):
            bInit = tf.keras.initializers.GlorotNormal(seed=seed)
        elif (bias_initializer == 'he'):
            bInit = tf.keras.initializers.he_normal(seed=seed)

        self.w = self.add_weight(
            name = self._name + ' W',
            shape = (self._input_dim, self._output_dim),
            initializer = wInit,
            trainable = True)
        tf.cast(self.w, tf.float32)

        if (self._has_bias == True):
            self.b = self.add_weight(
                name = self._name + ' b',
                shape = (self._output_dim,),
                initializer = bInit,
                trainable = True)
        else:
            bInit = tf.keras.initializers.Zeros()
            self.b = self.add_weight(
                name = self._name + ' b',
                shape = (self._output_dim,),
                initializer = bInit,
                trainable = True)
        tf.cast(self.b, tf.float32)

    ####################
    # Feeds the input into the layer.
    ####################
    def feed(self,
             inputs = None):

        if (self._has_bias == True):
            outputs = tf.matmul(inputs, self.w) + self.b
        else:
            outputs = tf.matmul(inputs, self.w)

        if (self._activation == 'sigmoid'):
            outputs = tf.nn.sigmoid(outputs)
        if (self._activation == 'tanh'):
            outputs = tf.keras.activations.tanh(outputs)
        if (self._activation == 'relu'):
            outputs = tf.nn.relu(outputs)
        if (self._activation == 'exponential'):
            outputs = tf.keras.activations.exponential(outputs)
        if (self._activation == 'elu'):
            outputs = tf.keras.activations.elu(outputs, alpha=1.0)
        if (self._activation == 'swish'):
            outputs = tf.keras.activations.swish(outputs)
        if (self._activation == 'softplus'):
            outputs = tf.nn.softplus(outputs)


        if (self._batch_normalization == True):
            mean, var = tf.nn.moments(outputs, axes=0, keepdims=True)
            outputs = (outputs-mean)/(tf.math.sqrt(var + self._epsilon))

        return outputs
```

B.7 myModel Class

```
"""
@author: Alberto Garcia Molina
@latest_update: 12/10/2020
"""

class myModel(tf.keras.Model):

    ####################
    # Initializes the model instance.
    ####################
    def __init__ (self,
                  name = 'myModel'):

        super(myModel, self).__init__()

        # Initializes the name and flags for the model.
        self._name = name
        self.built = False
        self._is_compiled = False
        self._has_dataset = False

        # Initializes the historical training variables of the model.
        self._num_epochs_trained = 0
        self._losses = []
        self._losses_domain = []
        self._losses_border = []
        self._losses_regularization = []
        self._losses_solution = []
        self._losses_validation = []

    ####################
    # Builds the layers of the model.
    ####################
    def build (self,
               input_dim = 2,
               hidden_dim = [5,5],
               output_dim = 2,
               activations = 'sigmoid',
               weight_initializers = 'xavier',
               bias_initializers = 'xavier',
               batch_normalization = False,
               supress_bias = False,
               seed = None,
               epsilon = 1e-12):

        # Sets up the basic characteristics of the layers in the model.
        self._input_dim = input_dim
        self._hidden_dim = hidden_dim
        self._output_dim = output_dim
        self._num_hidden_layers = len(hidden_dim)-1
        self._activations = activations
        self._weight_initializers = weight_initializers
        self._bias_initializers = bias_initializers
        self._batch_normalization = batch_normalization
        self._has_bias = not supress_bias

        self._layers = []

        # Constructs the input layer.
        layer = myLayer('Input_Layer')
        layer.build(input_dim = self._input_dim,
                    output_dim = self._hidden_dim[0],
                    activation = self._activations,
                    weight_initializer = self._weight_initializers,
                    bias_initializer = self._bias_initializers,
                    seed = seed,
                    batch_normalization = self._batch_normalization,
                    supress_bias = supress_bias,
                    epsilon = epsilon)
        self._layers.append(layer)
        self._trainable_weights.append(layer.variables[0])
        self._trainable_weights.append(layer.variables[1])

        # Constructs the hidden layers.
        for layer_num in range(1,self._num_hidden_layers+1):
            layer = myLayer('Hidden_Layer_'+str(layer_num))
            layer.build(input_dim = self._hidden_dim[layer_num-1],
                        output_dim = self._hidden_dim[layer_num],
                        activation = self._activations,
                        weight_initializer = self._weight_initializers,
                        bias_initializer = self._bias_initializers,
                        seed = seed,
                        batch_normalization = self._batch_normalization,
                        supress_bias = supress_bias,
                        epsilon = epsilon)
            self._layers.append(layer)
            self._trainable_weights.append(layer.variables[0])
            self._trainable_weights.append(layer.variables[1])

        # Constructs the output layer.
        layer = myLayer('Output_Layer')
        layer.build(input_dim = self._hidden_dim[-1],
                    output_dim = self._output_dim,
                    activation = None,
                    weight_initializer = self._weight_initializers,
                    bias_initializer = self._bias_initializers,
```

```
                    seed = seed,
                    batch_normalization = False,
                    supress_bias = supress_bias,
                    epsilon = None)
        self._layers.append(layer)
        self._trainable_weights.append(layer.variables[0])
        self._trainable_weights.append(layer.variables[1])

        # Raise flag if the neural network has been built successfuly.
        self.built = True

    ####################
    # Sets up the optimizer.
    ####################
    def set_up_optimizer (self,
                          optimizer_selection,
                          learning_rate = 1e-03,
                          epsilon = 1e-07):

        # Sets up the information of the optimizer.
        self._optimizer_selection = optimizer_selection
        self._learning_rate = learning_rate
        self._epsilon = epsilon

        # Adam Optimizer (1st Order)
        if (self._optimizer_selection == 'Adam'):
            self._optimizer1 = tf.keras.optimizers.Adam(learning_rate = self._learning_rate,
                                                        epsilon = self._epsilon,
                                                        amsgrad = False)
            self._optimizer2 = tf.keras.optimizers.Adam(learning_rate = self._learning_rate,
                                                        epsilon = self._epsilon,
                                                        amsgrad = False)
        # AMSGrad Optimizer (1st Order)
        elif (self._optimizer_selection == 'AMSGrad'):
            self._optimizer1 = tf.keras.optimizers.Adam(learning_rate = self._learning_rate,
                                                        epsilon = self._epsilon,
                                                        amsgrad = True)
            self._optimizer2 = tf.keras.optimizers.Adam(learning_rate = self._learning_rate,
                                                        epsilon = self._epsilon,
                                                        amsgrad = True)
        # Nadam Optimizer (1st Order)
        elif (self._optimizer_selection == 'Nadam'):
            self._optimizer1 = tf.keras.optimizers.Nadam(learning_rate = self._learning_rate,
                                                         epsilon = self._epsilon)
            self._optimizer2 = tf.keras.optimizers.Nadam(learning_rate = self._learning_rate,
                                                         epsilon = self._epsilon)
        # AdaGrad Optimizer (1st Order)
        elif (self._optimizer_selection == 'AdaGrad'):
            self._optimizer1 = tf.keras.optimizers.Adagrad(learning_rate = self._learning_rate,
                                                           epsilon = self._epsilon)
            self._optimizer2 = tf.keras.optimizers.Adagrad(learning_rate = self._learning_rate,
                                                           epsilon = self._epsilon)
        # AdaDelta Optimizer (1st Order)
        elif (self._optimizer_selection == 'AdaDelta'):
            self._optimizer1 = tf.keras.optimizers.Adadelta(learning_rate = self._learning_rate,
                                                            rho = 0.95,
                                                            epsilon = self._epsilon)
            self._optimizer2 = tf.keras.optimizers.Adadelta(learning_rate = self._learning_rate,
                                                            rho = 0.95,
                                                            epsilon = self._epsilon)
        # RMSProp Optimizer (1st Order)
        elif (self._optimizer_selection == 'RMSProp'):
            self._optimizer1 = tf.keras.optimizers.RMSprop(learning_rate=self._learning_rate,
                                                           epsilon = self._epsilon)
            self._optimizer2 = tf.keras.optimizers.RMSprop(learning_rate=self._learning_rate,
                                                           epsilon = self._epsilon)
        # Vanilla SDG Optimizer (1st Order)
        elif (self._optimizer_selection == 'Vanilla_SGD'):
            self._optimizer1 = tf.keras.optimizers.SGD(learning_rate = self._learning_rate,
                                                       nesterov = False)
            self._optimizer2 = tf.keras.optimizers.SGD(learning_rate = self._learning_rate,
                                                       nesterov = False)
        # SGD with Momentum Optimizer (1st Order)
        elif (self._optimizer_selection == 'Momentum_SGD'):
            self._optimizer1 = keras.optimizers.SGD(learning_rate = self._learning_rate,
                                                    momentum = 0.9,
                                                    nesterov = False)
            self._optimizer2 = keras.optimizers.SGD(learning_rate = self._learning_rate,
                                                    momentum  = 0.9,
                                                    nesterov = False)
        # SGD with Nesterov Momentum Optimizer (1st Order)
        elif (self._optimizer_selection == 'Nesterov_SGD'):
            self._optimizer1 = keras.optimizers.SGD(learning_rate = self._learning_rate,
                                                    momentum = 0.9,
                                                    nesterov = True)
            self._optimizer2 = keras.optimizers.SGD(learning_rate = self._learning_rate,
                                                    momentum = 0.9,
                                                    nesterov = True)
        # BFGS Optimizer (2st Order)
        elif (self._optimizer_selection == 'BFGS'):
            self._optimizer = secondOrderOptimizers(name = 'BFGS',
                                                    model = self)
        # L-BFGS Optimizer (2st Order)
        elif (self._optimizer_selection == 'L-BFGS'):
            self._optimizer = secondOrderOptimizers(name = 'L-BFGS',
                                                    model = self)
        else:
            self._is_compiled = False
            raise Exception("Invalid optimizer.")

    ####################
    # Builds the problem instance and training set up.
```

```
    ####################
    def compile (self,
                 differential_operator = None,
                 external_force = None,
                 exact_solution = None,
                 optimizer_selection = None,
                 learning_rate = 1e-03,
                 epsilon = 1e-07,
                 scale_factor = 1,
                 loss_fuction = 'square_L2_error',
                 regularization = None,
                 regularization_coef = 0,
                 clip_gradient = 'global'):

        # Sets up he problem solved by the model.
        self._differential_operator = differential_operator
        self._external_force = external_force
        self._exact_solution = exact_solution

        #  Sets up the regularization and loss options.
        self._scale_factor = scale_factor
        self._loss_fuction = loss_fuction
        self._regularization = regularization
        self._regularization_coef = regularization_coef
        self._clip_gradient = clip_gradient

        # Constructs the optimizer and validates the instances.
        if (regularization == None):
            print('No regularization introduced, using default None')
        self._required_derivative_order = problemInstance.\
                                          instance_exists(differential_operator = self._differential_operator,
                                          external_force = self._external_force,
                                          exact_solution = self._exact_solution)
        self.set_up_optimizer(optimizer_selection = optimizer_selection,
                              learning_rate = learning_rate,
                              epsilon = epsilon)

        # Raise flag if the problem and training instance has been built successfuly.
        self._is_compiled = True

    ####################
    # Feed forward of the neural network, returning also the gradient wrt inputs.
    ####################
    def predict (self,
                 inputs,
                 return_derivative_order = 0):

        if (self.built == False):
            raise Exception("Cannot feed forward, the model is not built.")

        outputs_derivatives = []
        if (return_derivative_order in (0,1,2,3)):

            # Output with 0 order derivative.
            if (return_derivative_order == 0):
                outputs = self._layers[0].feed(inputs)
                for layer_ind in range(1, self._num_hidden_layers+2):
                    outputs = self._layers[layer_ind].feed(outputs)
                tf.debugging.check_numerics(outputs, message = 'NaN occurred in network output.')

            # Output with 1st order derivatives.
            if (return_derivative_order == 1):
                with tf.GradientTape(persistent = False) as tape_ord1:
                    tape_ord1.watch(inputs)
                    outputs = self._layers[0].feed(inputs)
                    for layer_ind in range(1, self._num_hidden_layers+2):
                        outputs = self._layers[layer_ind].feed(outputs)
                    # Adaptation for multi-valued functions in one derivative (Burgers Operator).
                if (self._output_dim < 2):
                    outputs_1st_der = tape_ord1.gradient(outputs,
                                                         inputs)
                else:
                    outputs_1st_der = tape_ord1.batch_jacobian(outputs,
                                                               inputs)
                outputs_derivatives.append(outputs_1st_der)
                del tape_ord1
                tf.debugging.check_numerics(outputs,
                                            message = 'NaN occurred in network output.')
                tf.debugging.check_numerics(outputs_1st_der,
                                            message = 'NaN occurred in network 1st derivative output.')

            # Output with 2nd order derivatives.
            elif (return_derivative_order == 2):
                outputs_1st_der = []
                outputs_2nd_der = []
                input_component_list = tf.unstack(inputs, axis = 1)
                for dim in range(inputs.shape[1]):
                    input_component_list[dim] = tf.expand_dims(input_component_list[dim], axis = 1)
                for dim in range(inputs.shape[1]):
                    with tf.GradientTape(persistent = True) as tape_ord2:
                        with tf.GradientTape(persistent = True) as tape_ord1:
                            tape_ord2.watch(input_component_list[dim])
                            tape_ord1.watch(input_component_list[dim])
                            reconst_inputs = tf.squeeze(tf.stack(input_component_list, axis = 1), axis = 2)
                            outputs = self._layers[0].feed(reconst_inputs)
                            for layer_ind in range(1, self._num_hidden_layers+2):
                                outputs = self._layers[layer_ind].feed(outputs)
                        outputs_1st_der_var = tape_ord1.gradient(outputs,
                                                                 input_component_list[dim])
                    outputs_2nd_der_var = tape_ord2.gradient(outputs_1st_der_var,
                                                             input_component_list[dim])
                    outputs_1st_der.append(outputs_1st_der_var)
```

```
                    outputs_2nd_der.append(outputs_2nd_der_var)
                    del tape_ord1
                    del tape_ord2

                outputs_derivatives.append(tf.squeeze(tf.stack(outputs_1st_der, axis = 1), axis = 2))
                outputs_derivatives.append(tf.squeeze(tf.stack(outputs_2nd_der, axis = 1), axis = 2))

                tf.debugging.check_numerics(outputs,
                                            message = 'NaN occurred in network output.')
                tf.debugging.check_numerics(outputs_derivatives[0],
                                            message = 'NaN occurred in network 1st derivative output.')
                tf.debugging.check_numerics(outputs_derivatives[1],
                                            message = 'NaN occurred in network 2nd derivative output.')

            # Output with 3rd order derivatives. (CORRECT FOR THE THIRD ORDER DERIVATIVE RIGHT)
            #elif (return_derivative_order == 3):
            #    with tf.GradientTape(persistent = False) as tape_ord3:
            #        tape_ord3.watch(inputs)
            #        with tf.GradientTape(persistent = False) as tape_ord2:
            #            tape_ord2.watch(inputs)
            #            with tf.GradientTape(persistent = False) as tape_ord1:
            #                tape_ord1.watch(inputs)
            #                outputs = self._layers[0].feed(inputs)
            #                for layer_ind in range(1, self._num_hidden_layers+2):
            #                    outputs = self._layers[layer_ind].feed(outputs)
            #            outputs_1st_der = tape_ord1.gradient(outputs,
            #                                                 inputs)
            #            outputs_derivatives.append(outputs_1st_der)
            #            del tape_ord1
            #        outputs_2nd_der = tape_ord2.gradient(outputs_1st_der,
            #                                             inputs)
            #        outputs_derivatives.append(outputs_2nd_der)
            #        del tape_ord2
            #    outputs_3rd_der = tape_ord3.gradient(outputs_2nd_der,
            #                                         inputs)
            #    outputs_derivatives.append(outputs_3rd_der)
            #    del tape_ord3
            #    tf.debugging.check_numerics(outputs,
            #                                message = 'NaN occurred in network output.')
            #    tf.debugging.check_numerics(outputs_1st_der,
            #                                message = 'NaN occurred in network 1st derivative output.')
            #    tf.debugging.check_numerics(outputs_2nd_der,
            #                                message = 'NaN occurred in network 2nd derivative output.')
            #    tf.debugging.check_numerics(outputs_3rd_der,
            #                                message = 'NaN occurred in network 3rd derivative output.')

        else:
            raise Exception("Invalid order for network derivative computation.")

        return outputs, outputs_derivatives

    ####################
    # Calculates the loss function.
    ####################
    def loss_function (self,
                       inputs_domain,
                       border_data = None,
                       is_training = False,
                       use_only_domain = False,
                       use_only_border = False):

        # Initializes the losses variables.
        loss_domain = tf.constant(0.)
        loss_border = tf.constant(0.)
        loss_regularization = tf.constant(0.)
        loss_solution = tf.constant(0.)

        # Evaluates the left-hand-side and the right-hand-side of the differential equation and solution.

        # Use the gradient balancing regularization.
        if (self._regularization != 'Gradient_Type'):
            outputs, outputs_derivatives = self.predict(inputs = inputs_domain,
                                                        return_derivative_order = self._required_derivative_order)
        else:
            with tf.GradientTape(persistent = True) as tape_reg:
                tape_reg.watch(self._trainable_weights)
                outputs, outputs_derivatives = self.predict(inputs = inputs_domain,
                                                            return_derivative_order = self._required_derivative_order)
            outputs_param_der = tape_reg.gradient(outputs,
                                                  self._trainable_weights,
                                                  unconnected_gradients = tf.UnconnectedGradients.ZERO)
            outputs_derivatives_param_der = tape_reg.gradient(outputs_derivatives[0],
                                                             self._trainable_weights,
                                                             unconnected_gradients = tf.UnconnectedGradients.ZERO)
            for weight_ind in range(self._num_hidden_layers+2):
                loss_regularization += tf.reduce_mean(tf.square(outputs_derivatives_param_der[2*weight_ind]
                                                                - outputs_param_der[2*weight_ind]))
                loss_regularization += tf.reduce_mean(tf.square(outputs_derivatives_param_der[2*weight_ind+1]
                                                                - outputs_param_der[2*weight_ind+1]))
            del tape_reg

        diff_op_output = problemInstance.\
                             differential_operator(inputs = inputs_domain,
                                                   outputs = outputs,
                                                   outputs_derivatives = outputs_derivatives,
                                                   differential_operator = self._differential_operator,
                                                   input_dim = self._input_dim,
                                                   output_dim = self._output_dim)
        ext_force_output = problemInstance.\
                             external_force(inputs = inputs_domain,
                                            external_force = self._external_force,
                                            input_dim = self._input_dim,
```

```
                                        output_dim = self._output_dim)

        exact_sol_output = problemInstance.\
                            exact_solution(inputs = inputs_domain,
                                        exact_solution = self._exact_solution,
                                        input_dim = self._input_dim,
                                        output_dim = self._output_dim)

        # Evaluates the left-hand-side of the border conditions.
        # Right-hand-side already computed in border_data[1:n].
        if (border_data != None):
            outputs_border, outputs_derivatives_border = self.predict(inputs = border_data[0],
                                                                      return_derivative_order = self._required_derivative_order)
            lhs_border = problemInstance.\
                                lhs_boundary_condtions (inputs = border_data[0],
                                                        outputs = outputs_border,
                                                        outputs_derivatives = outputs_derivatives_border,
                                                        border_type = self._border_type,
                                                        border_batch_size = self._border_training_batch_size,
                                                        external_force = self._external_force,
                                                        required_derivative_order = self._required_derivative_order,
                                                        input_dim  = self._input_dim,
                                                        output_dim  = self._output_dim)

        # Computes the loss function for the L2 Error.
        if (self._loss_fuction == 'L2_error'):
            loss_domain = tf.reduce_mean(
                            tf.norm(diff_op_output-ext_force_output,
                                    ord = 'euclidean',
                                    axis = 1))
            if (border_data != None):
                for ind in range(len(border_data)-1):
                    loss_border += tf.reduce_mean(
                                    tf.norm(lhs_border[ind] - border_data[ind+1],
                                            ord='euclidean',
                                            axis=1))
            if (self._regularization == 'Tikhonov'):
                for weight_ind in range(self._num_hidden_layers+2):
                    loss_regularization += tf.reduce_mean(
                                            tf.norm(self._trainable_weights[2*weight_ind],
                                                    ord='euclidean',
                                                    axis = 1))
                    loss_regularization += tf.reduce_mean(
                                              tf.norm(self._trainable_weights[2*weight_ind+1],
                                                      ord='euclidean',
                                                      axis = 0))
            elif (self._regularization == None
                  or self._regularization == 'Gradient_Type'
                  or self._regularization == 'Quadratic_Balance'):
                pass
            else:
                print('Invalid regularization option, defaulting to none.')
                self._regularization = None
            loss_solution = tf.reduce_mean(
                                tf.norm(outputs-exact_sol_output,
                                        ord='euclidean',
                                        axis=1))

        # Computes the loss function for the Square L2 Error (MSE).
        elif (self._loss_fuction == 'square_L2_error'):
            loss_domain = tf.reduce_mean(
                            tf.reduce_sum(tf.square(diff_op_output-ext_force_output),
                                          axis = 1,
                                          keepdims = True))
            if (border_data != None):
                for ind in range(len(border_data)-1):
                    loss_border += tf.reduce_mean(
                                    tf.reduce_sum(tf.square(lhs_border[ind] - border_data[ind+1]),
                                                  axis = 1,
                                                  keepdims = True))
            if (self._regularization == 'Tikhonov'):
                for weight_ind in range(self._num_hidden_layers+2):
                    loss_regularization += tf.reduce_mean(
                                            tf.reduce_mean(tf.square(self._trainable_weights[2*weight_ind]),
                                                           axis = 1,
                                                           keepdims = True))
                    loss_regularization += tf.reduce_mean(
                                            tf.reduce_mean(tf.square(self._trainable_weights[2*weight_ind+1]),
                                                           axis = 0,
                                                           keepdims = True))
            elif (self._regularization == None
                  or self._regularization == 'Gradient_Type'
                  or self._regularization == 'Quadratic_Balance'):
                pass
            else:
                print('Invalid regularization option, defaulting to none.')
                self._regularization = None
            loss_solution = tf.reduce_mean(
                              tf.reduce_sum(tf.square(outputs-exact_sol_output),
                                            axis = 1,
                                            keepdims = True))

        # Computes the loss function for the Absolute Error (L1).
        elif (self._loss_fuction == 'absolute_error'):
            loss_domain = tf.reduce_mean(
                            tf.reduce_sum(tf.abs(diff_op_output-ext_force_output),
                                          axis = 1,
                                          keepdims = True))
            if (border_data != None):
                for ind in range(len(border_data)-1):
                    loss_border += tf.reduce_mean(
                                    tf.reduce_sum(tf.abs(lhs_border[ind] - border_data[ind+1]),
```

```
                                            axis = 1,
                                            keepdims = True))
            if (self._regularization == 'Tikhonov'):
                for weight_ind in range(self._num_hidden_layers+2):
                    loss_regularization += tf.math.reduce_mean(
                                            tf.reduce_mean(tf.abs(self._trainable_weights[2*weight_ind]),
                                                           axis = 1,
                                                           keepdims = True))
                    loss_regularization += tf.math.reduce_mean(
                                            tf.reduce_mean(tf.abs(self._trainable_weights[2*weight_ind+1]),
                                                           axis = 0,
                                                           keepdims = True))
            elif (self._regularization == None
                  or self._regularization == 'Gradient_Type'
                  or self._regularization == 'Quadratic_Balance'):
                pass
            else:
                print('Invalid regularization option, defaulting to none.')
                self._regularization = None
            loss_solution = tf.reduce_mean(
                                tf.reduce_sum(tf.abs(outputs-exact_sol_output),
                                            axis = 1,
                                            keepdims = True))

        # Experimental: Computes the loss (MSE) proportional to the external force.
        elif (self._loss_fuction == 'force_proportional_error'):
            loss_domain = tf.reduce_mean(
                            tf.square((diff_op_output-ext_force_output)
                                      *(ext_force_output+1e-12)))
            if (border_data != None):
                for ind in range(len(border_data)-1):
                    loss_border += tf.reduce_mean(
                                    tf.square(lhs_border[ind] - border_data[ind+1]))
            if (self._regularization == 'Tikhonov'):
                for weight_ind in range(self._num_hidden_layers+2):
                    loss_regularization += tf.reduce_mean(
                                            tf.reduce_mean(tf.square(self._trainable_weights[2*weight_ind]),
                                                           axis = 1,
                                                           keepdims = True))
                    loss_regularization += tf.reduce_mean(
                                            tf.reduce_mean(tf.square(self._trainable_weights[2*weight_ind+1]),
                                                           axis = 0,
                                                           keepdims = True))
            elif (self._regularization == None
                  or self._regularization == 'Gradient_Type'
                  or self._regularization == 'Quadratic_Balance'):
                pass
            else:
                print('Invalid regularization option, defaulting to none.')
                self._regularization = None
            loss_solution = tf.reduce_mean(
                                tf.square(outputs-exact_sol_output))

        # Computes the square of the MSE, i.e. the ||.||^{4}_{2} error .
        elif (self._loss_fuction == 'square_MSE'):
            loss_domain = tf.reduce_mean(
                            tf.reduce_sum(tf.square(tf.square(diff_op_output-ext_force_output)),
                                          axis = 1,
                                          keepdims = True))
            if (border_data != None):
                for ind in range(len(border_data)-1):
                    loss_border += tf.reduce_mean(
                                    tf.reduce_sum(tf.square(tf.square(lhs_border[ind] - border_data[ind+1])),
                                                  axis = 1,
                                                  keepdims = True))
            if (self._regularization == 'Tikhonov'):
                for weight_ind in range(self._num_hidden_layers+2):
                    loss_regularization += tf.reduce_mean(
                                            tf.reduce_mean(tf.square(tf.square(self._trainable_weights[2*weight_ind])),
                                                           axis = 1,
                                                           keepdims = True))
                    loss_regularization += tf.reduce_mean(
                                            tf.reduce_mean(tf.square(tf.square(self._trainable_weights[2*weight_ind+1])),
                                                           axis = 0,
                                                           keepdims = True))
            elif (self._regularization == None
                  or self._regularization == 'Gradient_Type'
                  or self._regularization == 'Quadratic_Balance'):
                pass
            else:
                print('Invalid regularization option, defaulting to none.')
                self._regularization = None
            loss_solution = tf.reduce_mean(
                                tf.reduce_sum(tf.square(outputs-exact_sol_output),
                                            axis = 1,
                                            keepdims = True))

        # Experimental: Computes the loss (MSE) proportional to the square of the inputs.
        elif (self._loss_fuction == 'input_proportional_error'):
            loss_domain = tf.reduce_mean(
                            tf.square((diff_op_output-ext_force_output)
                                      *inputs_domain*inputs_domain))
            if (border_data != None):
                for ind in range(len(border_data)-1):
                    loss_border += tf.reduce_mean(
                                    tf.square(lhs_border[ind] - border_data[ind+1]))
            if (self._regularization == 'Tikhonov'):
                for weight_ind in range(self._num_hidden_layers+2):
                    loss_regularization += tf.reduce_mean(
                                            tf.reduce_mean(tf.square(self._trainable_weights[2*weight_ind]),
                                                           axis = 1,
                                                           keepdims = True))
```

```
                    loss_regularization += tf.reduce_mean(
                                           tf.reduce_mean(tf.square(self._trainable_weights[2*weight_ind+1]),
                                                          axis = 0,
                                                          keepdims = True))
            elif (self._regularization == None
                  or self._regularization == 'Gradient_Type'
                  or self._regularization == 'Quadratic_Balance'):
                pass
            else:
                print('Invalid regularization option, defaulting to none.')
                self._regularization = None
            loss_solution = tf.reduce_mean(
                                tf.square(outputs-exact_sol_output))

        # Computes the loss (MSE) with respect to one component.
        elif (self._loss_fuction == 'square_L2_error_1st_comp'):
            loss_domain = tf.reduce_mean(tf.square(diff_op_output-ext_force_output)[:,0])
            if (border_data != None):
                for ind in range(len(border_data)-1):
                    loss_border += tf.reduce_mean(
                                       tf.reduce_sum(tf.square(lhs_border[ind] - border_data[ind+1]),
                                                     axis = 1,
                                                     keepdims = True))
            if (self._regularization == 'Tikhonov'):
                for weight_ind in range(self._num_hidden_layers+2):
                    loss_regularization += tf.reduce_mean(
                                           tf.reduce_mean(tf.square(self._trainable_weights[2*weight_ind]),
                                                          axis = 1,
                                                          keepdims = True))
                    loss_regularization += tf.reduce_mean(
                                           tf.reduce_mean(tf.square(self._trainable_weights[2*weight_ind+1]),
                                                          axis = 0,
                                                          keepdims = True))
            elif (self._regularization == None
                  or self._regularization == 'Gradient_Type'
                  or self._regularization == 'Quadratic_Balance'):
                pass
            else:
                print('Invalid regularization option, defaulting to none.')
                self._regularization = None
            loss_solution = tf.reduce_mean(tf.square(outputs-exact_sol_output)[0])

        # Error for invalid loss option.
        else:
            raise Exception("Invalid loss option. Please recompile with a valid name.")

        # Experimental: Implement the quadratic loss balance regularization.
        if (self._regularization == 'Quadratic_Balance'):
            if (inputs_border != None and expected_outputs_border != None):
                loss_regularization += tf.sqrt(tf.square(loss_domain-loss_border))

        # Implements the train only domain or border options
        coef_domain = 1
        coef_border = 1
        if (use_only_domain == True):
            coef_border = 0.
        if (use_only_border == True):
            coef_domain = 0.

        # Computes the total loss and checks for explosions.
        loss = coef_domain*loss_domain + coef_border*loss_border
        tf.debugging.check_numerics(loss_domain, message='NaN occurred in domain loss funtion.')
        tf.debugging.check_numerics(loss_border, message='NaN occurred in border loss funtion.')
        tf.debugging.check_numerics(loss_regularization, message='NaN occurred in regularization loss funtion.')
        tf.debugging.check_numerics(loss, message='NaN occurred in total loss funtion.')

        # If the method is set to training mode, the losses are saved on the historical training variables.
        if (is_training == True):
            self._losses_domain.append(loss_domain.numpy())
            self._losses_border.append(loss_border.numpy())
            self._losses_regularization.append(loss_regularization.numpy())
            self._losses.append(loss.numpy())
            self._losses_solution.append(loss_solution.numpy())

        return loss, loss_domain, loss_border, loss_regularization

    ####################
    # Computes the gradient wrt the weights.
    ####################
    def back_propagation (self,
                          inputs,
                          border_data,
                          is_training,
                          split_gradient = False,
                          display_gradient_norm = False,
                          normalize_gradient = False,
                          train_only_domain = False,
                          train_only_border = False):

        # Executes the back propagation
        with tf.GradientTape(persistent = True) as tape_bp:
            tape_bp.watch(self._trainable_weights)
            loss, loss_domain, loss_border,\
            loss_regularization = self.loss_function(inputs_domain = inputs,
                                                     border_data = border_data,
                                                     is_training = is_training,
                                                     use_only_domain = train_only_domain,
                                                     use_only_border = train_only_border)

            total_loss_f = loss+self._regularization_coef*loss_regularization

        gradient_update = tape_bp.gradient(total_loss_f,
```

```
                                        self._trainable_weights,
                                        unconnected_gradients = tf.UnconnectedGradients.ZERO)

        # Splits the gradient wrt to each individual loss component.
        if (split_gradient == True):
            gradient_update_domain = tape_bp.gradient(loss_domain,
                                                      self._trainable_weights,
                                                      unconnected_gradients = tf.UnconnectedGradients.ZERO)
            gradient_update_border = tape_bp.gradient(loss_border,
                                                      self._trainable_weights,
                                                      unconnected_gradients = tf.UnconnectedGradients.ZERO)
            gradient_update_regularization = tape_bp.gradient(loss_regularization,
                                                              self._trainable_weights,
                                                              unconnected_gradients = tf.UnconnectedGradients.ZERO)
        del tape_bp

        # Avoids NAN propagation by rightfully setting them to 0.
        gradient_update = [tf.where(tf.math.is_nan(g), tf.zeros_like(g), g)
                           for g in gradient_update]
        if (split_gradient == True):
            gradient_update_domain = [tf.where(tf.math.is_nan(g), tf.zeros_like(g), g)
                                      for g in gradient_update_domain]
            gradient_update_border = [tf.where(tf.math.is_nan(g), tf.zeros_like(g), g)
                                      for g in gradient_update_border]
            gradient_update_regularization = [tf.where(tf.math.is_nan(g), tf.zeros_like(g), g)
                                      for g in gradient_update_regularization]

        # Applies clipping regularization to bound the gradients.
        if (self._clip_gradient == 'global'):
            gradient_update = tf.clip_by_global_norm(gradient_update, 1e+1)[0]
            gradient_update_domain = tf.clip_by_global_norm(gradient_update_domain, 1e+1)[0]
            gradient_update_border = tf.clip_by_global_norm(gradient_update_border, 1e+1)[0]
            gradient_update_regularization = tf.clip_by_global_norm(gradient_update_regularization, 1e+1)[0]
        elif (self._clip_gradient == 'value'):
            gradient_update = [tf.clip_by_value(g, clip_value_min  = -1e+1, clip_value_max = 1e+1)
                               for g in gradient_update]
            gradient_update_domain = [tf.clip_by_value(g, clip_value_min  = -1e+1, clip_value_max = 1e+1)
                                      for g in gradient_update_domain]
            gradient_update_border = [tf.clip_by_value(g, clip_value_min  = -1e+1, clip_value_max = 1e+1)
                                      for g in gradient_update_border]
            gradient_update_regularization = [tf.clip_by_value(g, clip_value_min  = -1e+1, clip_value_max = 1e+1)
                                              for g in gradient_update_regularization]
        elif (self._clip_gradient == 'norm'):
              gradient_update = [tf.clip_by_norm(g, 1e+1) for g in gradient_update]
              gradient_update_domain = [tf.clip_by_norm(g, 1e+1) for g in gradient_update_domain]
              gradient_update_border = [tf.clip_by_norm(g, 1e+1) for g in gradient_update_border]
              gradient_update_regularization = [tf.clip_by_norm(g, 1e+1) for g in gradient_update_regularization]
        elif (self._clip_gradient == None):
            pass
        else:
            print('Invalid clipping option, defaulting to global.')
            self._clip_gradient = 'global'

        # Applies gradient normalization regularization.
        if (normalize_gradient == True):
            norm = tf.linalg.global_norm(gradient_update)
            gradient_update = [g/norm for g in gradient_update]

        # Rescale Gradient Regularization (Always On)
        if (border_data != None):
            for layer_num in range(self._num_hidden_layers+2):
                weight_norm_domain = tf.norm(gradient_update_domain[2*layer_num],
                                             ord = 'euclidean',
                                             axis = 0)
                bias_norm_domain = tf.norm(gradient_update_domain[2*layer_num+1],
                                           ord = 'euclidean',
                                           axis = 0)
                weight_norm_border = tf.norm(gradient_update_border[2*layer_num],
                                             ord = 'euclidean',
                                             axis = 0)
                bias_norm_border = tf.norm(gradient_update_border[2*layer_num+1],
                                           ord = 'euclidean',
                                           axis = 0)
                weight_norm_regularization = tf.norm(gradient_update_regularization[2*layer_num],
                                                     ord = 'euclidean',
                                                     axis = 0)
                bias_norm_regularization = tf.norm(gradient_update_regularization[2*layer_num+1],
                                                   ord = 'euclidean',
                                                   axis = 0)

                weight_norm = tf.minimum(weight_norm_domain, weight_norm_border)
                bias_norm = tf.minimum(bias_norm_domain, bias_norm_border)

                gradient_update_domain[2*layer_num] = gradient_update_domain[2*layer_num]*weight_norm/(weight_norm_domain+1e-31)
                gradient_update_domain[2*layer_num+1] = gradient_update_domain[2*layer_num+1]*bias_norm/(bias_norm_domain+1e-31)

                gradient_update_border[2*layer_num] = gradient_update_border[2*layer_num]*weight_norm/(weight_norm_border+1e-31)
                gradient_update_border[2*layer_num+1] = gradient_update_border[2*layer_num+1]*bias_norm/(bias_norm_border+1e-31)

                gradient_update_regularization[2*layer_num] = gradient_update_regularization[2*layer_num]*weight_norm/(
                      weight_norm_regularization+1e-31)
                gradient_update_regularization[2*layer_num+1] = gradient_update_regularization[2*layer_num+1]*weight_norm/(
                      bias_norm_regularization+1e-31)

                gradient_update[2*layer_num] = (self._scale_factor*gradient_update_domain[2*layer_num]
                                              + gradient_update_border[2*layer_num]
                                              + self._regularization_coef*gradient_update_regularization[2*layer_num])
                gradient_update[2*layer_num+1] = (self._scale_factor*gradient_update_domain[2*layer_num+1]
                                                + gradient_update_border[2*layer_num+1]
                                                + self._regularization_coef*gradient_update_regularization[2*layer_num+1])

        # Displays the gradient(s) if the option is selected.
```

```
        if (display_gradient_norm == True):
            total_norm = tf.linalg.global_norm(gradient_update)
            domain_norm = tf.linalg.global_norm(gradient_update_domain)
            border_norm = tf.linalg.global_norm(gradient_update_border)
            regularization_norm = tf.linalg.global_norm(gradient_update_regularization)
            print('   Total Gradient Norm', str(total_norm.numpy()))
            print('   Domain Gradient Norm', str(domain_norm.numpy()))
            print('   Border Gradient Norm', str(border_norm.numpy()))
            print('   Regularization Gradient Norm', str(regularization_norm.numpy()))
            for layer_num in range(self._num_hidden_layers+2):
                print('      ', self._layers[layer_num]._name, ' Total Weight Gradient Norm: ',
                      str(tf.norm(gradient_update[2*layer_num],
                                  ord = 'euclidean', axis = 1).numpy()))
                print('      ', self._layers[layer_num]._name, ' Domain Weight Gradient Norm: ',
                      str(tf.norm(gradient_update_domain[2*layer_num],
                                  ord = 'euclidean', axis = 1).numpy()))
                print('      ', self._layers[layer_num]._name, ' Border Weight Gradient Norm: ',
                      str(tf.norm(gradient_update_border[2*layer_num],
                                  ord = 'euclidean', axis = 1).numpy()))
                print('      ', self._layers[layer_num]._name, ' Regularization Weight Gradient Norm: ',
                      str(tf.norm(gradient_update_regularization[2*layer_num],
                                  ord = 'euclidean', axis = 1).numpy()))
                print('      ', self._layers[layer_num]._name, ' Total Bias Gradient Norm: ',
                      str(tf.norm(gradient_update[2*layer_num+1],
                                  ord = 'euclidean', axis = 0).numpy()))
                print('      ', self._layers[layer_num]._name, ' Domain Bias Gradient Norm: ',
                      str(tf.norm(gradient_update_domain[2*layer_num+1],
                                  ord = 'euclidean', axis = 0).numpy()))
                print('      ', self._layers[layer_num]._name, ' Border Bias Gradient Norm: ',
                      str(tf.norm(gradient_update_border[2*layer_num+1],
                                  ord = 'euclidean', axis = 0).numpy()))
                print('      ', self._layers[layer_num]._name, ' Regularization Bias Gradient Norm: ',
                      str(tf.norm(gradient_update_regularization[2*layer_num+1],
                                  ord = 'euclidean', axis = 0).numpy()))

        return gradient_update

    ####################
    # Applies an optimization step.
    ####################
    def apply_training_step (self,
                             inputs,
                             border_data,
                             split_gradient = False,
                             display_gradient_norm = False,
                             normalize_gradient = False,
                             train_only_domain = False,
                             train_only_border = False):

        if (self._optimizer_selection != 'L-BFGS' and self._optimizer_selection != 'BFGS'):
            gradient_update = self.back_propagation(inputs = inputs,
                                                    border_data = border_data,
                                                    is_training = True,
                                                    split_gradient = split_gradient,
                                                    display_gradient_norm = display_gradient_norm,
                                                    normalize_gradient = normalize_gradient,
                                                    train_only_domain = train_only_domain,
                                                    train_only_border = train_only_border)
            if (train_only_domain == True):
                self._optimizer1.apply_gradients(zip(gradient_update, self._trainable_weights))
            elif (train_only_border == True):
                self._optimizer2.apply_gradients(zip(gradient_update, self._trainable_weights))
            else:
                self._optimizer1.apply_gradients(zip(gradient_update, self._trainable_weights))
        else:
            self._optimizer.apply_gradients()

        # Applies clipping regularization to bound the weights.
        for layer_num in range(self._num_hidden_layers+2):
            self._trainable_weights[2*layer_num].assign(tf.clip_by_value(self._trainable_weights[2*layer_num],
                                                                         clip_value_min = -1e+5,
                                                                         clip_value_max = +1e+5))
            self._trainable_weights[2*layer_num+1].assign(tf.clip_by_value(self._trainable_weights[2*layer_num+1],
                                                                           clip_value_min = -1e+5,
                                                                           clip_value_max = +1e+5))

            # Clip by magnitude
            #weight_tensor = self._trainable_weights[2*layer_num]
            #weight_sign = tf.math.sign(self._trainable_weights[2*layer_num])
            #clipped_weight_tensor = tf.clip_by_value(tf.abs(weight_tensor),
            #                                         clip_value_min = 1e-3,
            #                                         clip_value_max = 1e+2)
            #self._trainable_weights[2*layer_num].assign(weight_sign * clipped_weight_tensor)
            #
            #bias_tensor = self._trainable_weights[2*layer_num+1]
            #bias_sign = tf.math.sign(self._trainable_weights[2*layer_num+1])
            #clipped_bias_tensor = tf.clip_by_value(tf.abs(bias_tensor),
            #                                       clip_value_min = 1e-3,
            #                                       clip_value_max = 1e+2)
            #self._trainable_weights[2*layer_num+1].assign(bias_sign * clipped_bias_tensor)

    ####################
    # Loads the training sets to train the network.
    ####################
    def use_training_sets (self,
                           data_set):

        if (data_set == None):
            raise Exception("No data set loaded.")

        training_batch_size, border_training_batch_size, validation_batch_size, \
        input_dim, method, domain, border = data_set.get_set_metadata()
```

```
        if (self._input_dim != input_dim):
            raise Exception("Data set input dimension incompatible with neural network.")

        training_set, border_training_set, validation_set = data_set.get_sets()
        print('Dataset uploaded.')

        return training_batch_size, border_training_batch_size, validation_batch_size,\
               training_set, border_training_set, validation_set,\
               method, domain, border

    ####################
    # Trains the model.
    ####################
    def fit (self,
             data_set = None,
             num_epochs = 1000,
             plot_interval = 500,
             validation_interval = 5000,
             stagnation_stop_tol = 1e-16,
             split_gradient = False,
             display_gradient_norm = False,
             normalize_gradient = False,
             train_only_domain = False,
             train_only_border = False,
             dual_training = False,
             auto_batch_rotation = 0):

        # Loads the data set.
        training_batch_size, border_training_batch_size, validation_batch_size,\
        training_set, border_training_set, validation_set,\
        method, domain_type, border_type = self.use_training_sets(data_set = data_set)

        self._border_type = border_type
        self._border_training_batch_size = border_training_batch_size

        border_data =  problemInstance.\
                            border_data_prep(border_inputs = border_training_set,
                                             border_type = border_type,
                                             border_batch_size = self._border_training_batch_size,
                                             external_force = self._external_force,
                                             required_derivative_order = self._required_derivative_order,
                                             input_dim = self._input_dim,
                                             output_dim = self._output_dim)

        # Loads data in optimizer if BFGS or L-BFGS
        if (self._optimizer_selection == 'L-BFGS' or self._optimizer_selection == 'BFGS'):
            self._optimizer.optimizer_train_data (inputs = training_set,
                                                  border_data = border_data,
                                                  split_gradient = split_gradient,
                                                  display_gradient_norm = display_gradient_norm,
                                                  normalize_gradient = normalize_gradient,
                                                  train_only_domain = train_only_domain,
                                                  train_only_border = train_only_border)

        # First Iteration.
        start_time = time.perf_counter()
        target_total_num_epochs = self._num_epochs_trained + num_epochs - 1
        if (self._num_epochs_trained == 0):
            loss_validation, _, _, _ = self.loss_function (inputs_domain = validation_set,
                                                           border_data = border_data,
                                                           is_training = False,
                                                           use_only_domain = False,
                                                           use_only_border = False)
            self._losses_validation.append(loss_validation.numpy())

            self._num_epochs_trained += 1
            self.apply_training_step (inputs = training_set,
                                      border_data = border_data,
                                      split_gradient = split_gradient,
                                      display_gradient_norm = display_gradient_norm,
                                      normalize_gradient = normalize_gradient,
                                      train_only_domain = train_only_domain,
                                      train_only_border = train_only_border)

            print('Epoch:', str(self._num_epochs_trained), 'Training Loss:', str(self._losses[-1]),
                  'Training Loss wrt Sol:', str(self._losses_solution[-1]))
            print('Domain Loss:', str(self._losses_domain[-1]),
                  'Border Loss:', str(self._losses_border[-1]),
                  'Regularization Loss:', str(self._losses_regularization[-1]))

        # Training Loop
        while (self._num_epochs_trained <= target_total_num_epochs):
            self._num_epochs_trained += 1
            if (dual_training == True):
                if (self._num_epochs_trained % 2 == 0):
                    train_only_domain = True
                    train_only_border = False
                else:
                    train_only_domain = False
                    train_only_border = True

            self.apply_training_step (inputs = training_set,
                                      border_data = border_data,
                                      split_gradient = split_gradient,
                                      display_gradient_norm = display_gradient_norm,
                                      normalize_gradient = normalize_gradient,
                                      train_only_domain = train_only_domain,
                                      train_only_border = train_only_border)

            print('Epoch:', str(self._num_epochs_trained), 'Training Loss:', str(self._losses[-1]),
                  'Training Loss wrt Sol:', str(self._losses_solution[-1]))
            print('Domain Loss:', str(self._losses_domain[-1]),
```

```
                  'Border Loss:', str(self._losses_border[-1]),
                  'Regularization Loss:', str(self._losses_regularization[-1]))

            # Stagnation Stop.
            if (abs(self._losses_domain[-1]-self._losses_domain[-2]) <= stagnation_stop_tol
                and abs(self._losses_border[-1]-self._losses_border[-2]) <= stagnation_stop_tol
                and abs(self._losses_regularization[-1]-self._losses_regularization[-2]) <= stagnation_stop_tol):
                break

            # Early Stop (Validation loss evaluation).
            if (self._num_epochs_trained % validation_interval == 0):
                loss_validation, _, _, _ = self.loss_function (inputs_domain = validation_set,
                                                               border_data = border_data,
                                                               is_training = False,
                                                               use_only_domain = False,
                                                               use_only_border = False)
                self._losses_validation.append(loss_validation.numpy())
                print('Evaluation Loss on Epoch' , str(self._num_epochs_trained), ':',
                      str(self._losses_validation[-1]))
                if (self._losses_validation[-1] > self._losses_validation[-2]):
                    print('Early stop - Loss wrt validation set worsen.')
                    break

                # Autoregulates the regularization coefficient if it only improves the regularization term.
                if (self._losses_domain[-1] > self._losses_domain[-validation_interval]
                    or self._losses_border[-1] > self._losses_border[-validation_interval]):
                        print('Regularization is too strong reducing factor by /5.')
                        self._regularization_coef = self._regularization_coef/5

            # Rotate batches.
            if (auto_batch_rotation != 0):
                new_sample = myDataSets(training_batch_size = training_batch_size,
                                        border_training_batch_size = border_training_batch_size,
                                        validation_batch_size = validation_batch_size,
                                        input_dim = self._input_dim,
                                        method = method,
                                        domain = domain_type,
                                        border = border_type,
                                        seed = None)

                training_batch_size, border_training_batch_size, validation_batch_size,\
                training_set, border_training_set, validation_set,\
                method, domain_type, border_type = self.use_training_sets(data_set = data_set)

            # Plot loss every specified number of epochs while training.

            if (self._num_epochs_trained % plot_interval == 0):
                real_start_epoch = self._num_epochs_trained - num_epochs
                auxiliryPlotting.plot_loss_function(model = self,
                                                    init_range = self._num_epochs_trained-plot_interval+1,
                                                    end_range = -1,
                                                    subdivide_losses = False,
                                                    use_log_scale = False)

        end_time = time.perf_counter()
        print("Execution Time:", end_time - start_time)

#########################33
# A REVISAR
#########################33

    def save(self, filepath=''):

        config_list = []

        config_list.append(self._name)
        config_list.append(self._num_epochs_trained)
        config_list.append(self._losses_from_domain)
        config_list.append(self._losses_from_border)
        config_list.append(self._losses_from_regularization)
        config_list.append(self._losses)
        config_list.append(self._losses_wrt_solution)
        config_list.append(self._losses_wrt_validation)

        config_list.append(self._input_dim)
        config_list.append(self._hidden_dim)
        config_list.append(self._output_dim)
        config_list.append(self._activation)
        config_list.append(self._weight_initializers)
        config_list.append(self._bias_initializers)
        config_list.append(self._batch_normalization)

        config_list.append(self._trainable_weights)

        config_list.append(self._learning_rate)
        config_list.append(self._scale_factor)
        config_list.append(self._regularization_coef)

        config_list.append(self._differential_operator)
        config_list.append(self._external_force)
        config_list.append(self._exact_solution)
        config_list.append(self._loss_fuction)
        config_list.append(self._regularization)
        config_list.append(self._clip_gradient)
        config_list.append(self._optimizer_selection)

        pickle_out = open(filepath + self.name + ".pickle", "wb")
        pickle.dump(config_list, pickle_out)
        pickle_out.close()
        files.download(self.name + ".pickle")

    def load(filepath='', filename=''):
```

```
        pickle_in = open(filepath + filename + ".pickle", "rb")
        config_list = pickle.load(pickle_in)

        ld_name = config_list[0]
        model = myModel(ld_name)
        model._num_epochs_trained = config_list[1]
        model._losses_from_domain = config_list[2]
        model._losses_from_border = config_list[3]
        model._losses_from_regularization = config_list[4]
        model._losses = config_list[5]
        model._losses_wrt_solution = config_list[6]
        model._losses_wrt_validation = config_list[7]

        ld_input_dim = config_list[8]
        ld_hidden_dim = config_list[9]
        ld_output_dim = config_list[10]
        ld_activation = config_list[11]
        ld_weight_initializers = config_list[12]
        ld_bias_initializers = config_list[13]
        ld_batch_normalization = config_list[14]
        model.build(ld_input_dim, ld_hidden_dim, ld_output_dim,
                    ld_activation, ld_weight_initializers, ld_bias_initializers,
                    ld_batch_normalization, None)

        layer_copy = model._layers.copy()
        model._trainable_weights = config_list[15]
        model._layers = layer_copy

        for layer_num in range(0, model._num_hidden_layers+2):
            model._layers[layer_num]._trainable_weights = model._trainable_weights[2*layer_num:2*(layer_num+1)]
            model._layers[layer_num].w = model._layers[layer_num]._trainable_weights[0]
            model._layers[layer_num].b = model._layers[layer_num]._trainable_weights[1]

        ld_learning_rate = config_list[16]
        ld_scale_factor = config_list[17]
        ld_regularization_coef = config_list[18]
        ld_differential_operator = config_list[19]
        ld_external_force = config_list[20]
        ld_exact_solution = config_list[21]
        ld_loss_fuction = config_list[22]
        ld_regularization = config_list[23]
        ld_clip_gradient = config_list[24]
        ld_optimizer_selection = config_list[25]

        model.compile(learning_rate = ld_learning_rate, scale_factor = ld_scale_factor,
                      differential_operator = ld_differential_operator,
                      external_force = ld_external_force,
                      exact_solution = ld_exact_solution, loss_fuction = ld_loss_fuction,
                      regularization = ld_regularization, regularization_coef = ld_regularization_coef,
                      clip_gradient = ld_clip_gradient, optimizer_selection = ld_optimizer_selection)

        return model

    def save_weights(self, filepath=''):

        pickle_out = open(filepath + self.name + "_weights.pickle", "wb")
        pickle.dump(self._trainable_weights, pickle_out)
        pickle_out.close()

    def set_weights(self, filepath='', filename=''):

        pickle_in = open(filepath + filename + ".pickle", "rb")
        self._trainable_weights = pickle.load(pickle_in)
```

B.8 execution Cell

```
"""
@author: Alberto Garcia Molina
@latest_update: 12/10/2020
"""

# Creates the training/validation set.
sample = myDataSets(training_batch_size = 10000,
                    border_training_batch_size = 10000,
                    validation_batch_size = 1000,
                    input_dim = 1,
                    method = 'uniform-hit-collocation',
                    domain = 'hypercube-0-1', #'quarter-hypercube-0-1' #'hypercube-0-2'
                    border = 'hypercube-0-1', #'side-x_1-y_0' #'side-x_1-y_0_expanded' #'two_sides-x_0-y_0'
                    seed = 1993)

#Initializes the model.
model = myModel('model_name')
model.build(input_dim = 1,
            hidden_dim = [3,4,3],
            output_dim = 1,
            activations = 'sigmoid', #'tanh' #'swish' #'exponential' #'softplus' #'relu' #'elu'
            weight_initializers = 'xavier', #'he' #'normal_0_1' #'uniform_-1_1' #'zeros' #'ones'
            bias_initializers = 'xavier',  #'he' #'normal_0_1' #'uniform_-1_1' #'zeros' #'ones'
            batch_normalization = False, # Actually layer normalization.
            supress_bias = False, # Make the neurons have no bias terms.
            seed = 1993,
            epsilon = 1e-12)
model.compile(differential_operator = 'Constant', #'Divergence' #'Advection' #'Laplacian' #'Clairaut'
              external_force = 'Force_Constant', #'Force_Divergence' #'Force_Advection' #'Force_Laplacian' #'Force_Laplacian'
              exact_solution = 'Sol_Polynomial_2Deg_1D_1D', #'Sol_Polynomial_2Deg_1D_1D' #'Sol_Polynomial_2Deg_2D_1D' #'
```

```
                    Sol_Polynomial_2Deg_2D_2D'
              optimizer_selection = 'Adam', #'AMSGrad' #'Nadam' #'AdaGrad' #'AdaDelta' #'RMSProp' #'Vanilla_SGD' #'Momentum_SGD'
                    #'Nesterov_SGD' #'BFGS' #'L-BFGS'
              learning_rate = 1e-02,
              epsilon = 1e-31,
              scale_factor = 1,
              loss_fuction = 'square_L2_error', #'L2_error' #'square_L2_error' #'absolute_error' #'square_MSE' #'
                    square_L2_error_1st_comp'
              regularization = None, #'Gradient_Type' #'Tikhonov'
              regularization_coef = 0.1,
              clip_gradient = 'global' #'value' 'norm'
              )

# Trains the model.
model.fit (data_set = sample,
           num_epochs = 15000,
           validation_interval = 1000,
           plot_interval = 500,
           stagnation_stop_tol = 1e-16,
           split_gradient = True, # Has to always be set to True
           display_gradient_norm = False,
           normalize_gradient = False,
           train_only_domain = False,
           train_only_border = False,
           dual_training = False, # Experimental, do not use. Set to False.
           auto_batch_rotation = False # Automatically generates new collocations with the same set-up as sample every
                plot_interval iterations.
           )

# Optional Plots:
auxiliryPlotting.plot_loss_function(model = model,
                                    init_range = 0,
                                    end_range = -1,
                                    subdivide_losses = True,
                                    use_log_scale = True)

auxiliryPlotting.plot_model(data_set = sample,
                            model = model,
                            plot_real_sol = True)

auxiliryPlotting.plot_error (data_set = sample,
                             model = model)

auxiliryPlotting.plot_loss_comparison (models = [model1, model2, model3],
                                       names = [model1_name, model3_name, model3_name],
                                       title = ['loss of model 1', 'loss of model 2', 'loss of model 3'],
                                       init_range = 0,
                                       end_range = -1,
                                       subdivide_losses = True,
                                       use_log_scale = True)

# save/load options:

model.save(filepath='C//...')
model.load(filepath='C//...')
```

Bibliography

[1] R. L. Bishop and S. I. Goldberg, *Tensor analysis on manifolds*. Dover, Dec. 1980.

[2] C. Reisinger and G. Wittum, "Efficient hierarchical approximation of high-dimensional option pricing problems," *SIAM J. Sci. Comput.*, vol. 29, pp. 440–458, 2007.

[3] L. Grzelak, J. Witteveen, M. Suárez-Taboada, and C. Oosterlee, "The stochastic collocation Monte Carlo sampler: Highly efficient sampling from 'expensive' distributions," *Quantitative Finance*, vol. 19, pp. 339–356, 2019.

[4] J. Sirignano and K. Spiliopoulos, "DGM: A deep learning algorithm for solving partial differential equations," *Journal of Computational Physics*, vol. 375, p. 1339–1364, Dec 2018.

[5] W. Hughes, J. H. Merkin, and R. Sturman, "Analytic solutions of partial differential equations," 2003/4. MATH3414, School of Mathematics, University of Leeds, lectures notes.

[6] S. Kepley and T. Zhang, "A constructive proof of the Cauchy-Kovalevskaya theorem for ordinary differential equations.," *arXiv: Classical Analysis and ODEs*, 2019.

[7] D. G. Gaidashev, "CHAPTER 2 The Cauchy-Kovalevskaya Theorem." `http://www2.math.uu.se/~gaidash/1MA216/CK.pdf`. Accessed: 2020-09-01.

[8] A. Meade and A. Fernandez, "The numerical solution of linear ordinary differential equations by feedforward neural networks," *Mathematical and Computer Modelling*, vol. 19, no. 12, pp. 1 – 25, 1994.

[9] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural ordinary differential equations," *ArXiv*, vol. abs/1806.07366, 2018.

[10] M. Jaderberg, W. Czarnecki, S. Osindero, O. Vinyals, A. Graves, D. Silver, and K. Kavukcuoglu, "Decoupled neural interfaces using synthetic gradients," *ArXiv*, vol. abs/1608.05343, 2017.

[11] W. E, J. Han, and A. Jentzen, "Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations," *Communications in Mathematics and Statistics*, vol. 5, p. 349–380, Nov 2017.

[12] C. Beck, E. Weinan, and A. Jentzen, "Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations," *Journal of Nonlinear Science*, vol. 29, pp. 1563–1619, Aug. 2019.

[13] J. Berg and K. Nyström, "A unified deep artificial neural network approach to partial differential equations in complex geometries," *Neurocomputing*, vol. 317, pp. 28–41, 2018.

[14] T. Dockhorn, "A discussion on solving partial differential equations using neural networks," *ArXiv*, vol. abs/1904.07200, 2019.

[15] S. H. Kolluru, "A neural network based method to solve boundary value problems," *ArXiv*, vol. abs/1909.11082, 2019.

[16] I. Lagaris, A. Likas, and D. Papageorgiou, "Neural-network methods for boundary value problems with irregular boundaries," *IEEE transactions on neural networks*, vol. 11 5, pp. 1041–9, 2000.

[17] I. Lagaris, A. Likas, and D. Fotiadis, "Artificial neural networks for solving ordinary and partial differential equations," *IEEE transactions on neural networks*, vol. 9 5, pp. 987–1000, 1998.

[18] A. Al-Aradi, A. Correia, D. Naiff, G. Jardim, and Y. F. Saporito, "Solving nonlinear and high-dimensional partial differential equations via deep learning," *arXiv: Computational Finance*, 2018.

[19] L. Ruthotto and E. Haber, "Deep neural networks motivated by partial differential equations," *Journal of Mathematical Imaging and Vision*, vol. 62, pp. 352–364, 2019.

[20] P. Chaudhari, A. M. Oberman, S. Osher, S. Soatto, and G. Carlier, "Deep relaxation: partial differential equations for optimizing deep neural networks," *Research in the Mathematical Sciences*, vol. 5, pp. 1–30, 2017.

[21] M. Raissi and G. Karniadakis, "Hidden physics models: Machine learning of nonlinear partial differential equations," *ArXiv*, vol. abs/1708.00588, 2018.

[22] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics informed deep learning (Part I): Data-driven solutions of nonlinear partial differential equations," *ArXiv*, vol. abs/1711.10561, 2017.

[23] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics informed deep learning (Part II): Data-driven discovery of nonlinear partial differential equations," *ArXiv*, vol. abs/1711.10566, 2017.

[24] P. Simard, B. Victorri, Y. LeCun, and J. Denker, "Tangent prop - a formalism for specifying selected invariances in an adaptive network," in *NIPS*, 1991.

[25] W. Czarnecki, S. Osindero, M. Jaderberg, G. Swirszcz, and R. Pascanu, "Sobolev training for neural networks," in *NIPS*, 2017.

[26] R. Grosse, "CSC321 Lecture 10: Automatic Differentiation." `https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/slides/lec10.pdf`. Accessed: 2020-09-01.

[27] A. G. Baydin, B. Pearlmutter, A. Radul, and J. Siskind, "Automatic differentiation in machine learning: a survey," *ArXiv*, vol. abs/1502.05767, 2017.

[28] S. Sun, Z. Cao, H. Zhu, and J. Zhao, "A survey of optimization methods from a machine learning perspective," *IEEE Transactions on Cybernetics*, vol. 50, pp. 3668–3681, 2020.

[29] A. Gaier and D. R. Ha, "Weight agnostic neural networks," in *NeurIPS*, 2019.

[30] S. Ruder, "An overview of gradient descent optimization algorithms," *ArXiv*, vol. abs/1609.04747, 2016.

[31] Y. Li, C. Wei, and T. Ma, "Towards explaining the regularization effect of initial large learning rate in training neural networks," *ArXiv*, vol. abs/1907.04595, 2019.

[32] L. N. Smith and N. Topin, "Super-convergence: Very fast training of neural networks using large learning rates," *ArXiv*, vol. abs/1708.07120, 2017.

[33] P. Chaudhari, A. Choromanska, S. Soatto, Y. LeCun, C. Baldassi, C. Borgs, J. Chayes, L. Sagun, and R. Zecchina, "Entropy-SGD: Biasing gradient descent into wide valleys," *ArXiv*, vol. abs/1611.01838, 2016.

[34] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning* (S. Dasgupta and D. McAllester, eds.), vol. 28 of *Proceedings of Machine Learning Research*, (Atlanta, Georgia, USA), pp. 1139–1147, PMLR, 17–19 Jun 2013.

[35] J. C. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, 2011.

[36] M. D. Zeiler, "Adadelta: An adaptive learning rate method," *ArXiv*, vol. abs/1212.5701, 2012.

[37] G. Hinton, "Lecture 6a. Overview of mini-batch gradient descent." `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Accessed: 2020-09-01.

[38] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. cite arxiv:1412.6980 Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[39] S. J. Reddi, S. Kale, and S. Kumar, "On the convergence of Adam and beyond.," in *ICLR*, OpenReview.net, 2018.

[40] T. Dozat, "Incorporating nesterov momentum into Adam," 2016.

[41] J. Zhang, I. Mitliagkas, and C. Ré, "Yellowfin and the art of momentum tuning," *ArXiv*, vol. abs/1706.03471, 2019.

[42] J. Rafati and R. F. Marcia, "Quasi-Newton optimization methods for deep learning applications.," *CoRR*, vol. abs/1909.01994, 2019.

[43] J. Martínez, "Practical quasi-Newton methods for solving nonlinear systems," *Journal of Computational and Applied Mathematics*, vol. 124, pp. 97–121, 2000.

[44] J. Nocedal and S. J. Wright, "Numerical optimization (springer series in operations research and financial engineering)," 2000.

[45] R. Bollapragada, D. Mudigere, J. Nocedal, H.-J. M. Shi, and P. Tang, "A progressive batching L-BFGS method for machine learning," in *ICML*, 2018.

[46] J. Martens, "Deep learning via hessian-free optimization.," in *ICML* (J. Fürnkranz and T. Joachims, eds.), pp. 735–742, Omnipress, 2010.

[47] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *ArXiv*, vol. abs/1710.05941, 2018.

[48] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural networks : the official journal of the International Neural Network Society*, vol. 107, pp. 3–11, 2018.

[49] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (J. Fürnkranz and T. Joachims, eds.), pp. 807–814, 2010.

[50] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," vol. 9 of *Proceedings of Machine Learning Research*, (Chia Laguna Resort, Sardinia, Italy), pp. 249–256, JMLR Workshop and Conference Proceedings, 13–15 May 2010.

[51] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[52] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[53] J. Ba, J. Kiros, and G. E. Hinton, "Layer normalization," *ArXiv*, vol. abs/1607.06450, 2016.

[54] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *NIPS*, 2006.

[55] A. Andoni, R. Panigrahy, G. Valiant, and L. Zhang, "Learning polynomials with neural networks," in *ICML*, 2014.

[56] Z.-Q. J. Xu, "A note of using Tensorflow to code Laplacian operator in high dimension." `https://ins.sjtu.edu.cn/people/xuzhiqin/pub/laplaciancode.pdf`. Accessed: 2020-09-01.