Abstract

A couple of decades ago, I. Lagaris and A. Likas presented a new numerical method making use of feedforward neural networks to solve differential equations. This method can be used to solve initial and boundary value problems and has shown to provide accurate solution when used with single hidden layered net of fixed architecture. However, not many results have been presented on the choice of the hyper-parameters of the network and how it influences the computed solution. This thesis aims at answering those questions by numerical study of this method. The first investigation looks at how the error of the numerical solution decreases as the number of points used to train the network increases. Then, some similar tests are performed for the complexity of the network and its depth. Lastly, it will be shown that the choice of activation function does not matter as long as the function is regular enough and non-linear.

Keywords: Neural Network, Machine learning, Differential Equations, Numerical Solutions, Parameter sensitivity.

Contents

Chapter 1

Background

1.1 Artificial Neural Networks

Artificial Neural Networks (ANN) are computing systems first inspired by the biological neural network in animals' brain, having the initial goal of solving problems in a way similar to human reasoning. Further understanding of the biological neural system has since shown that ANN's have been created based on a wrong brain model, but they have proved to be very interesting and efficient for many problems such as in machine vision, financial sciences, pattern recognition in DNA, weather forecasting or to quickly compute a complex but well known function. Thus this field of research keeps attracting a lot of attention from researchers and industrials. Even though some people expect the ANN to work well only to solve problems that the human brain is capable to solve efficiently, mathematicians started to find interest in them as they can bring some new interesting features when looking for solution of differential equations. In particular they can be really useful to get a grasp of a non-visual situation in real time. For instance considering a mathematical model of in vivo system (internal system during surgery for example), having a neural network mapping the state of the system in real time let the surgeon visualize or even predict the effects of the surgery in a time that regular method such as finite volumes/elements/differences could not achieve. Indeed, they show excellent interpolation capabilities as well as decent extrapolation accuracy. Moreover, ANN's are by construction, very suited for parallel implementation, and then the solving can be accelerated using parallel computing.

1.1.1 Description of an artificial neural network

ANN are described by a collection of units, or neurons to keep the analogy with biological systems, which computes the "information" given by the previous units or by the input and transmits the information about its own state to the following units or to the network output, giving a prediction associated with the input.

Mathematically, a neural network is a mapping $N : \mathbb{R}^n \to \mathbb{R}^r$ from a class of function in which each element is described by a vector p of parameters. The function N can be seen as a composition of functions, the number of compositions defining the depth of the network. Those are vectorial mapping $g_i : \mathbb{R}^{l_i} \to \mathbb{R}^{l_{i+1}}$ where the largest $l_i \in \mathbb{N}$ define the width of the network.

Outline of a network

A simple neural network is basically composed of three parts:

- an input layer to which the input is given. No computation is made at this point, thus this layer is sometimes not mentioned as part of the network;
- one or more hidden layers. Those are the layers in which most of the computation is made. The output of the units in those layers can as well be connected to the next layer's units, to other (hidden, input or output) layers or to itself. If each unit in a layer is connected to every unit in the previous, the layer is said fully connected, as opposed to sparsely connected layers;
- an output layer which can have one or more units depending on the problem.

This can be represented as in Figure 1.1, where the outline of a single output feedforward neural network with two hidden layers and four inputs units is described. This network's depth is $K = 2$ and the width is $H = 5$. This type of neural network where the output of every unit is only broadcast to the next layer is called Feed-Forward Neural Networks (FFNN) and for the rest of this work, only feedforward, fully connected network will be considered. For this particular kind of ANN, each unit is connected to the next layer by a set of weights and a linear combination of the previous layer's units is used to compute the output of every unit. The output of the network is always linear.

Units

For an input $x \in \mathbb{R}^n$, consider h_j^i the unit j in the hidden layer i.
The unit h_j^i, which is given the inputs $z^i(x) \in \mathbb{R}^{l_i}$, from previous layers, is characterized by the following components:

- an output $a_j^i(z^i)$, linear in the method covered in this paper;
- a function σ, called activation function;
- a set of connections to the units in the next layer.

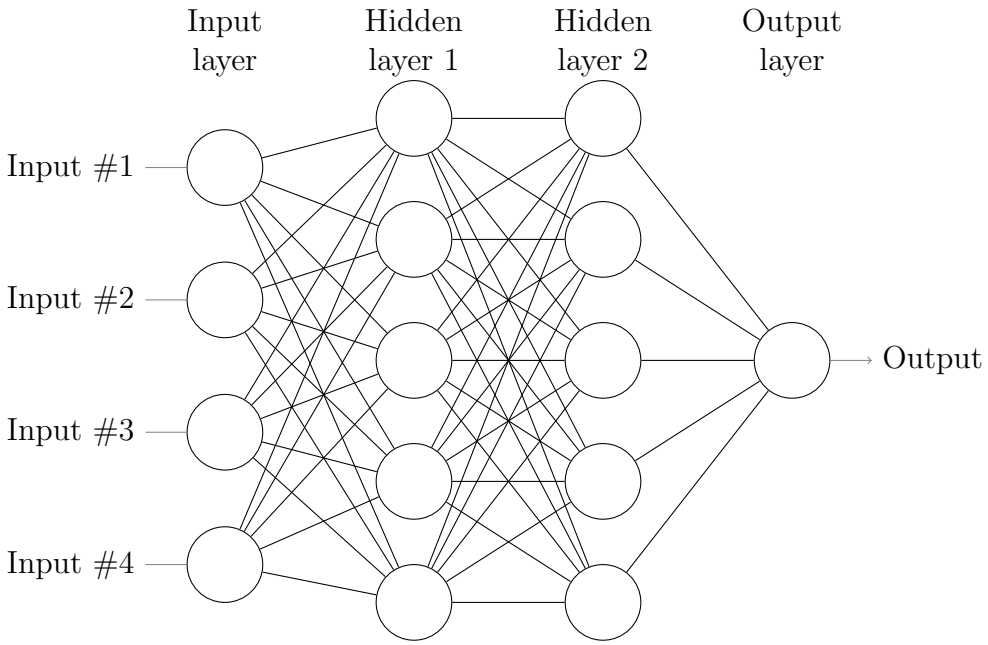


Figure 1.1: Outline of a fully connected multilayer perceptron.

From the analogy with the biological systems, the units get the possibility to be either activated or not, that is to say have an output $a_j^i \in \{0, 1\}$. It is the purpose of the activation function to determine this state. If the neuron is not activated, then it's output should not be broadcast any farther in the network. However, the linear output of a unit is not bounded, and therefore the output actually broadcast to the next layer is given by $\sigma(a_j(z_j))$.

Activation function

To emulate the state of the neurons, i.e. activated or not, one must define a threshold function which decide if the neuron should be considered activated or not. The use of an activation function allows to control the unit's output so that it stays bounded in the interval $[0, 1]$. A simple case would be to use the Heaviside function

$$\theta(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases},$$

or similar step function, but this class of functions could cause a loss of information, making the predictions of the network accurate for only one kind of problem and input. One would also encounter problems during the training of the network as derivatives of the network are usually needed for this part, as will be discussed later. A wiser choice would be to use continuous smooth approximations of these θ functions, like sigmoid function. Sigmoid functions get their name from their character-

istic S-shaped graph and are often used in neural network. For instance, one of the most widely used activation function is the hyperbolic tangent:

$$\tanh(x) = \frac{e^{2x} + 1}{e^{2x} - 1}.$$

Often, the term sigmoid function is used to refer to one frequently used activation function called logistic function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

There are lot of different activation functions with different properties that can make them more suited for certain kind of problems and networks. Another interesting function is the 'Inverse Square Root Unit (ISRU)':

$$f(x) = \frac{x}{\sqrt{1 + \alpha x^2}}.$$

One can cite as well the tangent function tan and its inverse arctan, the 'Rectified Linear Unit (ReLu)':

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases},$$

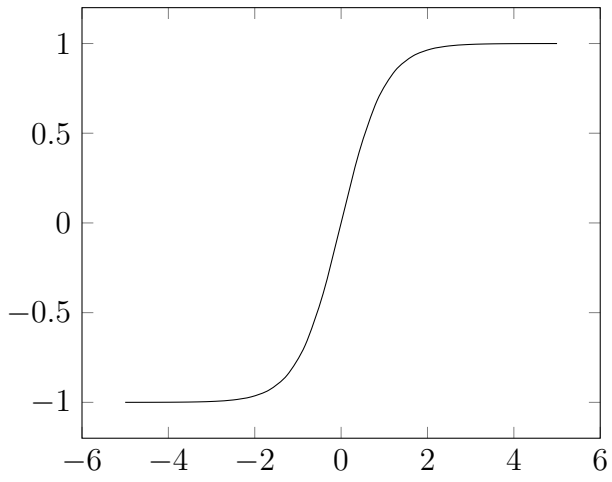
or the so called 'Soft Plus':

$$f(x) = \ln(1 + e^x).$$

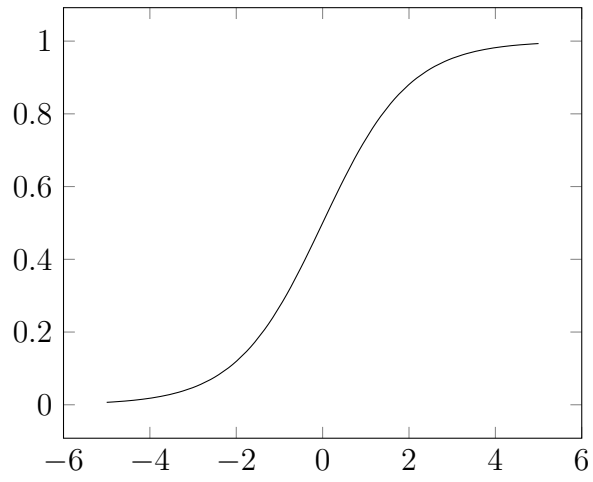
It is also possible to use the identity as activation function but it has been proved that the potential of a multilayer perceptron with such a function is very restricted since a network of the sort with any number of hidden layers can be described as a network with a single hidden layer and it will be shown numerically that it is not the most suited for the method investigated here.

The properties of the chosen activation functions are of most importance. It is common to wish for it to be differentiable everywhere so that gradient-based optimization techniques can be used. Less obvious is the non-linearity of the function, making a three layers (i.e. one hidden layer) feedforward neural network an universal function approximator [5].

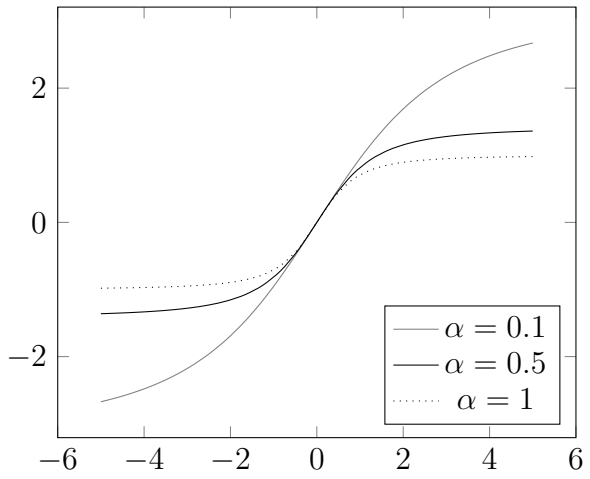
The efficiency of the sigmoid type functions as been demonstrated both numerically, as they are widely used, and theoretically with studies demonstrating their properties as function approximators [5].



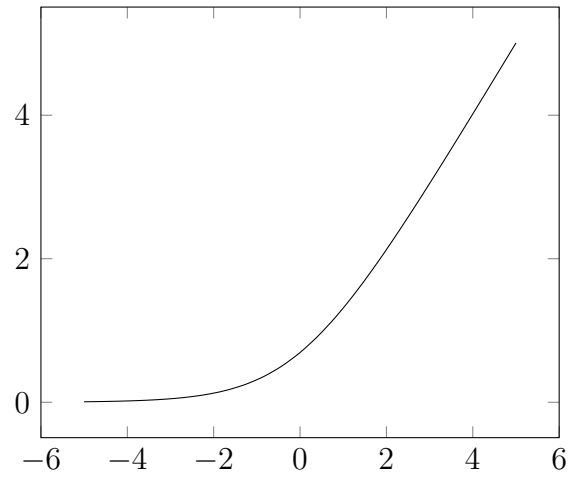
(a) Hyperbolic tangent.



(b) Logistic function (sigmoid).



(c) ISRU.



(d) Soft plus function.

Weights and biases

To complete the forwarding process of the information, it is needed to define how the units are linked together. Since in the networks considered, each layer is fully connected, the input of the unit h_j^{i+1} can be defined as a linear combination of the outputs of the units h_k^i for $k = 1, ..., H_i$, with H_i being the number of units in the layer i.
Let $w_{jk}^i \in \mathbb{R}$ be the weights between h_k^{i+1} and h_j^i, hence the output of the unit j in the layer $i+1$ is:

$$z_j^{i+1}(x) = \sigma\Big(\sum_k w_{jk}^i a_k^i(z_k^i)\Big).$$

Considering a single output feedforward network with one hidden layer of q units, the output N of such a net for an input $x \in \mathbb{R}^d$ and a set of weights and biases $p \in \mathbb{R}^{q(d+2)}$ would be:

$$N(x, p) = \sum_{k=1}^{q} v_k \sigma_k(W_k \cdot x + b_k),$$

where $W_k = (w_{1k}, w_{2k}, ..., w_{dk})$ is the vector of weights linking the input to the hidden unit k.
Fundamentally, the set of parameters p specifies a network among the set of neural networks with a given architecture (size and activation function).
After constructing the class of network used to solve a problem, one has to select the most suited one. That is to say find the p that minimizes the prediction error of the network. This is called "training the network".

1.1.2 Training

The training process of a neural consists in three phases:

- First, determine a loss (or cost) function J, function of the parameters p of the network, which will be used to judge the accuracy of the network's predictions over a set of n data. A typical loss function would be the squared error between the output $N(x_i)$ of the network and the expected output $o(x_i)$ given by experimental data: $J(p) = \sum_{i=1}^{n} |N(x_i) - o(x_i)|^2$;

- The minimization of the functional defined at step one using suitable minimization method. The backpropagation method is usually used to find the derivatives of the output N with respect to the weights and biases and update them through a gradient descent, but any minimization algorithm could be used;

- A test phase during which a new data set not used in the last step are confronted to the network to evaluate if it has been well trained.

The derivatives of a single hidden layer net's output N with respect to any of its arguments are easy to analytically compute and can be found in [12].
In sum, the goal of the training is to find the set of parameters p^* for which the network maps best the training set inputs $\{x_k\}_k$ to the training set outputs $\{o(x_k)\}_k$. Though, since the interest of an ANN is their generalization properties for a model, one has to be very cautious not to let the network over-learn (or over-fit) to the training data-set.
The term over-learning refers to networks which are losing their generalization (interpolation and extrapolation) capacities during learning. That is to say, it is possible for a network to be trained over a sample set, providing accurate predictions on it, but in the meantime loses its capacity of generalization by not being able to give precise prediction for inputs out of the training set.
Thus it is wise to not have only one criteria for termination of the minimization, like a threshold for the loss function or the norm of the gradient, but also to keep a part of the training set as a testing set. These data can then be used to test the generalization capacity of the network during the training by feeding them to a secondary loss function. Then, if the value of this loss function starts stagnating or increasing, it might means that the network is starting to over-learn and the training should be stopped. Another thing to be taken into account is the complexity of the error surface. Though for some very simple problems and if the activation function satisfies some conditions it can be proved that the problem in hand is convex. Usually the function J is highly non-convex, and in particular for the method covered here where the functional is slightly different from usual sum of square differences between the network prediction and the targeted output.
An easy proof of why the loss function is not convex is as follow:

Let N be a single hidden-layered FFNN with single input and output. Assume that

$$p = (w_1, ..., w_i, ..., w_j, .., w_H, b_1, ..., b_H, v_1, ..., v_H)$$

is a minimum of the functional J used to train N. Then interchange two units h_i and h_j in the hidden layer and the corresponding weights w_i and w_j linking them to the previous layer and denote by $\tilde{p}$ the new vector:

$$\tilde{p} = (w_1, ..., w_j, ..., w_i, .., w_H, b_1, ..., b_H, v_1, ..., v_H).$$

Then, for any input x:

$$N(x, p) = \sum_{k=1}^{H} v_k \sigma(w_k x + b_k) = N(x, \tilde{p}).$$

Therefore, $J(p) = J(\tilde{p})$ and the minimum is not unique. Thus the function can't be convex. Therefore, when training using gradient based method, the minimization

has a great dependency on the initialization of the parameters p_0. Some studies has been led trying to find the best range of initialization for deep networks [22]. Although it is practically too complicated to find global minimum for J, it has been empirically shown that using large-size networks can tackle the issue [4]. The probability of the training to end on a "bad" (in the sense of high value of the loss function) minimum is lowered as the network grows in size while most local minima becomes equivalent and of similar performance on test set.

1.1.3 Approximation capabilities of feedforward neural networks

When it comes to approximate a function f, referred as target function, by any kind of method, it is of utmost importance to have theoretical background on the overall error committed.
To discuss the approximation capabilities of a network, one has to differentiate two types of error. According to Barron [1], they are:

- the approximation error;
- the estimation error;

The approximation error is the minimal distance between the space of all neural networks and the target function, it can be understood as the distance from the best neural network approximator to the target function, while the estimation error is the error committed when approximating this ideal neural network. The error of the latter type is expected to have a certain decreasing behavior when the complexity of the network or the quality of the training (number of training samples) increases.
For the networks of interest here, a number of papers suggests that multilayer feedforward neural network are universal approximators (for a wide variety of target functions) [5], [11], [21], [2], [7].

First, Cybenko (1989) [5] showed that single layered perceptrons with sigmoid activation functions are capable to approximate any continuous function on the n-dimensional unit cube:

Theorem 1.1.1. *[5]*

Let I_n be the n-dimensional unit cube and σ be any continuous sigmoidal function. Then finite sums of the form:

$$G(x) = \sum_{j=1}^{N} v_j \sigma(w_j x + b_j)$$

are dense in $C(I_n)$.

Here σ is called a sigmoidal function if it satisfies

$$\sigma(x) \to \begin{cases} 1, & x \to \infty \\ 0, & x \to -\infty \end{cases}$$

This results has then been extended by Hornik to any continuous function ([10], [11]) and not restricted to single hidden-layered FFNN. Let $C^d(\Omega)$ be the set of continuous function $f : \Omega \subset \mathbb{R}^d \to \mathbb{R}$ and $\|f\|_{L^p} = \left(\int_\Omega |f(x)|^p dx \right)^{\frac{1}{p}}$ then:

Theorem 1.1.2. *[10]*
The set of multilayer feedforward neural network with arbitrary many hidden layers is uniformly dense in $C^d(\Omega)$ for any probability measure μ and for the class of norms $\|.\|_p$, with $p \in [1, \infty)$.

This result remains valid for continuous vectorial functions $f : \mathbb{R}^d \to \mathbb{R}^l$.
In other words, for every function in $C^d(\Omega)$ and every $K \in \mathbb{N}$, $K \geq 1$, there exists a multilayer perceptron with K hidden layers capable of approximating it up to any desired accuracy if provided with enough hidden units.
This last theorem holds for all the examples concerned in this work, but in practice, it is not possible to deal with such arbitrary complex neural networks since it makes finding the suitable parameters too complicated. Therefore, the problem is to find convergence rates for the total error, composed of the approximation and the estimation error.
Barron [1][2] provided such convergence rates, function of the number of units, the dimension of the input, the number of training points and a constant dependent on the Fourier transform of the function to approximate. Therefore, his result assume a certain regularity of the target function. He stated that, for the $\|.\|_2$ norm, the convergence rate for the approximation error is of order $(q^{-1/2})$, where q is the number of units of a single hidden layer feedforward network. In other words, for any smooth enough target function f there exists a sequence of networks $\{f_q\}_q$ with q hidden units such that

$$\|f - f_q\|_{L^2} \leq O\big(C_f q^{-1/2}\big)$$

where C_f is a constant dependent of the regularity of f. Furthermore, Barron found a bound for the estimation error, which is function of the number of training points n:

$$\|f - f_q\|_{L^2} \leq O\big(n^{-1/2} \times \log(n)^{1/2}\big).$$

Barron's results hold for single hidden layer feedforward neural networks and for target functions with enough regularity.
McCaffrey and Gallant [20] extended these results to functions in Sobolev spaces $H^m(\Omega)$, where $H^m(\Omega) = \left\{ f;\ \int_\Omega |D^k f(x)|^2 dx < \infty,\ k = 0, ..., m \right\}$ with $D^k f$ being any derivative of order k of f.

1.2 Related work and our contribution

The study of artificial neural networks (ANN) has shown rapid growth in the past decades. It has became one of the most important field among processing systems researches. ANN are of most interest for engineers because of their adaptability, the wide range of problems they can solve, their compatibility with parallel processing and also because, as of now, they are the only suitable method to solve certain type of problems such as image or speech recognition. In the mean time, a rise in the need of numerical solution to differential equations is pulling researches towards more and more computationally efficient solving schemes. However, most of the usual methods (finite differences/elements/volumes/spectral methods) show weakness when it comes to high-order problems, such as those who can arise in financial engineering. Indeed they are based on domain discretizations and therefore the number of unknown grows very fast with the order of the equation, leading to a complicated and computationally expensive algebraic problem. As the use of neural networks has shown to be successful for different fields of engineering, it has come naturally to apply them to differential equations. A first attempt is presented by Lee and Kang [14] where they proposed a method merging finite differences and neural networks (Hopfield model). While they already highlighted the possibility to easily implement such solver on parallel architecture, they show that the numerical solution presents a tendency to be oscillating around the exact solution for some fairly simple cases. A paper from A. J. Meade and A. A. Fernandez [17] introduced a method making use of first order splines and hard limit activation function to solve linear ordinary differential equations (ODE's). Their method requires to impose some conditions on the network parameters, constructing a very specific feedforward network, but they showed both theoretically and numerically that the computed solution has an error decreasing with a quadratic convergence rate with the number of hidden units. Lagaris et al. [12] published a paper, used as a basis for this work, presenting a new method based on neural networks to solve ordinary as well as partial differential equations. This method, presented more in detail in the next chapter of this paper, makes use of a separation of the boundary conditions and the neural network by writing the approximate solution as a sum of two components: the first term is related to the boundary conditions and the second is dependent on the network output to satisfy the differential equation. Therefore, they managed to create a numerical solution written in closed analytic form, making profit of the interpolation properties of neural networks. Contrary to the previous method, the solution constructed has the property of exactly satisfying the boundary conditions of the equation. These method is based on the function approximations qualities of feedforward neural networks, which have been demonstrated many times before [5],[11],[15],[1]. Although that paper only presented the solution of differential equations over regular orthogonal box domains, a second paper issued in 2000 expands the results to irregular arbitrary shaped boundaries [13].

The aforementioned methods all make use of feedforward neural networks and their approximation abilities but do not really make use of deep networks, as the papers only show results for the most shallow networks. More recently, J. Sirignano and K. Spiliopoulos published a paper presenting their method using deep neural networks. Contrary to [12], they use stochastic gradient descent to avoid the issue of forming a mesh (sampling of points on the equation domain). Their algorithm also gives a more computationally efficient way to compute the derivatives of the net, therefore provides the possibility to solve higher dimensional problems with deeper networks. The previous methods make use of a very simple class of ANN, feedforward neural networks. The superposition of nonlinear units can be seen as a function expansion. There also exist methods that use radial basis networks [23] to find solution of differential equations [19].

This paper will present investigations on the neural network method introduced by Lagaris. Firstly, it will be given background informations on neural networks, introducing terminology, positive traits and issues usually encountered when working with neural networks as well as an introduction to the neural network method investigated. Also, the method will be illustrated for different differential equations, farther used for the investigations. These investigations are presented in the second part, divided into three sections. The first one focuses on the training set, the second on the architecture of the network used and the latest on the activation function. The last part will be dedicated to the conclusion as well as possible future work.

Chapter 2

Presentation of the method

In their paper [12] I. Lagaris and A. Likas present a method to solve differential equations using a neural network. This section will be dedicated to the presentation of this method.

They pretend this method can be applied to single ordinary differential equation (ODE), systems of ODE's and to partial differential equations (PDE's) defined on orthogonal box boundaries. They show some interesting features of this method for real life application, such as the limited size of the system to be solved and the interpolation properties of the computed solution as well as the applications for high dimensional problems. This method makes use of simple fully connected feedforward neural network (or multilayer perceptron) that have been showed to be universal approximators (Th. 1.1.2).

2.1 Problem formulation

Consider the following general 2^{nd} order differential equation defined over a domain $D \subset \mathbb{R}^n$:

$$G\big(x, \Psi(x), \nabla\Psi(x), \nabla^2\Psi(x)\big) = 0, \tag{2.1.1}$$

where $x = (x_1, ..., x_n) \in D$ and with Ψ satisfying some boundary conditions, of Neumann and/or Dirichlet type, over the boundary $S = \partial D$.

The main idea of the method is to look for a solution to this equation as a sum of two terms, one satisfying the boundary conditions and the other is constructed as to not contribute to those and is computed using the output of a multilayer feedforward neural network.
To do so, the collocation method is used. The first step is to choose a discretization of the domain D and its boundary S, $\hat{D}$ and $\hat{S}$ respectively. Thus the equation 2.1.1

can be transformed into a system of equations over $\hat{D}$:

$$G\big(x^{(i)}, \Psi(x^{(i)}), \nabla\Psi(x^{(i)}), \nabla^2\Psi(x^{(i)})\big) = 0,\ x^{(i)} \in \hat{D}, \tag{2.1.2}$$

with the solution Ψ still satisfying the constraints on the boundaries.
The collocation method suggests to look for a solution $\Psi_t(x, p)$, called trial solution, where $p \in \mathbb{R}^d$ is a set of parameters that we want to adjust to solve the equation (2.1.2) in the following sense:

$$\begin{cases} \min\limits_{p\in\mathbb{R}^d} \sum\limits_{x^{(i)}\in\hat{D}} \Big(G\big(x^{(i)}, \Phi(x^{(i)}), \nabla\Phi(x^{(i)}), \nabla^2\Phi(x^{(i)})\big)\Big)^2 \\ p \in \mathbb{R}^d \text{ such that } \Psi_t \text{ satisfies the boundary conditions,} \end{cases} \tag{2.1.3}$$

where the minimization is under constraints from the boundary conditions of the initial equation. That is to say we want to find the set of parameters p^* minimizing the square error made on the function defining the differential equation. We consider the square error in order to have better properties on the function, making the process of minimization easier and less costly.

The method presented in [12] suggests to use the following trial solution:

$$\Psi_t(x, p) = A(x) + F\big(x, N(x, p)\big), \tag{2.1.4}$$

where $A(x)$ is such that it satisfies the boundary conditions and F is constructed as to not contribute to the boundary conditions, i.e. $\forall x \in \hat{S},\ F\big(x, N(x, p)\big) = 0$. $N(x, p)$ is the output of a single output feedforward neural network with n input units. Here p represent the weights and bias of this network. By doing so, the constrained minimization problem (2.1.3) becomes an unconstrained minimization problem, since the conditions on p are already satisfied by A, by construction of the trial solution:

$$\min_{p\in\mathbb{R}^d} \sum_{x^{(i)}\in\hat{D}} G\big(x^{(i)}, \Phi(x^{(i)}), \nabla\Phi(x^{(i)}), \nabla^2\Phi(x^{(i)})\big)^2, \tag{2.1.5}$$

which is more convenient to handle numerically and mathematically. Since A makes no use of any adjustable parameter, the only parameters dependent term is $N(x, p)$ the output of the network. Written as (2.1.4), Ψ_t is, with some non-costly assumptions on A and F, easily differentiable and can conveniently be used for further calculations. To ensure that the solution satisfies the correct boundary conditions, it is only needed that p is such that:

$$F(x, N(x, p)) < \infty.$$

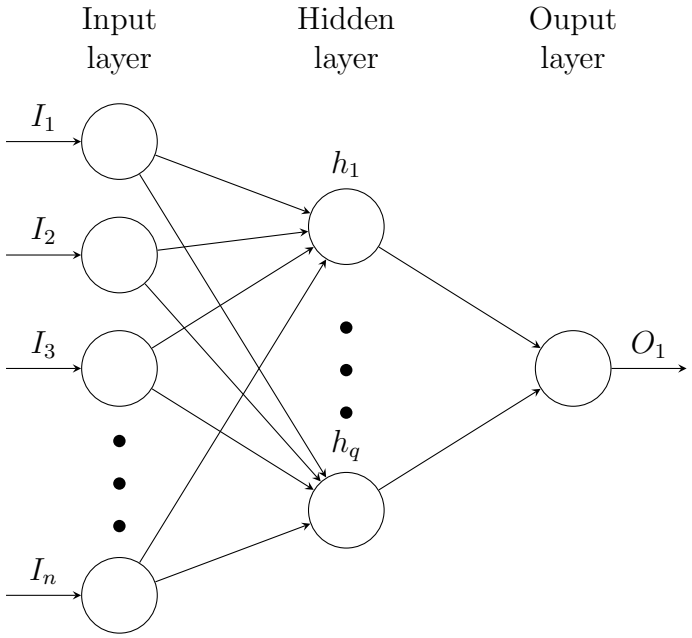
Contrary to other methods, the solution constructed here is not a discrete solution, neither of limited differentiability, it is indeed differentiable and in a closed analytic form.
The construction of the functions A and F is specific to the equation's characteristic and will be discussed for some class of differential equations in section 2.3.

2.2 Construction of the Network

After rewriting the problem and constructing the trial solution, what is left is to construct the neural network and to train it to satisfyingly solve the problem 2.1.5. The networks used for this method are fully connected multilayer perceptrons with single linear output.
Given the expressing of the trial solution Ψ_t, the network has to be trained using the function

$$J(p) = \sum_{x^{(i)} \in \hat{D}} \Big(G(x^{(i)}, \Psi, \nabla\Psi, \nabla^2\Psi) \Big). \tag{2.2.1}$$

If we consider a single hidden layer it is easy to express derivatives of this network with respect to any of the parameters. The training of the network here being equivalent to the minimization problem 2.1.5, it is classified as supervised learning [9] and the algorithms used to solve this kind of problem needs the gradient and/or Hessian of the loss function J. In most of the neural network's training processes only the gradient with respect to the weights is needed, but in this case this gradient involves the derivatives of the network with respect to its inputs x_j.
Typical, efficient architecture would be the following multilayer perceptron with one hidden layer, with linear output and the sigmoid (logistic function) as activation function for the $q \in \mathbb{N}$ hidden units, taking $x = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$ as input:



The last step of the method is to train the network, that is to say find the parameter p^* which minimizes the loss function $J(p)$. The minimization in this case

is a simple unconstrained minimization over $\mathbb{R}^n$. The minimization is done using the *minimization* function from the library *scipy.optimization*. This function provides a variety of minimization algorithms such as BFGS, L-BFGS, conjugate gradient. As discovered in [12], the BFGS shows better performances on this problem. Also, *scipy*'s L-BFGS (low memory BFGS) implementation has proved to be more stable than the standard version of the algorithm. Therefore, it was the one used for the experiments presented here and will be referred as BFGS for simplicity.

2.3 Brief illustration

This section presents trial solution for different types of differential equations. It is a rewrite of what can be found in [12]. Also, it is to be noted that the choice of the trial solution is not essentially unique.

First-order ordinary differential equation

Consider the following general first-order with the associated initial value condition:

$$\begin{cases} \Psi'(x) = f\big(x, \Psi(x)\big), \ \forall x \in [0, a] \\ \Psi(0) = A. \end{cases} \tag{2.3.1}$$

The trial solution for this equation can be constructed as:

$$\Psi_t(x, p) = A + x(x - a)N(x, p),$$

or as well:

$$\Psi_t(x, p) = A + \sin(\frac{x}{a}\pi)N(x, p).$$

In this case, the loss function is:

$$J(p) = \sum_{x^{(i)} \in \hat{D}} \big(\partial_x \Psi_t(x^{(i)}, p) - f(x^{(i)}, \Psi_t(x^{(i)}, p)\big)^2.$$

Second order equation

For the following initial value problem:

$$\begin{aligned} &\Psi''(x) = f(x, \Psi, \Psi'), \ \forall x \in [0, a] \\ &\Psi(0) = A \\ &\Psi'(0) = B, \end{aligned}$$

an appropriate trial solution is given by:

$$\Psi_t(x, p) = A + Bx + x^2 N(x, p),$$

and the loss function is similar to the first order:

$$J(p) = \sum_{x^{(i)} \in \hat{D}} \left(\partial^2_{xx} \Psi_t(x^{(i)}, p) - f(x^{(i)}, \partial_x \Psi_t(x^{(i)}, p), \Psi_t(x^{(i)}, p)\right)^2.$$

Consider the same equation but with Dirichlet conditions on both sides of the interval:

$$\begin{aligned} \Psi(0) &= A, \\ \Psi(a) &= B. \end{aligned}$$

Then the trial solution is slightly changed:

$$\Psi_t(x, p) = A(a - x) + Bx + x(a - x)N(x, p),$$

and the loss function is unchanged.

System of ODE's

In the case of a system K ODE's, the trial functions are the same as presented before, each function making use of a different network. The loss function of the system is then naturally the sum of the K loss function of each ODE. For instance, for a system of K first order equations:

$$\begin{aligned} \Psi_i'(x) &= f_i(x, \Psi_1, \Psi_2, ..., \Psi_K), \ x \in [0, a] \\ \Psi_i(0) &= A_i, \end{aligned}$$

the trial solutions are

$$\Psi_{t_i}(x, p) = A_i + xN_i(x, p_i),$$

and the networks are trained uniformly to minimize:

$$J(p) = \sum_{k=1}^{K} \sum_{x^{(i)} \in \hat{D}} \left(\partial_x \Psi_{t_k}(x, p) - f_k(x^{(i)}, \Psi_{t_1}, ..., \Psi_{t_K})\right)^2.$$

Higher dimensional equations

The extension to partial differential equations is straightforward and is presented in [12] for the second order Poisson's equation with Dirichlet and Dirichlet-Neumann boundary conditions. For higher dimensions or different boundary conditions, the method stays the same and also works for problems with irregular boundaries [13]. The next section presents results from an implementation of the method for different test problems.

2.4 Examples

Considering examples of differential equations that commonly arise in scientific problems, the goal will be to numerically show the qualities of the different architectures by comparing the accuracy of the computed solution. The equations we consider are of all kinds, from single ODE's of first or second order to PDE's. Those examples are simply particular cases of those presented in Section 2.3, for which the analytical solution is well known.
The first step of this study is to apply the method as it is presented in [12] to, first show that the neural network can actually solve those equations, and secondly to validate the code implemented.

The neural network solver has been implemented in the language *Python* and makes use of the libraries *numpy* and *scipy.optimize* for their efficient implementation of linear algebra's functions and algorithms. Another library used is *autograd* [16] which allows efficient computation of derivatives of numpy code. It will be used to compute the gradient of the loss function, saving computational time, increasing accuracy, during the training compared to multiple approximations of the gradient for each iteration of the algorithm.
The neural networks in this section are the same as in the reference paper: one hidden layer feedforward networks, with ten hidden units and using the logistic function $\sigma(x) = 1/(1 + e^{-x})$ as activation function for each hidden unit. The hidden units are each linked to one different bias unit. The output will be a linear combination of the hidden units, without bias. The training part will be done using the BFGS method [6], a quasi-Newton algorithm frequently used in deep learning, using a ten equidistant points training set over the definition interval of the equation. Note that the boundary of the interval are part of this sample. As in the reference paper, the accuracy displayed is maximal point-to-point deviation between the computed solution and the analytical solution $\Delta\Psi_t(x, p) = |\Psi_t(x, p) - \Psi_a(x)|$:

$$\|\Delta\Psi_t\|_{\hat{D}} = \max_{x^{(i)} \in \hat{D}} \Delta\Psi_t(x^{(i)}, p),$$

where $\hat{D}$ is the training set used to perform the training of the network.

2.4.1 First-order ordinary differential equation

The first problem considered in [12] is the following:

$$\begin{cases} \Psi'(x) + \left(x + \frac{1+3x^2}{1+x+x^3}\right)\Psi(x) = x^3 + 2x + x^2\frac{1+3x^2}{1+x+x^3}, \ x \in (0, 1] \\ \Psi(0) = 1 \end{cases}, \tag{2.4.1}$$

for which the analytical solution is:

$$\Psi_a(x) = \frac{e^{-x^2/2}}{1 + x + x^3} + x^2.$$

Following Section 2.1, the trial solution for this problem is chosen as follow:

$$\Psi_t(x,p) = 1 + xN(x,p),$$

with $N(x,p)$ the output of the neural network as described before. Using a ten equidistant points grid to train the network with the BFGS method has shown interesting results regarding the accuracy, as can be seen in Figure 2.1.

For this test, the value of the loss function after training is $J(p^*) \approx 4 \times 10^{-12}$. As mentioned earlier, the trial solution as it was constructed shows impressive capacities when it comes to computing the solution outside the training set. Indeed Figure 2.1b shows that the accuracy over randomly sampled points in the interval is not higher than at the training points.

Another similar first-order problem was also solved:

$$\begin{cases} \Psi'(x) + \frac{1}{5}\Psi(x) = e^{-x/5}\cos(x), \ x \in (0,2] \\ \Psi(0) = 0 \end{cases}, \tag{2.4.2}$$

with the analytical solution:

$$\Psi_a(x) = e^{-x/5}\sin(x),$$

for which the error on the definition interval is presented in Figure 2.2.

The order of accuracy is similar as for the first problem, with the value of the loss function reaching $J(p^*) \approx 6 \times 10^{-11}$ at the end of the training.

On these two examples, the neural network based method seems to perform quite well, with the solution of high accuracy even though the number of training points was very limited (ten points).

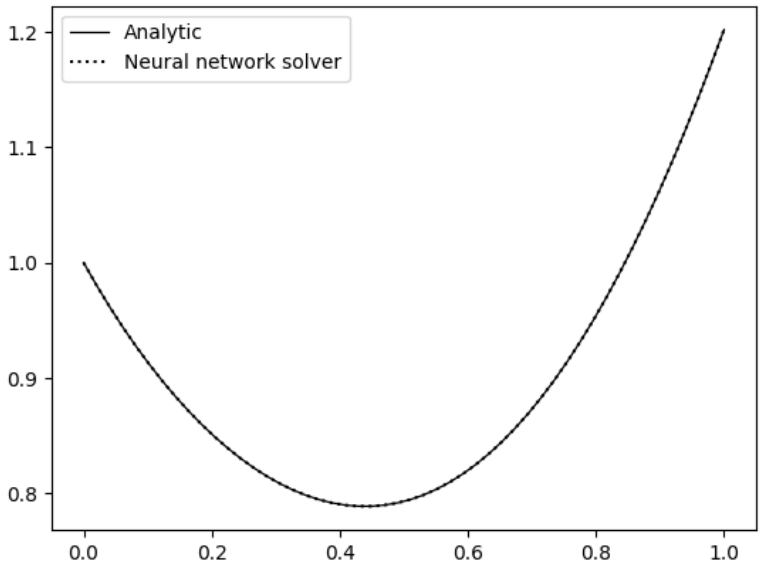
2.4.2 Second-order ordinary differential equation

The solution as it is constructed is easily differentiable, hence it is also easy to apply the method to arbitrary high order differential equations, without increasing the size of the problem or solving a more complex linear system. In this section, the solution of a second-order linear ODE is displayed, for two different boundary conditions types, an initial value problem:

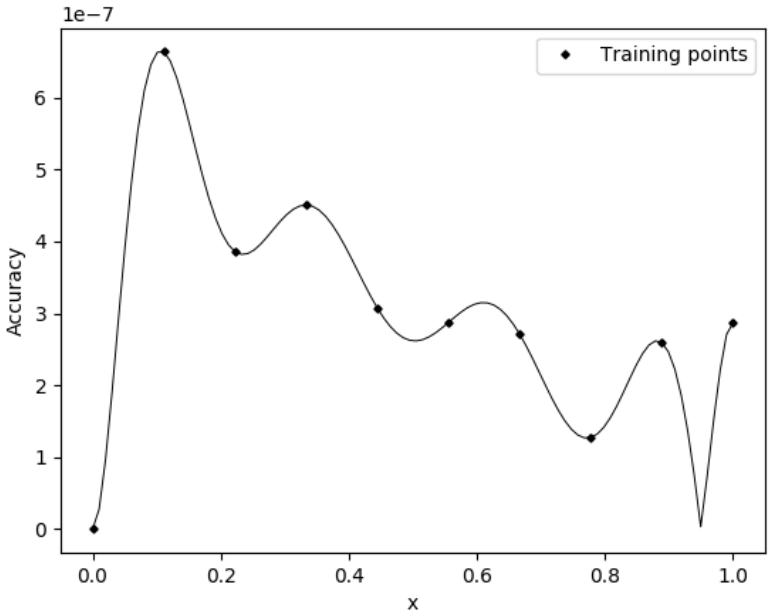
$$\begin{cases} \Psi''(x) + \frac{1}{5}\Psi'(x) + \Psi(x) = -\frac{1}{5}e^{-x/5}\cos(x), \ x \in (0,2] \\ \Psi(0) = 0 \\ \Psi'(0) = 1 \end{cases}., \tag{2.4.3}$$

and a boundary value problem:

$$\begin{cases} \Psi''(x) + \frac{1}{5}\Psi'(x) + \Psi(x) = -\frac{1}{5}e^{-x/5}\cos(x), \ x \in (0,1) \\ \Psi(0) = 0 \\ \Psi(1) = \sin(1)e^{-1/5} \end{cases}. \tag{2.4.4}$$



(a) Two solutions are almost not distinguishable.



(b) Maximal error at training points: 6×10^{-7}.

Figure 2.1: Comparative plot (a) and accuracy plot (b) for problem (2.4.1)

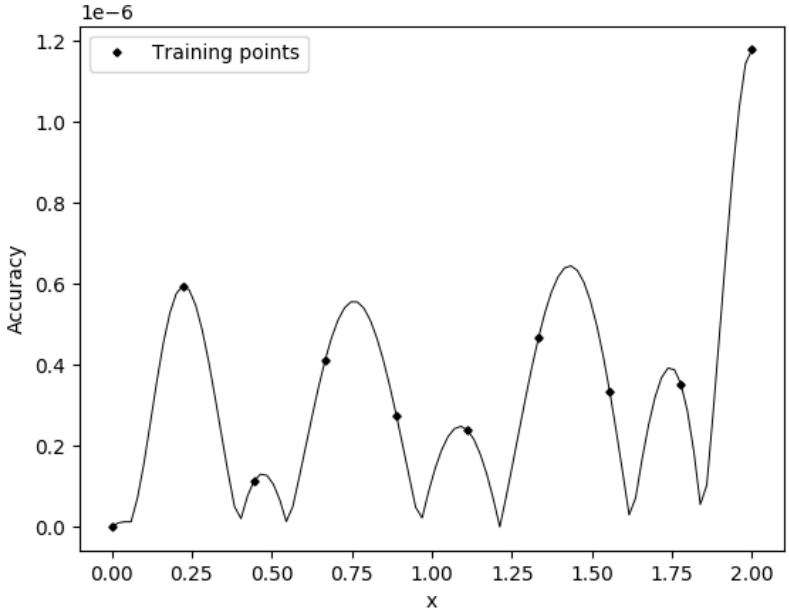


Figure 2.2: Problem 2.4.2: accuracy of the solution. $\|\Delta\Psi_t\|_{\hat{D}} = 1.18 \times 10^{-6}$

The equation is such that the solution is identical for both of these problems:

$$\Psi_a(x) = e^{-x/5}\sin(x).$$

The trial solution used to solve equation (2.4.3) is:

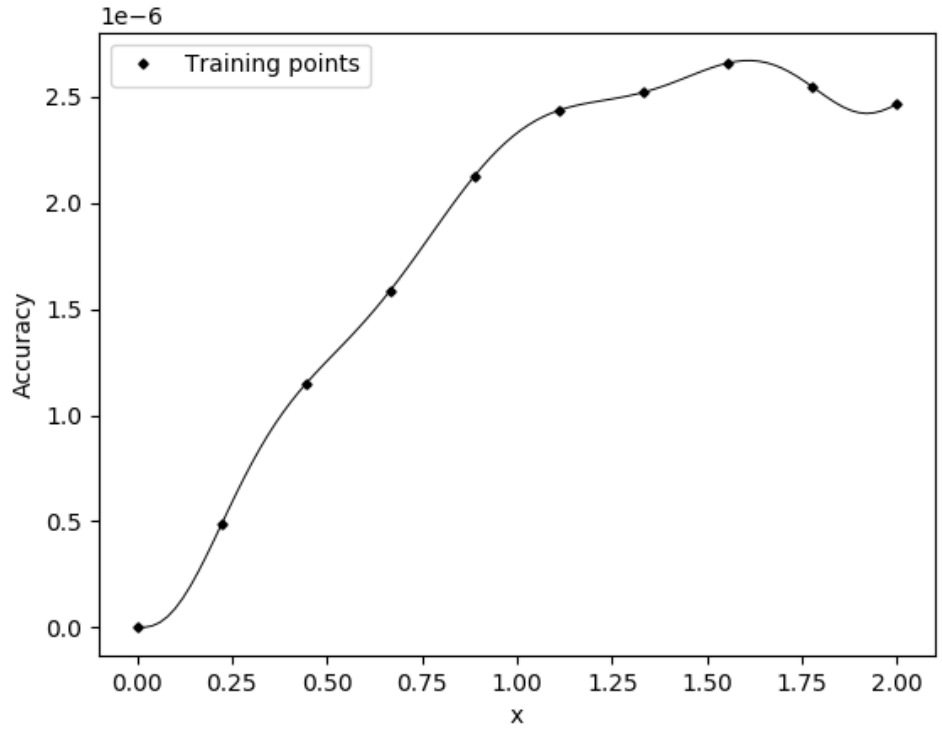
$$\Psi_t(x) = x + x^2 N(x, p),$$

and for equation (2.4.4):

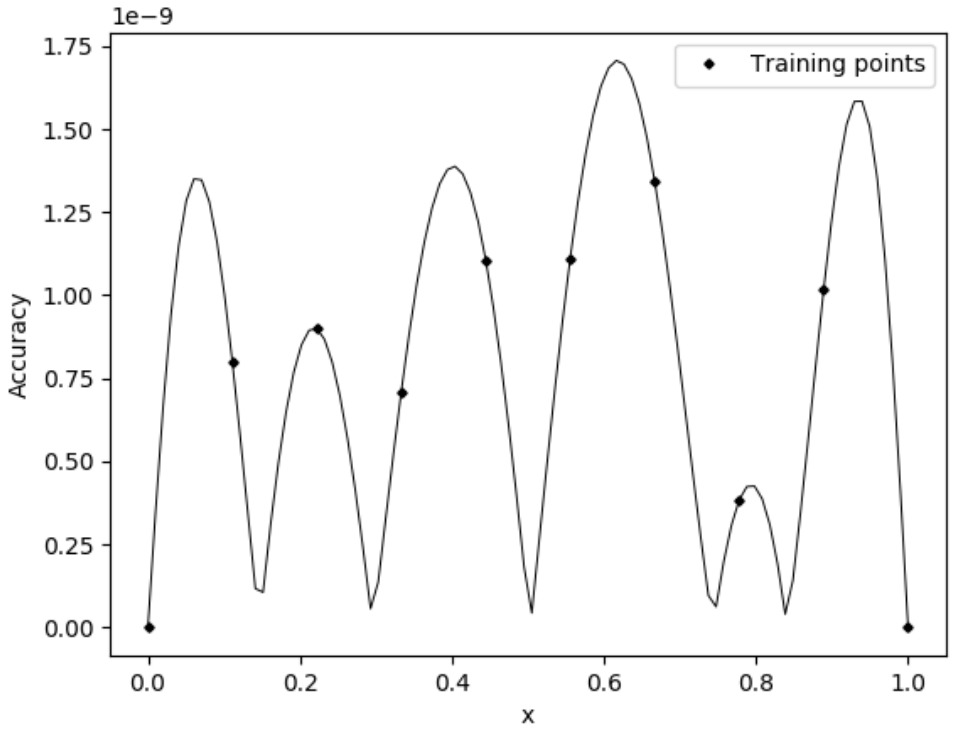
$$\Psi_t(x) = x\sin(1)e^{-1/5} + x(1-x)N(x, p).$$

Similarly as in the previous section for first-order ODE's, the method will be validated on those cases by displaying the deviation $\Delta\Psi_t(x, p)$ from the exact solution (2.3).

The accuracy difference between the two examples can be explained by the information missing on the right side of the interval, the network being less accurate the farther it goes away from the boundary condition. This leads to an error globally increasing with x (Fig. 2.3a).



(a) $\|\Psi_t\|_{\hat{D}} \approx 1.12 \times 10^{-10}$.



(b) $\|\Delta\Psi_t\|_{\hat{D}} = 1.34 \times 10^{-13}$.

Figure 2.3: Accuracy for the second order ODE: (a) with initial value conditions; (b) with boundary value conditions.

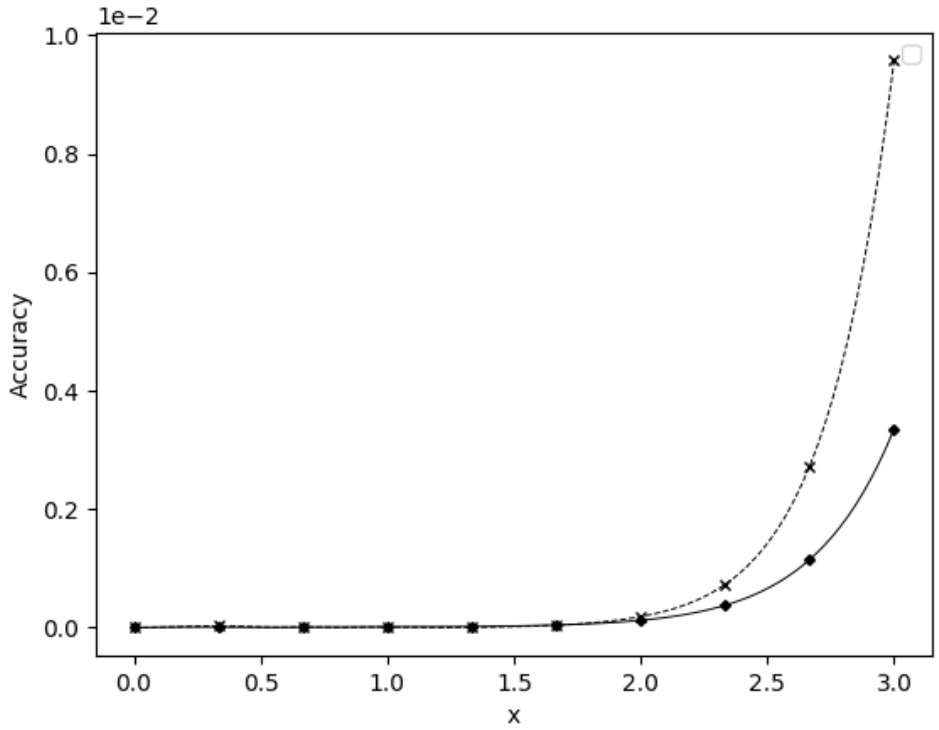


Figure 2.4: Accuracy of the solutions of problem 2.4.5. The continuous line corresponds to Ψ, the dashed line to Φ

2.4.3 System of coupled ODE's

This section presents the results when the method is applied on the following system of two coupled first-order ODE's:

$$\begin{cases} \Psi'(x) = \cos(x) + \Psi^2(x) + \Phi(x) - 1 - x^2 - \sin^2(x), \ x \in (0,3] \\ \Phi'(x) = 2x - (1+x^2)\sin(x) + \Psi(x)\Phi(x), \ x \in (0,3] \\ \Psi(0) = 0, \ \Phi(0) = 1 \end{cases} \tag{2.4.5}$$

but it should be possible to generalize and apply the method to a system of k ordinary differential equations of arbitrary order (see Section 2.3).
Unlike the other equation, the accuracy of the solution computed is much smaller, for the loss around 7×10^{-7}, the accuracy of the solutions is around 10^{-2}, as displayed in Figure 2.4. The accuracy at the training points is denoted by diamonds (for Ψ) and cross (Φ).

It should be noted that for system of equations, the number of unknowns (bias and weigths) is multiplied by the number of equations in the system compared to single ODE's. Indeed each trial solution Ψ_t and Φ_t make use of a different network

with similar architecture but with different set of parameters p_1 and p_2, both network trained as one, only related by the loss function $J(p_1, p_2)$.

2.4.4 Partial differential equations

Partial differential equations are of most importance, arising in most physical problems as well as in biology. Most of the time in real life problems, it is not possible to find an exact solution and the additional dimensions compared to ODE's make them very costly to solve numerically, even when reduced to a two dimensional space. Thus, neural network can be of great help to solve this issue, since the dimension of the discrete problem does not grow as fast when the number of points increases as it can for more ordinary method (finite differences, finite volumes, spectral methods or finite elements).
In this section, the method is only applied for second order elliptic equations within the unit square domain $\Omega = [0,1] \times [0,1]$.

2.4.5 Poisson's equation

Dirichlet boundary conditions

The first PDE considered is the very common Poisson's equation with Dirichlet boundary conditions:

$$\begin{cases} \Delta\Psi(x,y) = e^{-x}(x-2+y^3+6y), & x \in \Omega \\ \Psi(0,y) = f_0(y) = y^3, & y \in [0,1] \\ \Psi(1,y) = f_1(y) = (1+y^3)e^{-1}, & y \in [0,1] \\ \Psi(x,0) = g_0(x) = xe^{-x}, & x \in [0,1] \\ \Psi(x,1) = g_1(x) = e^{-x}(x+1), & x \in [0,1] \end{cases} \tag{2.4.6}$$

This PDE admits as a solution the function:

$$\Psi_a(x) = e^{-x}(x+y^3).$$

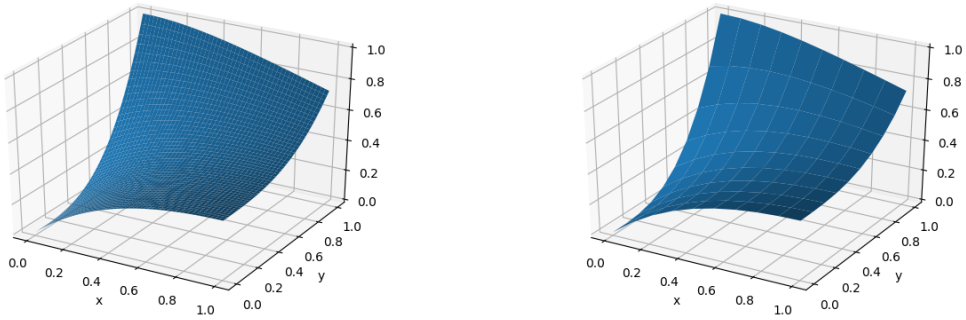
Following the process described in the previous section, one can look for a trial solution for this problem under the form:

$$\Psi_t(x,y,p) = A(x,y) + x(1-x)y(1-y)N(x,y,p),$$

with

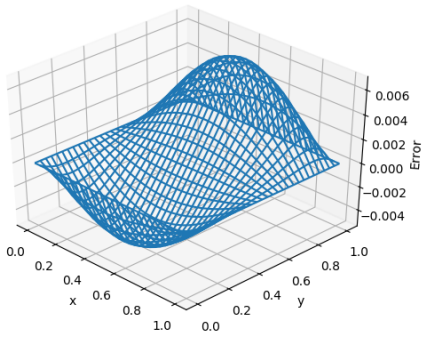
$$\begin{aligned} A(x,y) = &(1-x)y^3 + x(1+y^3)e^{-1} \\ &+(1-y)\big(xe^{-x} - xe^{-1}\big) \\ &+y\Big(e^{-x}(x+1) - \big((1-x)+2xe^{-1}\big)\Big) \end{aligned},$$

with Figure 2.5 displays the solution computed with this method as well as the error surface and a plot of the analytic solution Ψ_a.



(a) Exact solution Ψ_a.

(b) Trained trial solution Ψ_t.



(c) Error surface $\Psi_t - \Psi_a$ on the training set.

Figure 2.5: Problem 5 is solved using a neural network of 8 hidden units with sigmoid activation function and ten training points.

Mixed Dirichlet-Neumann boundary conditions

A common boundary condition type arising in mathematical models of physical systems is the Neumann boundary condition. Those are constraints on the (normal) derivatives of the function on the boundary. They are often coupled with Dirichlet's type conditions to form mixed boundary conditions problems such as:

$$\begin{cases} \Delta\Psi(x,y) = \sin(\pi x)\big(2-\pi^2 y^2\big), & x \in \Omega \\ \Psi(0,y) = f_0(y) = 0, & y \in [0,1] \\ \Psi(1,y) = f_1(y) = 0, & y \in [0,1] \\ \Psi(x,0) = g_0(x) = 0, & x \in [0,1] \\ \frac{\partial\Psi}{\partial y}(x,1) = g_1(x) = 2\sin(\pi x), & x \in [0,1] \end{cases} \tag{2.4.7}$$

for which the analytical solution is given by:

$$\Psi_a(x,y) = y^2 \sin(\pi x).$$

One possible trial solution to solve this equation is:

$$\Psi_t(x,y,p) = 2y\sin(\pi x) + x(1-x)y\big(N(x,y,p) - N(x,1,p) - \partial_y N(x,1,p)\big).$$

Non linear PDE

This method of computing solutions of PDE's is not limited by linearity of the equations. The following second order, non linear equation can also be solved using the same kind of neural network, with similar number of parameters:

$$\begin{cases} \Delta\Psi(x,y) + \Psi(x,y)\frac{\partial\Psi}{\partial y}(x,y) = \sin(\pi x)\big(2-\pi^2 y^2 + 2y^3\sin(\pi x)\big), & x \in \Omega \\ \Psi(0,y) = 0, & y \in [0,1] \\ \Psi(1,y) = 0, & y \in [0,1] \\ \Psi(x,0) = 0, & x \in [0,1] \\ \frac{\partial\Psi}{\partial y}(x,1) = 2\sin(\pi x), & x \in [0,1] \end{cases} \tag{2.4.8}$$

In this case, an appropriate trial solution is the same as for the previous linear equation with similar boundary conditions, which implies that the linearity of the equation does not change the way the method is constructed.

Chapter 3

Accuracy of single hidden layer network

By construction of the trial solution, there is no network of any complexity such that it corresponds exactly to the analytical solution. Indeed, the analytical solution is not expected to have a decomposition of the kind: $\Psi(x) = A(x) + F(x)$, with $F \in C^k(\Omega)$ such that it can be approximated by a feedforward net. Therefore, the approximation error of the trial solution is never null and is not converging to the exact solution.
Though, this error might be insignificant compared to the estimation error from the training.

3.1 On the training set

One parameter of the network that is likely to change or increase in size is the training set. It is the only parameter that can be changed after the construction of the network and/or after training. Therefore, it is interesting to know how changes on this set influence the computed solution.

3.1.1 Size of the training set

As presented in Section 1.1.3, for a fixed architecture of a neural network, the estimation error $e(n)$ for the norm $\|.\|_{L^p}$ is expected to decrease as $O\big(n^{-1/2}\big)$, where n is the number of training points used for the training. For readability purposes, the logarithm of this error may also be considered instead.

In this section, the neural networks used are all composed of 8 hidden units with sigmoid activation functions. They will be trained over a training set of equidistant points of increasing sizes and the resulting errors will be compared to expected theoretical convergence rates.
For this section, the accuracy to the exact solution will be estimated using the L^2

norm as well as for the maximal deviation $\|\Delta\Psi_t\|_{\hat{D}} = \max_{x_i \in \hat{D}} |\Psi_t(x_i, p^*) - \Psi(x_i)|$ for which there is no theorem about the convergence of such network. It will be made use of the analytical form of the trial solution to compute the continuous L^2 norm:

$$\|f\|_{L^2(\Omega)} = \int_\Omega |f(x)|^2 dx,$$

using a quadrature formula. This norm will be computed using *scipy.integrate*'s routine *quad* which gives very accurate results on the integration.

In this method, the training set can be chosen arbitrarily big since it does not come from experimental data or observations. Thus the estimation error can theoretically be made as small as desired by increasing the computational costs. However one of the main interests of this neural network solution is its generalization capabilities, so having an overly big training size might make the method lose its advantage compared to other traditional methods. Table 3.1 shows the accuracy with the number of training points $e(n)$, where $e(n) = \|\Psi_t - \Psi\|$ is the error for n points taken for a certain norm. Table (a) presents the accuracy for the L^2 norm and in table (b) is displayed the maximal deviation for different values of n.
As expected, the tables show a good improvement of the precision in both norm. Also, the logarithm of the error plot along n displayed in 3.1 for first-order ODE's shows that the curve has a convergence similar to $e^{-c\sqrt{n}}$, for some constant $c \in \mathbb{R}$. Moreover, the table shows that the maximum deviation has identical convergence rates as the L^2 error. However, the curves does not follow strictly this slope because the result of the minimization is not always the global minimum of the loss function. For the second-order problems, the behavior is much more chaotic, making impossible to extract a possible convergence rate, even though the solution is sometimes of even better accuracy than for first-order ODE's.

3.1.2 Different choice of grid

While generally the size of the discretization domain is the easiest (but not the most practical) way to improve accuracy, some method like the finite elements or spectral methods can be made more precise with a different grid choice. One of the possibilities is to use the Gauss-Legendre grids, where the points are the zeros of a certain polynomial. This section presents the investigation on the effect of the grid choice for the neural network method.
For this purpose, the previous grid of n equidistant points is replaced by a grid of Gauss-Legendre-Lobatto nodes. For reminder, these nodes compose the set $\{x_i\}_{i=1}^n$, $-1 = x_1 < x_2 < ... < x_n = 1$ where x_i is the *i-th* root of the derivate of the Jacobi polynomial:

$$J_n^{(1,1)}(x_i) = 0, \ \forall i = 2, ..., n-1$$

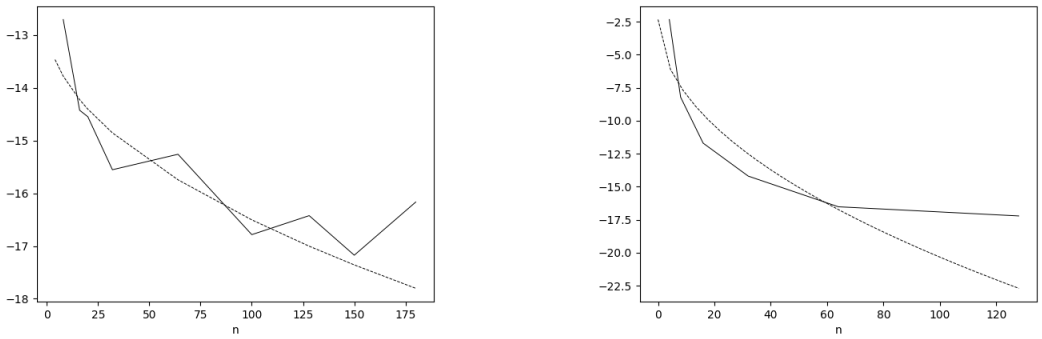
Table 3.1: Accuracy of example problems as a function of the number N of training points.

(a) Error in L^2 norm.

n	Problem 1	Problem 2	IV problem	BV problem
4	6.49×10^{-4}	9.73×10^{-2}	1.23×10^{-3}	1.61×10^{-6}
8	5.15×10^{-6}	2.73×10^{-4}	5.35×10^{-7}	2.25×10^{-9}
16	1.08×10^{-6}	8.34×10^{-6}	1.48×10^{-7}	2.89×10^{-9}
32	4.06×10^{-7}	6.86×10^{-7}	9.24×10^{-8}	1.61×10^{-9}
64	2.36×10^{-7}	6.71×10^{-8}	3.03×10^{-8}	1.39×10^{-9}
128	7.37×10^{-8}	3.35×10^{-8}	3.05×10^{-8}	2.45×10^{-9}

(b) Maximum deviation.

n	Problem 1	Problem 2	IV problem	BV problem
4	6.11×10^{-3}	5.70×10^{-2}	1.11×10^{-3}	1.5×10^{-5}
8	4.27×10^{-5}	1.81×10^{-4}	4.88×10^{-7}	4.26×10^{-9}
16	1.4×10^{-5}	5.9×10^{-6}	1.44×10^{-7}	5.07×10^{-9}
32	9.76×10^{-6}	5.29×10^{-7}	9.76×10^{-8}	2.68×10^{-9}
64	4.06×10^{-7}	6.08×10^{-8}	3.43×10^{-8}	2.48×10^{-9}
128	1.39×10^{-7}	4.15×10^{-8}	4.51×10^{-8}	4.68×10^{-9}



(a) Problem 1, $c = 3.8$. (b) Problem 2, $c = 1.8$.

Figure 3.1: Comparative logarithm plot of L^2 norm (full line) and curve of $e^{-c\sqrt{n}}$ (dashed line) for two first-order ODE's.

with

$$J_n^{(1,1)}(x) = 2^{-n} \sum_{k=0}^{n} \binom{n+1}{k} \binom{n+1}{n-k} (x-1)^{n-1}(x+1)^k$$

forming a set of orthogonal polynomial for the L^2 scalar product weighted by $w(x) = (1-x^2)$:

$$\|\psi(x)\|_{L^2(\Omega,w)} = \int_\Omega \psi(x)(1-x^2)dx$$

The set can be rescale to any interval $[a, b]$:

$$x_i = \frac{x_i}{b-a} + (b-a), \ \forall i \in \{0, 1, ..., n\}$$

These Gauss-Legendre-Lobatto nodes are preferred to other Gauss-Legendre grids since they include the boundary $\{a, b\}$. Without those, the computed solution Ψ_t can be lose accuracy significantly (see table 3.2). As comparison, accuracy is presented for grids of uniformly distributed points over the interval, hence not including the boundaries.

Table 3.2 shows the deviation from the analytical solution for each training point for a certain first estimation of the parameters p_0 and the average resulting L^2 norm. The maximum deviation can be estimated as the maximum deviation over the training set. Indeed it has been shown empirically that the deviation at any points is not higher than at the training points [12].
From this table, one can see that training at Gauss-Legendre and Gauss-Legendre-Lobatto nodes results in better accuracy, both in L^2 and L^∞, compared to equidistant points, even though the first type does not include boundaries. The training time remains equal for either of the sets. Obviously, using any randomly distributed training set ($\hat{D}_U$) is not a possibility since it can give very bad results.

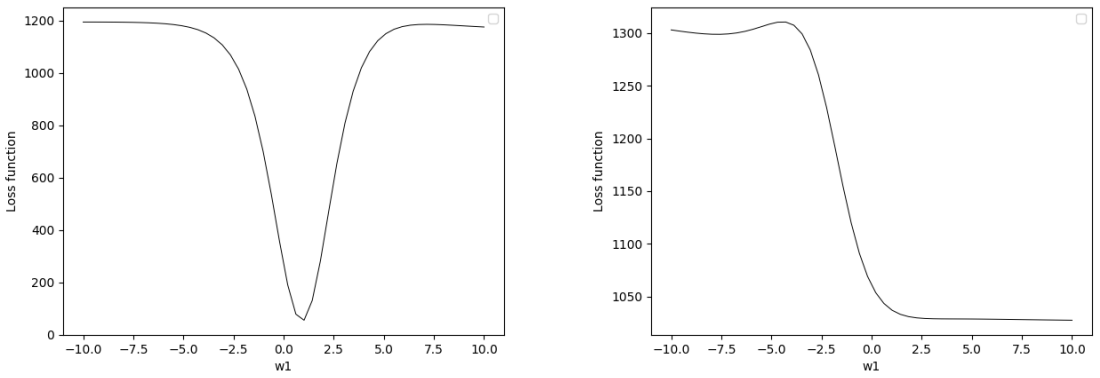
TP	$\hat{D}_{GLL}$	$\hat{D}_{GL}$	$\hat{D}_{Eq}$	$\hat{D}_U$
x_1	0	1.2×10^{-7}	0	9.3×10^{-1}
x_2	-3.1×10^{-7}	-1.9×10^{-7}	-1.5×10^{-7}	3.9
x_3	5.3×10^{-7}	2.2×10^{-7}	-4.8×10^{-7}	-7.2
x_4	-7.4×10^{-7}	-2.1×10^{-7}	-1.3×10^{-7}	-3.02×10^{-3}
x_5	7.1×10^{-7}	1.6×10^{-7}	-1.1×10^{-7}	-3.8×10^{-4}
x_6	-5.6×10^{-7}	-9.2×10^{-8}	-4.7×10^{-7}	-2.9×10^{-4}
x_7	2.8×10^{-7}	-6.1×10^{-9}	3.8×10^{-8}	-2.6×10^{-4}
x_8	-5.1×10^{-8}	8.6×10^{-8}	5.1×10^{-9}	-1.1×10^{-4}
x_9	-9.1×10^{-8}	-1.1×10^{-7}	-4.4×10^{-7}	-1×10^{-4}
x_{10}	4.9×10^{-8}	6.3×10^{-8}	6.6×10^{-7}	-9.6×10^{-5}
$\|\Psi_t - \Psi\|_{L^2}$	4.67×10^{-7}	5.1×10^{-7}	2.65×10^{-6}	2.6×10^{-1}

Table 3.2: Error of first-order ODE solution at ten training points (TP) for different training sets: Gauss-Legendre-Lobatto nodes ($\hat{D}_{GLL}$), equidistant grid ($\hat{D}_{Eq}$) and Gauss-Legendre grid not including boundary ($\hat{D}_{GL}$). $\hat{D}_U$ is a random discretization following an uniform distribution. Last row shows an average L^2 deviation of the trial solution for each type of grid.

3.2 On the network complexity

When designing a neural network, it is usually advised to have more training points than parameters in the net, to ensure a better training. There is no absolute rule about this but one that often comes up is to use a training set of around ten times the number of parameters. Though, for this method, it seems like it is possible to train the network correctly no matter the number of points used.

Also deeper networks (with more hidden layers) involve more parameters to be train, allowing the network to approximate more complex features of the target function but results in a greater complexity for the training process. Then, even if there were a relation between the depth of a network and the accuracy of the trial solution associated, it is not given that the best set of parameters has been reached after successful minimization. Indeed the surface error resulting from $J(p)$ is not necessarily convex or easy to minimize. Figure 3.2 shows two error curves when varying one weights w_1 linking the input to the first (among ten) hidden nodes, for



(a) (b)

Figure 3.2: Example of error surfaces and how they can be ill shaped for gradient descent.

a network solving an ODE with one hidden layer. Figure (a) seem very fitted for gradient descent while (b) shows the presence of a bad local minima.
Therefore, no matter the complexity of the network, if a gradient based algorithm is chosen for the training, it is obvious that the choice of the initial parameter plays a very important role on the final accuracy, even if the depth of the network is expected to reduce the difference between local minima [4].
Another issue that can arise while training the network is the possibility that the iterated set of parameters $p^{(k)}$ verifies $J(p^{(k-1)}) > J(p^{(k)}$ but $\Delta\Psi_t(x, p^{(k-1)}) \geq \Delta\Psi_t(x, p^{(k)})$. Figure 3.3 shows such a behavior, which could be associated with the well known over-learning concept presented in the first chapter.
The following results have to be taken with much caution because of the random initialization of the initial guess of parameters. Indeed, for the previous section, the size of the parameters set p does not depend on the number of training points. Hence, the error surface has the same shape and every test can be pursued using the same initialization p_0, leading to the same local, or global, minima when minimizing. However, when changing the complexity of the network, width or depth, the number of parameters increase accordingly, leading to more complex error surface and thus to different minimas. To try minimizing this issue, averages can be taken for different (30) initializations.

These difficulties have been investigating for feedforward deep neural network in [8] and they can explain the lack of conclusion for this investigation on the network complexity.

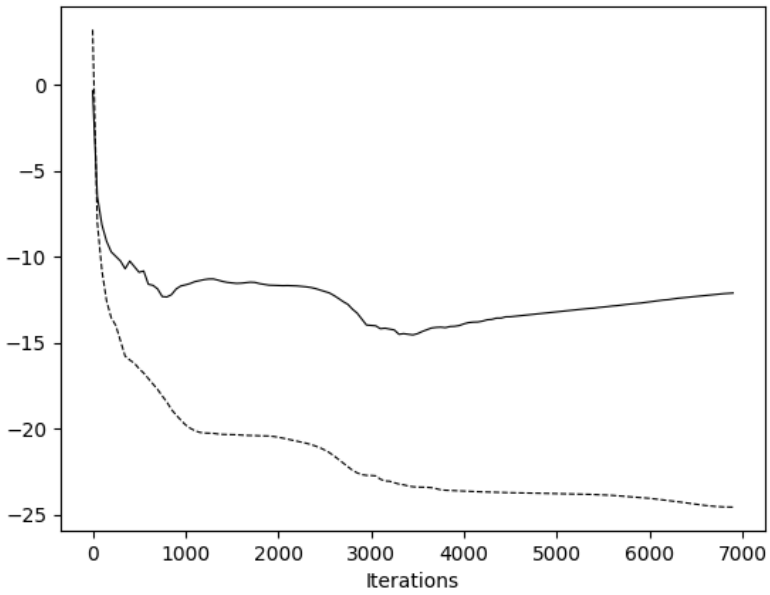


Figure 3.3: Example of bad effects of over-learning to the loss function (dashed line) on the deviation (full line).

3.2.1 On the width of single hidden layered networks

Similarly as for the size of the training set, it has been presented results on the convergence rate of the estimation error of feedforward networks with a single hidden layer (1.1.3). This section will be devoted to show if that results are verified for the method of interest here.
As a reminder, it is expected the network accuracy, with respect to any of the L^p norms, $p \in [1, \infty)$, to decrease as $O(q^{-1/2})$, up to a constant depending on the targeted function's Fourier transform, where q is the number of units (or width) of the hidden layer. As for the size of the training set, for readability purpose, the logarithm of the error will sometimes be taken, which is expected to follow a growth of:

$$\log\left(\|\Psi_t - \Psi\|_{L^2}\right) = O\Big(-\frac{1}{2}\log q\Big)$$

Table 3.3 shows the maximum deviation, the L^2 error of the trained trial function and the value of the loss function for the first ODE problem (equation 2.4.1), trained with a training set of size $n = 10$.

As can be seen, one hidden unit is not enough to catch the function complexity. For the rest, even though the accuracy starts increasing for small q, it will not keep growing after $q = 8$. This can be explained by the increased size of the problem (more parameters) and therefore of the complexity of error surface (more local min-

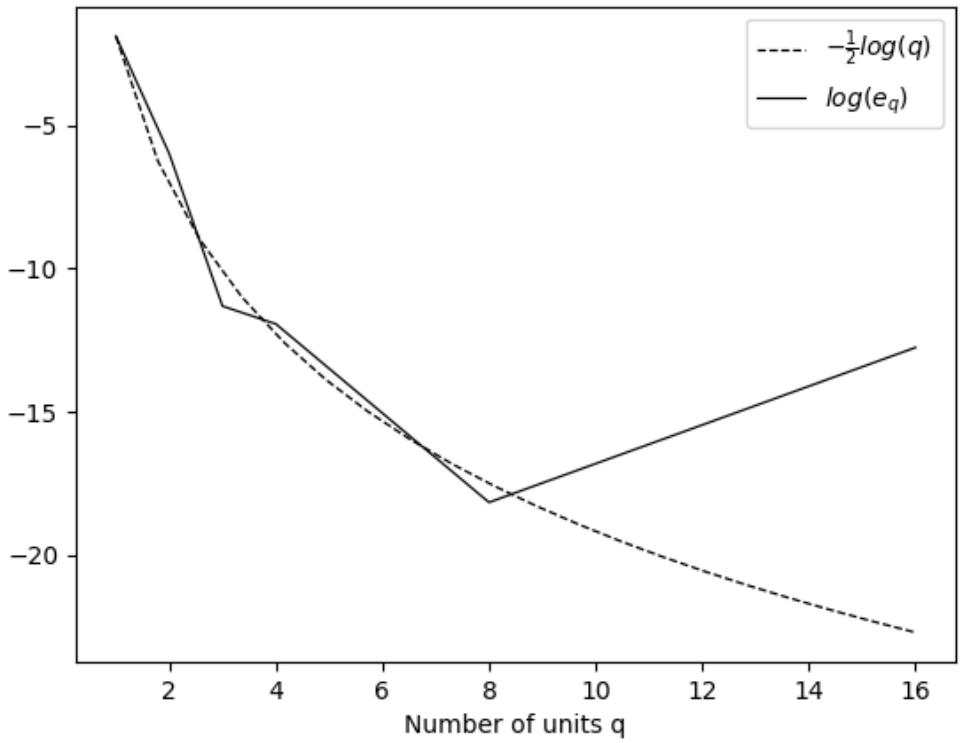


Figure 3.4: $\log\left(\|\Psi_t - \Psi\|_{L^2}\right)$ along q.

q	$\|\Psi_t - \Psi\|_{L^2}$	$\Delta\Psi_t$	$J(p^*)$
1	0.15	0.21	2.46
2	0.0025	0.0048	0.003
3	1.22×10^{-5}	2.45×10^{-5}	3.96×10^{-8}
4	6.56×10^{-6}	1.25×10^{-5}	3.05×10^{-10}
8	9.13×10^{-7}	1.77×10^{-6}	5.15×10^{-10}
16	2.88×10^{-6}	5.37×10^{-6}	2.54×10^{-10}
32	5.25×10^{-6}	9.55×10^{-6}	9.3×10^{-11}
64	5.67×10^{-6}	1.08×10^{-5}	2×10^{-9}
128	2.67×10^{-6}	4.8×10^{-6}	1.17×10^{-11}

Table 3.3: Results of training for different number q of hidden units, problem 1. Training set of ten equidistant points.

imas). For the first small values of q, the logarithm of the error follows a growth similar to the expected $-\frac{1}{2}\log(q)$, however for bigger q, the behavior is more chaotic (Figure 3.4).

Table 3.4 presents the accuracy of the method for the second order initial value conditions problem 2.3a for different width of the network. The figures are the average taken over 30 tests, each with a different initialization p_0. It shows that, as the target function $G(x, \Psi, \Psi', \Psi'')$ is more complex, the network needs more units ($q > 4$) to catch its complexity than for first order equations ($q > 2$). However, the error is not constantly decreasing as q grows.

3.2.2 On the number of layers

This section will be dedicated to the study of the solution's behavior when the number of layers in the network is increased.
For simplicity purpose, the number of units in each hidden layer will be the same, set to $q = 10$. Also the training will be performed using ten equidistant points $a = x_0 < x_2 < ... < x_9 = b$.

The first thing to be highlighted is that the number of unknowns n when the number of layers grows is only increasing linearly:

$$n(K) = K(q^2 + q) + q(d + 1 - q)$$

with K the number of hidden layers and q the width of every layer.

q	$\|\Psi_t - \Psi\|_{L^2}$	$\Delta\Psi_t$	$J(p^*)$
1	4.40×10^{-2}	3.99×10^{-1}	5.17
2	1.97×10^{-2}	1.81×10^{-1}	1.65
4	3.46×10^{-2}	7.94×10^{-2}	7.07×10^{-5}
8	8.40×10^{-5}	1.61×10^{-4}	2.17×10^{-10}
16	1.89×10^{-3}	3.61×10^{3}	2.76×10^{-11}
32	3.40×10^{-3}	6.42×10^{-3}	2.07×10^{-19}

Table 3.4: Effect of the width of the network on the trial solution for second-order initial value problem. Training completed on ten equidistant points.

K	$\|\Psi_t - \Psi\|_{L^2}$	$\Delta\Psi_t$	$J(p^*)$
1	2.46×10^{-3}	1.28×10^{-5}	3.5×10^{-9}
2	1.08×10^{-3}	2.33×10^{-6}	6.84×10^{-12}
4	1.04×10^{-3}	2.2×10^{-6}	1.19×10^{-11}
8	2.7×10^{-3}	1.44×10^{-5}	8.98×10^{-16}
16	3.66×10^{-3}	2.76×10^{-5}	5.41×10^{-9}
32	7.14×10^{-3}	6.02×10^{-5}	1.89×10^{-7}

Table 3.5: Quality of the approximation for different depths of the network.

Although the growth of the number of unknowns is not too important, as for the width, using deeper networks increases considerably the difficulty to train the network, that is to say, find a good minimum of the loss function that gives correct accuracy on the trial function. As it was observed by others ([3]), taking more complex networks is not necessarily of much interest since smaller nets can achieve the desired accuracy. Also it involves more computational complexity and eventually, the risk of over fitting.
Table 3.5 shows the accuracy of the approximation $\Psi_t(x, p*)$ for the two usual norms and how well it satisfies the equation $J(p*)$ for different networks of depth (number of hidden layers) K, for a first order ordinary equation.

This shows that a deeper network seems unnecessary as it only increases the cost of the training. However, further research could show that using deeper nets will, in probability, decrease the chances of the algorithm to end on a bad local minimum.

3.3 On the activation function

As presented before, one can choose in a wide variety of functions to construct a network. However, depending on the problem at hand and the training method, some can be more or less suitable, and some might not even be usable. While it is usual to need the first derivative of the activation function to be continuous in order to use gradient based method (typically, back propagation), solving ODE using this method needs the computation of higher derivatives of these functions. Indeed by construction of the loss function to be minimized, the derivatives of the trial solution are needed and hence, the derivatives of the neural network itself with respect to the input. While this can be analytically computed, it reduces the choice on the activation function to function $\sigma \in C^k$ where k is wanted higher (problems with Neumann boundary conditions) or equal to the order of the differential equation.
This study only made use of activation function in C^∞: hyperbolic tangent, logistic function, Soft Plus and ISRU presented in the section 1.1.1 as well as a linear function $f(x) = \alpha x + \beta$. The parameters of the two latest were arbitrarily chosen as to fit the sigmoid function's slope.

Numerical experiments did not show any difference in accuracy between those functions except for the linear one which seems not appropriate for this method. Indeed after only a few iterations the solver reaches a minimum of poor accuracy. It still reaches a reasonable deviation $\Delta\Psi \approx 10^{-2}$ in a couple of second at a minimum of the loss function $J(p^*) \approx 10^{-2}$.

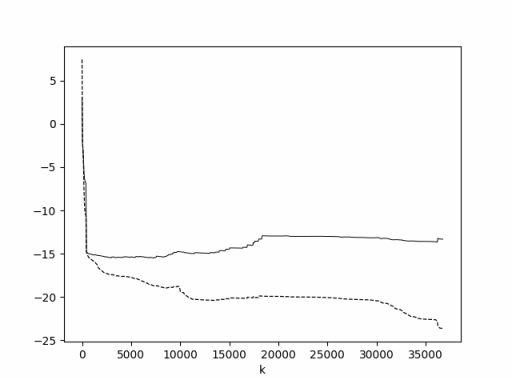
Activation	$\|\Psi_t - \Psi\|_{L^2}$	$\Delta\Psi_t$	$J(p^*)$	Time (s)	Iterations
Sigmoid	3.01×10^{-6}	5.57×10^{-6}	6.86×10^{-10}	755	20722
tanh	2.2×10^{-6}	4×10^{-6}	2.34×10^{-10}	510	21415
ISRU	2.50×10^{-6}	4.59×10^{-6}	2.57×10^{-10}	558	17958
Soft Plus	3.15×10^{-6}	5.85×10^{-6}	8.2×10^{-10}	502	14897
Linear	7.69×10^{-3}	1.06×10^{-2}	2.21×10^{-2}	2	95

Table 3.6: Average performances of the network training on two different ODE problems (first-order and initial value problem).

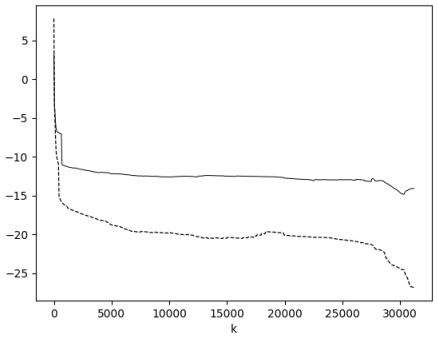
Table 3.6 shows the average performance of the method to solve the two first-order ODE's (equations 2.4.1 and 2.4.2). It appears that the choice of activation function does not bear much importance for such shallow neural network with a single hidden layer. Indeed, except the linear activation, comes to the same order of precision both on the solution and the loss function, and so in a similar time, with a slightly longer duration with the sigmoid function and a with a comparable

computational cost.

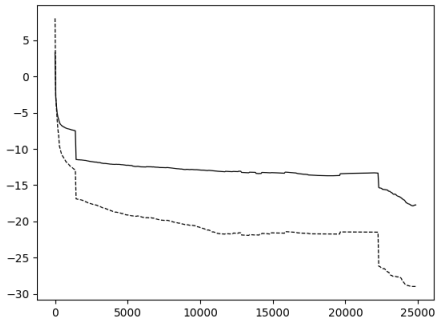
Since a small deviation is of greater importance in engineering problems, this issue could be solved by adding a criterion to the minimization, making the algorithm stop when the deviation starts increasing. However, it is not mathematically proven that this would give the highest accuracy possible (in term of deviation).



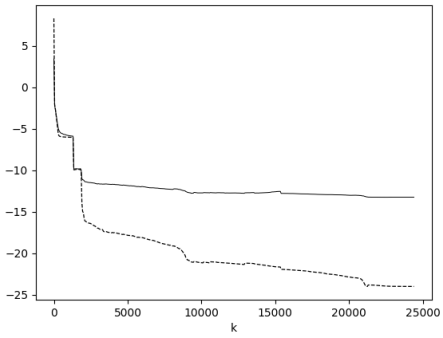
(a) Sigmoid. $\min_k e(p^k) = 1.86 \times 10^{-7}$, for $k = 7300$.



(b) Hyperbolic tangent. $\min_k e(p^k) = 3.55 \times 10^{-7}$ for $k = 30300$.



(c) ISRU. $\min_k e(p^k) = 1.72 \times 10^{-8}$ for $k = 24650$.



(d) Soft Plus. $\min_k e(p^k) = 1.71 \times 10^{-6}$ for $k = 21950$.

Figure 3.5: Error $log(e(p^k))$ at iteration k (full line) compared with $log(J(p^k))$ (dashed line) for different activation functions.

Figure 3.5 displays the training curves ($J(p^{(k)})$) and the maximum deviation curve at step k for the different activation functions. It shows the possibility that using sigmoid units could imply a bad over fitting effect as the deviation starts growing at the end of the training. Also, it is possible that such activation function is more accurate if the training is stopped at the appropriate step. Using sigmoid units as shown to have reached an accuracy at $k = 5000$ that the others types don't reach.

Therefore, it is an appropriate candidate in the choice of activation function. However, these conclusions are speculations and need to be investigated closer.

Chapter 4

Conclusion and future work

The neural network method investigated in this paper shows intriguing features. It had already been shown in the original paper from Lagaris that the computed solution can be of high accuracy. However, numerical experiments conducted here show that the neural network method has strong limitations due to the under-determined nature of the minimization problem. Indeed, it has not been possible during this work to show the role, if there is one, played by the complexity of the network because of that issue. It can be expected that the accuracy increases as the network get wider, with a possible convergence rate of $O(q^{-1/2})$. However, this rate would only holds for network which training reaches a global minimum of the loss function, while such a feat is only theoretical.
Also, it has been shown that, according to the theory [15], the choice of activation function does not fundamentally change the capacities of the constructed trial solution to solve the differential equation, as long as the choice is made on a non-linear and smooth enough function. The linearity of the activation function has proved to restrain the network complexity, in such a way that the trial solution is of poor accuracy.
The main result of this work is that the accuracy of the network can, in a certain measure, be controlled just by sampling more training points on the equation domain. A larger training set has the property of increasing the precision while not implying more parameters in the problem, and at smaller computational cost. It is proposed that this accuracy follows a convergence rate of $O\left(e^{-\sqrt{n}}\right)$.
However, these investigations highlighted the difficulty of numerical analyses of the neural network solver, due to the probabilistic and complex nature of the training problem.

Future work can be dedicated in trying to find a mean to determine the best architecture (width, depth) for the network, in the sense that it would maximize the accuracy of the trial solution. Also, finding ways to tackle the issue of complex error surfaces misleading optimization algorithm to non-optimal local minimum would permit an easier investigation of method based on feedforward neural networks. In

particular, the choice of the initial guess of the parameter p can be an interesting starting point. It has been proposed optimal range for the initial parameters for different activation functions who could lead to better training. Also, different networks are expected to have different qualities that could be more fitted for certain differential equations.
Although the simplicity of feedforward networks is attractive, better accuracy could be obtained using more complex networks. The zoo of neural networks types is vast and networks such as Radial Basis Networks, Recurrent Networks, Convolutional Networks are more and more used to solve complex tasks and therefore could be useful for differential equations.
Since the training of neural network has a probabilistic nature, efforts should be made to find the type, architecture, grid, used for the network in order to maximize the chances to avoid "bad" minimums.

Bibliography

[1] A. R. Barron, Approximation and estimation bounds for artificial neural networks, Machine Learning, vol. 14, pp. 115-133, 1994.

[2] A.R Barron, Universal approximation bounds for superpositions of a sigmoidal function, IEEE Transactions on Information Theory, 39, 930-945.

[3] Solving differential equations using neural networks, M. M. Chiaramonte and M. Kiener, 2013

[4] A. Choromanska et al., The Loss Surfaces of Multilayer Networks, Proceedings of the 18th International Conference on Artificial Intelligence and Statistics (AISTATS), Vol. 38, 2015.

[5] Cybenko, G.V. . Approximation by Superpositions of a Sigmoidal function. In van Schuppen, Jan H. Mathematics of Control, Signals, and Systems. Springer International. pp. 303314, 1989.

[6] R. Fletcher, *Practical Methods of Optimization*, New York: Wiley, 1987.

[7] K. Funahashi, On the approximate realization of continuous mappings by neural networks, Neural Networks, 2, 183-192.

[8] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, Journal of machine learning research, January 2010.

[9] J. Hertz, A. Krogh, R. G. Palmer, *Introduction to the theory of neural computation*, CRC Press, 1991.

[10] Hornick K., Stinchcombe M., White H., Multilayer feedforward networks are universal approximators, Neural Networks, Vol. 2, pp. 359-366, 1989.

[11] K. Hornik, Approximation Capabilities of Multilayer Feedforward Networks, Neural Networks, vol. 4, pp. 251-257, 1991.

[12] Lagaris I. E., Likas A., Fotiadis D. I., Artificial neural networks for solving ordinary and partial differential equations, IEEE Transactions on Neural Networks, 9 (5) (1998) 987-1000.

[13] I. E. Lagaris, A. Likas, D. G. Papageorgiou, Neural Network Methods for Boundary Value Problems with Irregular Boundaries, IEEE Transactions on Neural Networks, vol. 11, pp. 1041-1049, 2000.

[14] H. Lee, I.S Kang, Neural algorithm for solving differential equations, Journal of computational physics, Vol. 91, pp. 110-131 , 1990.

[15] M. Leshno et al., Multilayer Feedforward Networks With a Nonpolynomial Activation Function Can Approximate Any Function, Neural Networks, Vol. 6, pp. 861-867, 1993.

[16] D. Maclaurin, D. Duvenaud, M. Johnson, J. Townsend, github.com/HIPS/autograd.

[17] A.J. Maede, A.A Fernandez, The Numerical Solution of Linear Ordinary Differential Equations by Feedforward Neural Networks, Mathl. Comput. Modelling Vol. 19, No. 12, pp. 1-25, 1994.

[18] Meade Jr A.J., Fernandez A.A., Solution of nonlinear ordinary differential equations by feedforward neural networks, Mathematical and Computer Modelling, 20 (9) (1994) 19-44.

[19] N. Mai-Duy, Solving High Order Ordinary Differential Equations with Radial Basis Function Networks, Int. J. Numer. Meth. Engng, 2004.

[20] D.F. McCaffray, A.R. Gallant, Convergence rates for single hidden layer feedforward neural networks, Neural Networks, vol. 7, pp. 147-158, 1994.

[21] M. Stinchcombe, H. White, Multilayer feedforward networks can learn arbitrary mappings: Connectionist nonparametric regression with automatic and semi-automatic determination of network complexity, Neural Networks, vol. 3, pp. 535-550.

[22] D. Sussillo, L.F Abbott, Random Walk Initialization For Training Very Deep Feedforward Neural Networks, arXiv:1412.6558.

[23] N. Yadav, A. Yadav, M. Kumar, *An introduction to Neural Network Methods for Differential Equations.*, Springer.