FEDERAL UNIVERSITY OF SANTA CATARINA
TECHNOLOGY CENTER
DEPARTMENT OF AUTOMATION AND SYSTEMS ENGINEERING
UNDERGRADUATE PROGRAM IN CONTROL AND AUTOMATION ENGINEERING

Bruno Machado Pacheco

Physics-Informed Deep Equilibrium Models for Solving ODEs

Florianópolis
2022

Bruno Machado Pacheco

Physics-Informed Deep Equilibrium Models for Solving ODEs

Final report of the subject DAS5511 (Course Final Project) as a Concluding Dissertation of the Undergraduate Program in Control and Automation Engineering of the Federal University of Santa Catarina.
Supervisor: Prof. Eduardo Camponogara, Ph.D.

Florianópolis
2022

Ficha de identificação da obra

A ficha de identificação é elaborada pelo próprio autor.

Orientações em:

http://portalbu.ufsc.br/ficha

Bruno Machado Pacheco

Physics-Informed Deep Equilibrium Models for Solving ODEs

This dissertation was evaluated in the context of the subject DAS5511 (Course Final Project) and approved in its final form by the Undergraduate Program in Control and Automation Engineering

Florianópolis, July 2nd, 2022.

Prof. Hector Bessa Silveira, Dr.
Course Coordinator

Examining Board:

Prof. Eduardo Camponogara, Dr.
Advisor and Supervisor
UFSC/CTC/DAS

Prof. Danilo Silva, Dr.
Evaluator
UFSC/CTC/EEL

Prof. Marcelo de Lellis Costa de Oliveira, Dr.
Board President
UFSC/CTC/DAS

This work is dedicated to my father.

ACKNOWLEDGEMENTS

First and foremost, this work was made possible thanks to the guidance of Prof. Eduardo Camponogara, who accompanied me from the very beginning. Also thanks to my father and my girlfriend, who supported me in many different ways. Finally, I would like to thank all the masters who inspired me and helped me develop the faculties necessary for this work, specially Prof. João Goulart, Prof. Ivan Pontual, Prof. Gustavo Donatelli, Prof. Eduardo Camponogara (again), Jonathan Krauß, and Prof. Danilo Silva.

“The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work - that is correctly to describe phenomena from a reasonably wide area. Furthermore, it must satisfy certain esthetic criteria - that is, in relation to how much it describes, it must be rather simple.”

John von Neumann *apud* Leary (1955)

ABSTRACT

Physics-informed neural networks (PINNs) and deep equilibrium models (DEQs) are novel approaches that approximate deep learning's representational power to applications with realistic requirements such as robustness, data scarcity and explainability. PINNs propose an efficient way to train neural networks to model physical phenomena. DEQs are a new model architecture that can provide more representational power with fewer parameters. This work aims to study both and apply them to solve initial-value problems (IVPs) of ordinary differential equations (ODEs), in an approach coined physics-informed deep equilibrium model (PIDEQ). We implement the proposed approach and test it, analyzing the impacts of the multiple hyperparameters in the approximate solution of the Van der Pol oscillator. Our results show that indeed PIDEQ models are able to solve IVPs, providing approximate solutions with small errors.

Keywords: Physics-Informed Neural Network; Deep Equilibrium Model; Implict Model; Initial-Value Problem; Deep Learning.

RESUMO

Redes neurais para solução de problemas físicos, denominadas *physics-informed neural networks* (PINNs), e modelos profundos de equilíbrio (do inglês, DEQs), são contribuições recentes que facilitam o uso de modelos de aprendizagem profunda, conhecidos pela capacidade representativa, de aplicações com requisitos realistas de robustez, explicabilidade e escassez de dados. PINNs se mostraram uma forma eficiente de treinar redes neurais para modelar fenômenos físicos. DEQs, por outro lado, empregam uma nova arquitetura que promete mais capacidade de representação com menos parâmetros. Este trabalho consiste em um estudo de ambos, além de uma aplicação que combina-os para resolver problemas de valor inicial de equações diferenciais ordinárias, com um modelo chamado PIDEQ. A abordagem proposta para resolver esse tipo de problema foi implementada e testada utilizando o oscilador de Van der Pol, com uma análise de impacto dos seus diferentes hiperparâmetros. Os resultados mostram que, de fato, é possível treinar um PIDEQ para resolver o problema proposto, gerando soluções aproximadas com baixo erro.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

DEQ	deep equilibrium model
IAE	integral of absolute error
IVP	initial-value problem
ODE	ordinary differential equation
PIDEQ	physics-informed deep equilibrium model
PINN	physics-informed neural network
RK4	classic Runge-Kutta method

LIST OF SYMBOLS

$\boldsymbol{\theta}$	Parameter vector
σ	Sigmoid function
γ	Learning rate

CONTENTS

1 INTRODUCTION

The numerous successful applications of deep learning in the most varied industries (CIREGAN; MEIER; SCHMIDHUBER, 2012; KRIZHEVSKY; SUTSKEVER; HINTON, Geoffrey E, 2012; SILVER et al., 2016) have established it as a cornerstone of artificial intelligence. With such a fame, one could easily suppose that it can be a powerful tool to solve a very common problem in the study of dynamical systems: the initial-value problem (IVP) of differential equations.

Yet, deep learning models gather knowledge by learning from experience, which gives them a data-driven nature. Unfortunately, gathering sufficient data (samples, records, experiment history) from real-life dynamical systems can be costly, as it may require an assortment of measurement devices and expensive material to stimulate and observe the phenomena of interest. This makes training deep learning models, in such context, quite challenging.

Raissi, Perdikaris, and Karniadakis (2019) proposed to optimize the dynamics of the model instead of its outputs. This way, one can train a deep learning model to approximate the dynamics of a system, given that an accurate description of these dynamics is known (e.g., in the form of differential equations). Therefore, only minimal data is necessary to cover the initial and boundary conditions.

At the same time, Bai, Zico Kolter, and Koltun (2019) and Ghaoui et al. (2019) proposed a novel approach to deep learning: to define the output of a model implicitly, in the form of a solution to an equilibrium equation. A model defined this way is effectively an infinite-depth network (BAI; ZICO KOLTER; KOLTUN, 2019) with residual connections(HE et al., 2016), which results in a great representational power with a small amount of parameters. This reduced set of parameters improves the model's explainability, which in turn helps with deriving causality and robustness, all of which are desirable features in safety-critical applications (e.g., in the automation industry).

Even though successful applications of deep equilibrium model (DEQ)s have already appeared in the literature (BAI; KOLTUN; KOLTER, 2020), it is still unknown, for the majority of deep learning applications, the problems in which DEQs can succeed over traditional models. Since to generate the output of an equilibrium model one has to solve an equilibrium equation, these models present an iterative behavior, hence similar to commonly used differential equation solvers such as the Runge-Kutta algorithm. Therefore, given the novelty of the model and its iterative behavior, the question this work aims to answer is: can a DEQ be trained to solve an IVP of an ordinary differential equation (ODE) in a physics-informed approach?

1.1 OBJECTIVES

This work aims to study, implement and validate the use of Deep Equilibrium models as effective and efficient solvers for IVPs, making use of physics-informed training. An efficient model is twofold: it does not require costly (many samples or hard-to-acquire samples) data for training, and it does not require expensive hardware to generate fast results. At the same time, an effective model is one that can provide accurate results on a large domain, i.e., it can solve a given ODE for a long range of its independent variable.

More specifically, the work described in here aims to:

1. Implement a Deep Equilibrium model following the design proposed by Bai, Zico Kolter, and Koltun (2019);
2. Design and implement a physics-informed neural network (PINN) training algorithm suitable for DEQs, building on top of the work by Raissi, Perdikaris, and Karniadakis (2019);
3. Evaluate the physics-informed deep equilibrium model in the task of solving an IVP.

With these steps, it is expected that a solid conclusion will be achieved on the suitability of DEQs, a state-of-the-art deep learning architecture, on this novel application.

In the first two chapters below, a brief introduction to ODEs (Ch. 2) and deep learning (Ch. 3) is presented, aiming to lay a foundation and establish a concise reference. The following chapters are dedicated to the novel contributions of PINNs (Ch. 4) and DEQs (Ch. 5), which are central for the aim of this work. Finally, Chapter 6 presents the experimental results of a physics-informed DEQ trained to solve an IVP. These results are discussed in Chapter 7 together with an outlook of future work.

2 ORDINARY DIFFERENTIAL EQUATIONS

Differential equations were first defined upon the invention of calculus. In fact, they were studied by both Newton and Leibniz, and by many of the greatest mathematicians such as Euler and Bernoulli. Even when limited to ODEs, it would be unreasonable to assume that one chapter of a bachelor thesis could cover what many authors have dedicated entire books to communicate. Therefore, it is assumed that the reader has a basic (undergraduate level) understanding of the theory of differential equations[1].

This first chapter is, then, a short review of ODEs and how to solve them numerically. This way, the focus is on reviewing the fundamentals and establishing a concise notation for the following chapters. Furthermore, the approach to numerically solve ODEs and the Van der Pol oscillator are also introduced, as they are essential for the experiments described in Chapter 6.

2.1 INTRODUCTION AND DEFINITIONS

A differential equation can be seen as a description of the relationship between unknown quantities and their rates of change. Because of this broader definition and direct relationship to applications, differential equations arise naturally in many fields of applied sciences (HAIRER; WANNER; NØRSETT, 1993), as it is often the case that the rate of change of a certain quantity of interest is related to the rate of change of other quantities. A classical example of a differential equation is Newton's second law of motion,

$$F\left(x\left(t\right)\right) = M\frac{d^2}{dt^2}x(t), \tag{2.1}$$

in which $x\left(t\right)$ is the position of an object at time t, M is its mass, and $F(x)$ is the force under which the object is at a given position. This example highlights one of the great powers of the differential equations, which is to describe the dynamics of a certain phenomenon without explicitly defining it.

For a more tangible definition, any equation that contains the derivatives of (at least) one unknown function with respect to (at least) one independent variable is called a *differential equation* (ZILL, 2013). A differential equation that involves only *ordinary* derivatives, that is, only derivatives of functions with respect to a single variable, (such as Equation (2.1)) is called an ODE. ODEs can be represented in the normal form

$$\frac{d^n\boldsymbol{x}(t)}{dt^n} = \mathcal{N}'\left(t, \boldsymbol{x}\left(t\right), \frac{d\boldsymbol{x}(t)}{dt}, \ldots, \frac{d^{n-1}\boldsymbol{x}(t)}{dt^{n-1}}\right).$$

in which $\boldsymbol{x} : \mathbb{R} \to \mathbb{R}^{m'}$, $\mathcal{N}' : \mathbb{R} \times \mathbb{R}^{nm'} \to \mathbb{R}^{m'}$ are continuous functions, $\frac{d^n\boldsymbol{x}(t)}{dt^n}$ is the vector containing the derivatives (with respect to t) of all $\boldsymbol{x}\left(t\right)$ components, and n is the

[1] Otherwise, the books of Zill (2013) and Simmons (2017) are great sources.

[1] I.e., the first (and only) row-vector of the Jacobian of $\boldsymbol{x}\left(t\right)$.

order of the highest derivative in the equation and is commonly referred to as the *order of the differential equation*.

Any function $\boldsymbol{\phi} : I \subset \mathbb{R} \to \mathbb{R}^{m'}$, I an interval, is said to be a *solution* of an n-th order ODE if it is n-times differentiable on I and satisfies

$$\frac{d^n \boldsymbol{\phi}(t)}{dt^n} = \mathcal{N}' \left(t, \boldsymbol{\phi}\left(t\right), \frac{d\boldsymbol{\phi}(t)}{dt}, \ldots, \frac{d^{n-1}\boldsymbol{\phi}(t)}{dt^{n-1}} \right), \forall t \in I.$$

Note, however, that the solutions are not necessarily unique. As an example, given Newton's second law of motion, as shown in equation (2.1), with constant force ($F(x) = C$), then it is easy to see that any second-order polynomial of the form:

$$\phi\left(t\right) = \frac{C}{2M} t^2 + a_1 t + a_0,\ a_0, a_1 \in \mathbb{R}$$

is a solution.

Now, given an n-th order ODE, let $\boldsymbol{y}_1\left(t\right) = \boldsymbol{x}\left(t\right), \boldsymbol{y}_2\left(t\right) = \frac{d\boldsymbol{x}(t)}{dt}, \ldots, \boldsymbol{y}_n\left(t\right) = \frac{d^{n-1}\boldsymbol{x}(t)}{dt^{n-1}}$. Note that we can now write the following *system* of first-order differential equations:

$$\begin{aligned}
\frac{d\boldsymbol{y_1}\left(t\right)}{dt} &= \frac{d\boldsymbol{x}\left(t\right)}{dt} = \boldsymbol{y}_2\left(t\right) \\
&\vdots \\
\frac{d\boldsymbol{y_{n-1}}\left(t\right)}{dt} &= \frac{d^{n-1}\boldsymbol{x}\left(t\right)}{dt^{n-1}} = \boldsymbol{y}_n\left(t\right) \\
\frac{d\boldsymbol{y_n}\left(t\right)}{dt} &= \frac{d^{n}\boldsymbol{x}\left(t\right)}{dt^{n}} = \mathcal{N}' \left(t, \boldsymbol{x}\left(t\right), \frac{d\boldsymbol{x}(t)}{dt}, \ldots, \frac{d^{n-1}\boldsymbol{x}(t)}{dt^{n-1}} \right) = \mathcal{N}'\left(t, \boldsymbol{y}_1\left(t\right), \ldots, \boldsymbol{y}_n\left(t\right)\right).
\end{aligned}$$

Then, for $m = nm'$, define $\boldsymbol{y} : \mathbb{R} \to \mathbb{R}^m$ such that

$$\boldsymbol{y}\left(t\right) = \begin{bmatrix} \boldsymbol{y}_1\left(t\right) \\ \vdots \\ \boldsymbol{y}_n\left(t\right) \end{bmatrix},$$

and $\mathcal{N} : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$ such that

$$\mathcal{N}\left(t, \boldsymbol{y}\left(t\right)\right) = \begin{bmatrix} \boldsymbol{y}_2 \\ \vdots \\ \boldsymbol{y}_n \\ \mathcal{N}'\left(t, \boldsymbol{y}_1\left(t\right), \ldots, \boldsymbol{y}_n\left(t\right)\right) \end{bmatrix}.$$

With this, it is easy to see that the first-order ODE

$$\frac{d\boldsymbol{y}\left(t\right)}{dt} = \mathcal{N}\left(t, \boldsymbol{y}\left(t\right)\right) \tag{2.2}$$

is equivalent to the original n-th order ODE, that is, given a solution for one of the equations, one can trivially derive the solution for the other. In other works, *any n-th order ODE can be written as a first-order ODE with n times the number of variables*.

As an example, let us take once again Newton's second law (Eq. (2.1)) with constant force $F(x) = C$. This equation in the normal form can be written

$$\frac{d^2x\,(t)}{dt^2} = \frac{C}{M}.$$

Now, define $\boldsymbol{y} : \mathbb{R} \to \mathbb{R}^2$, $\boldsymbol{y}(t) = (y_1\,(t)\,, y_2\,(t))$ such that $y_1(t) = x(t)$ and $y_2(t) = \frac{dx(t)}{dt}$. Thus, we can write the first order ODE

$$\frac{d\boldsymbol{y}\,(t)}{dt} = \begin{bmatrix} \frac{dy_1(t)}{dt} \\ \frac{dy_2(t)}{dt} \end{bmatrix} = \begin{bmatrix} y_2(t) \\ \frac{C}{M} \end{bmatrix}.$$

The solution to this equation is of the form

$$\begin{bmatrix} \frac{C}{2M}t^2 + a_1 t + a_0 \\ \frac{C}{M}t + a_1 \end{bmatrix},\ a_1, a_0 \in \mathbb{R},$$

which, from the definition of $\boldsymbol{y}(t)$, is in accord with the solution for the original ODE.

2.2 INITIAL VALUE PROBLEMS

It is a common problem to find an explicit definition of the unknown functions in a differential equation. For ODEs, it is often the case that, if one knows side conditions on the unknown function, then a solution exists and it is unique. More precisely, for an ODE defined as in (2.2), conditions of the form

$$\boldsymbol{y}\,(t_0) = \boldsymbol{y}_0$$

guarantee the existence and uniqueness of an (analytic) solution on the interval $I \subset \mathbb{R}$, $t_0 \in I$, if $\mathcal{N}$ is an analytic function[2] in $I \times R$, $R \subset \mathbb{R}^m$ with $\boldsymbol{y}_0 \in R$, which is the case for many practical applications (ISERLES, 2008). The problem of finding the solution given conditions as above is called the IVP.

IVPs show up frequently when the current (or past) value of the function is known and a future value is desired. Recalling Newton's second law example, suppose that, besides $F(x) = C$, it is also known that the object lies still at $t = 0$, i.e.,

$$y(0) = 0,\ \frac{dy(0)}{dt} = 0,$$

and we are interested in the solution on the interval $I = [0, T]$, for $T > 0$. Then, the only solution is

$$\phi\,(t) = \frac{C}{2M}t^2.$$

[2] A function is said analytic if and only if its Taylor series expansion converges in the entirety of its domain.

2.3 NUMERICAL SOLVERS

2.3.1 Euler's Method

In an IVP, one has the value of $\boldsymbol{y}(t)$ at a given instant in time t_0 and can find the slope at that time through $\mathcal{N}$. Therefore, it is natural that an efficient numerical solution is derived from the linear approximation of $\boldsymbol{y}(t)$ at t_0. In practice, this means to approximate $\mathcal{N}(t, \boldsymbol{y}(t)) \approx \mathcal{N}(t_0, \boldsymbol{y}_0), \forall t \in I^h$, where I^h is a neighborhood of t_0 (i.e., $I^h \subset I$ is an interval with length h such that $t_0 \in I^h$). Then, a solution can be approximated as

$$\begin{aligned}\boldsymbol{\phi}(t) &= \boldsymbol{y}(t_0) + \int_{t_0}^{t} \mathcal{N}(\tau, \boldsymbol{y}(\tau))\, d\tau \\ &\approx \boldsymbol{y}(t_0) + (t - t_0)\,\mathcal{N}(t_0, \boldsymbol{y}_0),\ t \in I^h.\end{aligned}$$

Of course, the performance of this naïve approach (for any non-constant $\mathcal{N}$) is satisfactory only for small h. However, one can perform many of such approximations with arbitrarily small h. Let us assume that $I = [t_0, t_N]$[3] and that one desires an approximation of an exact solution $\boldsymbol{\phi}(t)$ on I. Given a sequence $t_0, \ldots, t_i, \ldots, t_N$ such that $t_{i+1} - t_i = h$, $i = 1, \ldots, N$, let us call $\boldsymbol{y}_i$ the approximation of an exact solution at t_i for $i = 1, \ldots, N$, i.e., $\boldsymbol{y}_i \approx \boldsymbol{\phi}(t_i)$. One can use the first order approximation recursively and compute

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\mathcal{N}(t_i, \boldsymbol{y}_i),\ i = 1, \ldots, N,$$

which is as to give one large step, one takes multiple small steps in the same direction. By increasing the number of steps, one reduces the length of each step and, thus, reduces the approximation error $\|\boldsymbol{y}_{i+1} - \boldsymbol{\phi}(t_{i+1})\|$. By increasing the number of steps, one can achieve an arbitrarily low error (ISERLES, 2008). This approach is the *Euler's method*, and besides being the simplest way of effectively solving the IVP, it forms the basis of most of the commonly used advanced numerical solvers for ODEs.

2.3.2 Runge-Kutta Method

Euler's method comes short in that it uses *only* the information from the previous step at each iteration, i.e., it discards the information from $\boldsymbol{y}_{i-1}, \boldsymbol{y}_{i-2}, \ldots$ (when available). The *Runge-Kutta methods*, on the other hand, feed from the rich theory on numerical integration and use multiple approximations of $\boldsymbol{y}(t)$ to take a step. More precisely, the classic Runge-Kutta method (RK4), also known as classic Runge-Kutta, uses 4 approximations of the solution to estimate $\boldsymbol{y}_{i+1}$.

Similar to Euler's method, Runge-Kutta is also an iterative approach, which means that given $\boldsymbol{y}_i$ (an approximation of the solution at t_i) it will compute $\boldsymbol{y}_{i+1}$ (an

[3] If t_0 is not the extrema of the interval, one can just apply the method in both directions.

approximation of the solution at $t_{i+1} = t_i + h$). The intuition of RK4 is to iteratively approximate $\mathcal{N}$ (the slope) for the solution *within* the step (i.e, the interval $[t_i, t_{i+1}]$) and use this information to compute $\boldsymbol{y}_{i+1}$. Let us denote $\boldsymbol{k}_1, \ldots, \boldsymbol{k}_4$ as intermediate approximations of $\mathcal{N}$ such that

$$\begin{aligned}
\boldsymbol{k}_1 &= \mathcal{N}\left(t_i, \boldsymbol{y}_i\right) \\
\boldsymbol{k}_2 &= \mathcal{N}\left(t_i + \frac{h}{2}, \boldsymbol{y}_i + \frac{h}{2}\boldsymbol{k}_1\right) \\
\boldsymbol{k}_3 &= \mathcal{N}\left(t_i + \frac{h}{2}, \boldsymbol{y}_i + \frac{h}{2}\boldsymbol{k}_2\right) \\
\boldsymbol{k}_4 &= \mathcal{N}\left(t_i + h, \boldsymbol{y}_i + h\boldsymbol{k}_3\right).
\end{aligned}$$

Note that each approximation depends on the previous ones. Furthermore, one can see that $\boldsymbol{k}_1$ is the approximation of $\mathcal{N}$ at the beginning of the step (t_i), while $\boldsymbol{k}_2$ and $\boldsymbol{k}_3$ are at the middle ($t_i + \frac{h}{2} = t_{i+1} - \frac{h}{2}$) and $\boldsymbol{k}_4$ is at the end (t_{i+1}). Finally, $\boldsymbol{y}_{i+1}$ is computed taking a weighted average of these approximations as

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\frac{\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4}{6}.$$

Even though Runge-Kutta's approach seems quite simple and magic, it provides a significant error reduction in a still very lean algorithm. In fact, even when one bases around the number of evaluations of the $\mathcal{N}$ function (which tends to be the costlier operations for complex functions), it still outperforms the Euler's method, even with a step size 5 times larger[4]. This combination makes RK4 one of the most used ODE solvers yet. An example can be seen in Figure 1.

2.4 VAN DER POL OSCILLATOR

The Van der Pol oscillator was first described by Van der Pol (1920) on the study of oscillatory behavior in electrical circuits with triodes. Since then, it has been used plenty to model natural phenomena (LUCERO; SCHOENTGEN, 2013; FITZHUGH, 1961; NAGUMO; ARIMOTO; YOSHIZAWA, 1962; CARTWRIGHT et al., 1999). A common representation is as a second-order differential equation

$$\frac{d^2y(t)}{dt^2} = \mu\left(1 - y(t)^2\right)\frac{dy(t)}{dt} + y(t),$$

where $\mu \in \mathbb{R}$ dictates the *dampening* behavior. As has been shown in the previous section, the equation can be written in the form of Equation (2.2) as

$$\frac{d}{dt}\begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix} = \begin{bmatrix} y_2(t) \\ \mu\left(1 - y_1(t)^2\right) y_2(t) - y_1(t) \end{bmatrix}. \tag{2.3}$$

[4] Each iteration of RK4 makes 5 $\mathcal{N}$ function evaluations while Euler's method's makes only one evaluation for iteration, so we can estimate that RK4 is 5 times more expensive than Euler's.

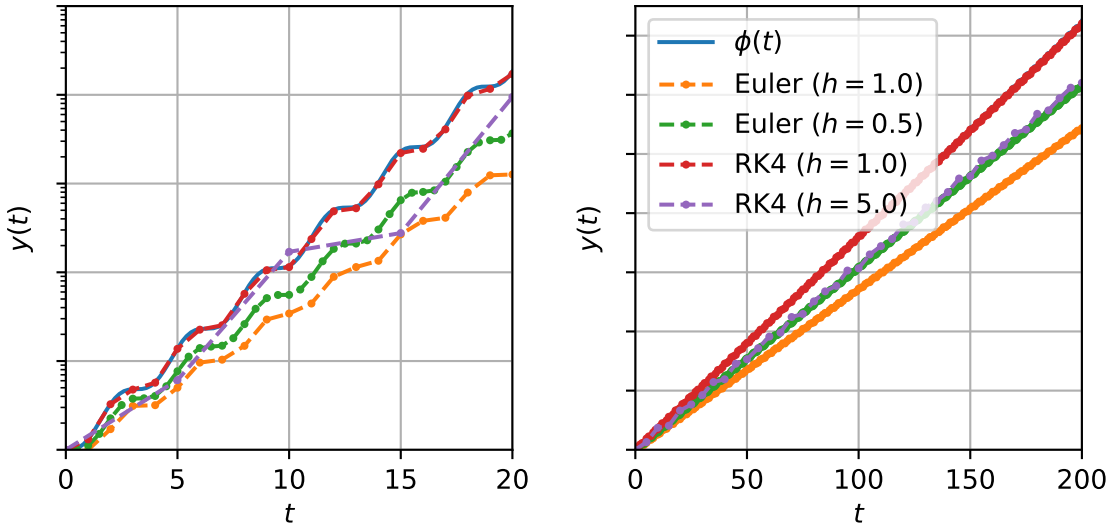
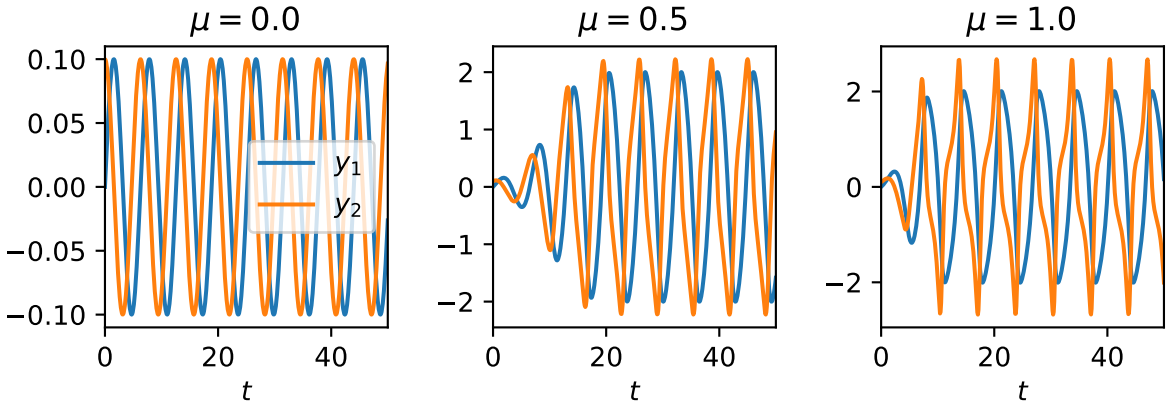


Figure 1 – Comparison of RK4 and Euler's method for $\frac{dy(t)}{dt} = y(t)\sin(t)^2$ with $y(0) = 1$. The analytical solution $\phi(t)$ is shown in the solid line while the solvers with different step sizes are shown in broken lines. The image on the right shows the impact in a long-term run, highlighting the comparison between RK4 with $h = 5$ and Euler's with $h = 1$.

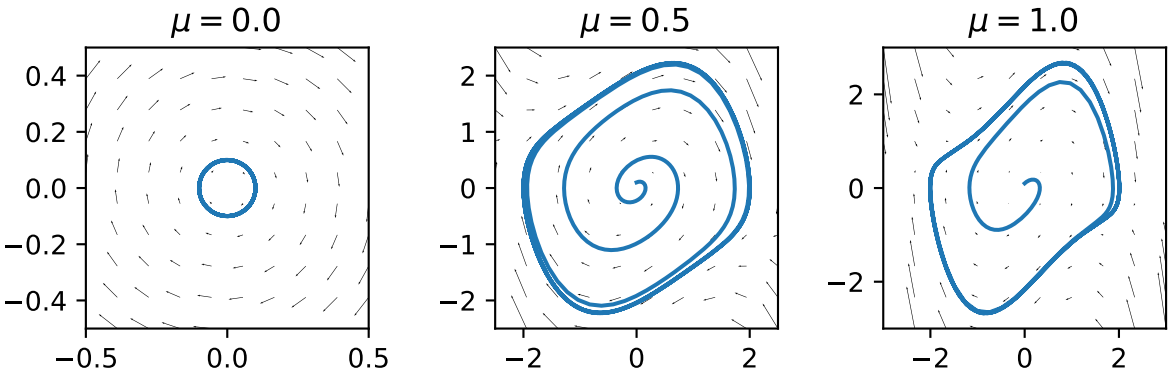
In the example shown in Figure 2, the impact of μ in the non-linearity of the solution is shown.

The Van der Pol oscillator has no analytic solution (PANAYOTOUNAKOS; PANAYOTOUNAKOU; VAKAKIS, 2003), so it requires an accompanying numerical solver to be studied. Nevertheless, it is well known that the oscillator has a unique and stable *limit cycle*[5](GRIMSHAW, 1993). These limit cycles can be seen in the examples shown in Figure 2b.

[5] Loosely defined, a limit cycle can be seen as the image of the solution when it becomes periodic. Refer to Grimshaw (1993) for a more precise definition.



(a) Time evolution



(b) State-space

Figure 2 – Examples of (numerical) solutions for the Van der Pol oscillator. The notation of Equation (2.3) is used. Different values of the dampening parameter are used, but the initial condition is fixed as $y_1(0) = 0$, $y_2(0) = 0.1$. In (a), the evolution of the numerical solution over 50 seconds is shown. In (b), the trajectory of the numerical solution (y_2 in the vertical axis and y_1 in the horizontal axis, i.e., a state-space plot) is shown for the same time horizon.

3 DEEP LEARNING

In this chapter, a brief overview of the key elements of deep learning are presented. Deep Learning is a broad area of knowledge that can be seen (and presented) through many viewpoints, which results in a multitude of notations and few well-established naming conventions. Here, the approach of Goodfellow, Bengio, and Courville (2016) is followed, being altered only to make it consistent with the other chapters. Furthermore, even though physics-informed learning and implicit models can be seen as branches of deep learning, they are introduced in separated chapters for their relevance.

3.1 INTRODUCTION AND DEFINITION

Even though *Deep Learning* may seem like a novel and exciting technology, it has been studied under many different names since the 1940s (GOODFELLOW; BENGIO; COURVILLE, 2016). Deep learning involves composing multiple levels[1] of representation learning. Representation learning, in turn, is about automatically extracting higher level features from the input data (LECUN; BENGIO; HINTON, Geoffrey, 2015; BENGIO; COURVILLE; VINCENT, 2013), aiming to facilitate the extraction of useful information, for example, to classify the data into given categories.

Following these definitions, we can see a deep learning *model* as a function with multiple levels of representations. A simple way of putting this definition into terms would be to define a model with L *layers* of representations as a function D such that, for a given input x,

$$\begin{aligned} z^{[0]} &= x \\ z^{[i]} &= f^{[i]}(z^{[i-1]}),\ i = 1, \ldots, L \\ D(x) &= z^{[L]}. \end{aligned} \tag{3.1}$$

From this notation, it is easy to see that each function $f^{[i]}$ maps the outputs of the previous layer into the input of the next, ideally achieving a higher abstraction level.

In this context, we can see the goal of *learning* as finding functions $f^{[i]}$ that indeed extract the desired information, culminating in the model returning the desired output. A classic toy example in deep learning models is to approximate the Exclusive-OR function, that is, to find a model $D : \mathbb{R}^2 \to \mathbb{R}$ such that

$$\begin{aligned} D([0,0]^T) = D([1,1]^T) = 0 \\ D([1,0]^T) = D([0,1]^T) = 1. \end{aligned}$$

[1] The understanding of these levels as a learning *depth* is the origin of the naming for deep learning.

We can construct this model's layers as functions $f^{[1]} : \mathbb{R}^2 \to \mathbb{R}^2$ and $f^{[2]} : \mathbb{R}^2 \to \mathbb{R}$ such that

$$f^{[1]}(\boldsymbol{z}) = \begin{bmatrix} OR(\boldsymbol{z}) \\ NAND(\boldsymbol{z}) \end{bmatrix}$$
$$f^{[2]}(\boldsymbol{z}) = AND(\boldsymbol{z}).$$

Note how these functions are extracting simple information from the input, yet their stacking can approximate very well the desired behavior.

3.2 DEEP FEEDFORWARD NETWORKS

Deep feedforward networks, often times also called *Neural Networks*[2], are the most essential deep learning models. They were inspired by the works of Rosenblatt (1957) and refined through the many decades since, culminating in one of the most important deep learning models for practitioners and the basis of the most prominent results seen in recent years (GOODFELLOW; BENGIO; COURVILLE, 2016).

In this model, each layer is a simple affine operator followed by a nonlinear *activation* function. Following the notation previously presented, we can define a deep feedforward neural network $D^{FN}_{\boldsymbol{\theta}}$ with L layers as a stacking of *parametrized* functions, such that, for a given input $\boldsymbol{x}$,

$$\begin{aligned} \boldsymbol{z}^{[0]} &= \boldsymbol{x} \\ \boldsymbol{z}^{[i]} &= f^{[i]}_{\boldsymbol{\theta}^i}(\boldsymbol{z}^{[i-1]}) = g^{[i]}\left(A^{[i]}\boldsymbol{z}^{[i-1]} + \boldsymbol{b}^{[i]}\right), i = 1, \ldots, L \\ D^{FN}_{\boldsymbol{\theta}}(\boldsymbol{x}) &= \boldsymbol{z}^{[L]}, \end{aligned}$$

in which $\boldsymbol{\theta} = \left(\boldsymbol{\theta}^1, \ldots, \boldsymbol{\theta}^L\right)$ are the vectors of parameters and $g^{[i]}$ are the activation functions. It is common to see each of the $\boldsymbol{\theta}^i$ as a vector composed of the individual elements of the respective $A^{[i]}$ and $\boldsymbol{b}^{[i]}$. Usually, one denotes a single parameter vector $\boldsymbol{\theta}$ shared by all functions, assuming that each function "selects" only its own parameters, therefore, writing $f^{[i]}_{\boldsymbol{\theta}} = f^{[i]}_{\boldsymbol{\theta}^i}$.

Using this notation, we can say that, given a *target* function f^* and input data X, the task of learning a deep feedforward network $D^{FN}_{\boldsymbol{\theta}}$ is resumed to finding a set of parameters $\boldsymbol{\theta}$ such that $D^{FN}_{\boldsymbol{\theta}}(x) \approx f^*(x), \forall x \in X$.

Following the Exclusive-OR example, one can construct a two-layer deep feedforward network such that the functions $f^{[1]}_{\boldsymbol{\theta}} : \mathbb{R}^2 \to \mathbb{R}^2$ and $f^{[2]}_{\boldsymbol{\theta}} : \mathbb{R}^2 \to \mathbb{R}$ are

$$f^{[1]}_{\boldsymbol{\theta}}(\boldsymbol{z}) = \sigma\left(\begin{bmatrix} 2K & 2K \\ -2K & -2K \end{bmatrix}\boldsymbol{z} + \begin{bmatrix} -K \\ 3K \end{bmatrix}\right)$$
$$f^{[2]}_{\boldsymbol{\theta}}(\boldsymbol{z}) = \sigma\left(\begin{bmatrix} 2K & 2K \end{bmatrix}\boldsymbol{z} - 3K\right),$$

[2] Neural Networks are usually regarded as a more broad set of models, that include the deep feedforward networks. However, for the sake of simplicity, both will be treated as synonyms in this work.

where the chosen activation function $\sigma(.)$ is the sigmoid function[3] and the parameters are defined from $K \gg 1$ such that $\sigma(K) \approx 1$ and $\sigma(-K) \approx 0$. Then, it is easy to see that the first output of $f^{[1]}$ approximates the OR function applied to the input, while the second output approximates the NAND function and $f^{[2]}$ approximates an AND function.

For a network defined as previously, we say that the inner states $\boldsymbol{z}^{[i]}$ which are neither the input nor the output of the network (i.e., $i \notin \{0, L\}$) are called the *hidden layers* of the network. Note that even though the input and output dimensions are defined by the target function, the dimensions of the hidden layers are a design choice, as well as the number of hidden layers and the activation functions.

It has already been proven that a network with a single hidden layer can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^n$, for a broad range of activation functions, given that the hidden layer has enough dimensions (HORNIK; STINCHCOMBE; WHITE, 1989; LESHNO et al., 1993). Yet, even though this universal approximation theorem guarantees that such a network exists, it provides no way to find it. In practice, a network with a single hidden layer might need to be unfeasibly large to achieve the desired approximation, while deeper models can be far more efficient (GOODFELLOW; BENGIO; COURVILLE, 2016).

3.3 LEARNING

The Exclusive-OR example illustrates how a model built of multiple simple functions can approximate very well a target behavior. Yet, if we consider complex tasks (such as predicting age from humans' photographs), it is easy to see that many components and hundreds, maybe millions of parameters may be necessary[4]. Thus, it is not always reasonable (or even possible) to design these components manually. That is the reason *learning algorithms* are essential to make these models useful for realistic tasks.

One can understand what a learning algorithm is from the definition from Mitchell (1997): "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." To narrow down this broad definition to the scope of the presented work, we can consider the tasks in T to approximate a target function. Then, the experience given is the target function itself. Furthermore, the performance measure P is some sort of distance measure between the model and the target function.

[3] Applied element-wise where necessary.

[4] A simple example would be to consider a deep feedforward network designed to have images as inputs and output a single value. Even if small, 32-by-32 pixels, grayscale images are expected, the domain will have dimension 1024. Thus, if a single hidden layer of dimension 256 (a quarter of the input size) is desired before the output layer, the network will have over 200 thousand parameters.

However, it is often impractical to evaluate both the model and the target function in the entirety of the domain of interest. Such evaluation can be too expensive to compute and the behavior of the target function might not be known beforehand. Thus, the experience E is usually a (finite) set of samples from the domain paired with the outputs of the target function of interest. These can be measurements of a real process, like time (input) and the evolution of a quantity of interest (target function).

Roughly speaking, the deep learning algorithms of interest for this work are those that provide a deep learning model that maximizes a performance measure on data of the target function. This is called *training* a model.

3.3.1 Gradient Descent

The majority of deep learning algorithms of our interest require some sort of optimization. It would be natural to frame the definition of a learning algorithm as an optimization problem that maximizes the performance measure. Yet, not always the performance measure of interest is easy to compute or provides useful features for the optimization (e.g, differentiability). Therefore, it is usual to optimize indirectly, minimizing a *loss function* while aiming to improve the performance measure.

Given a model $D_{\boldsymbol{\theta}}^{FN} : \mathbb{R}^n \to \mathbb{R}^m$ and a target function $\boldsymbol{f}^* : U \subset \mathbb{R}^n \to \mathbb{R}^m$, an ideal performance measure could be $\int_U \|D_{\boldsymbol{\theta}}(\boldsymbol{x}) - \boldsymbol{f}^*(\boldsymbol{x})\| d\boldsymbol{x}$. Yet, this integral could be costly to compute and might not even be easily defined, as the target function might be accessible only through a distribution due to, e.g., noise. Instead, one could evaluate the model on a finite set of points from the domain $X = \{(\boldsymbol{x}, \boldsymbol{y}) \in U \times \mathbb{R}^m : \boldsymbol{y} = \boldsymbol{f}^*(\boldsymbol{x}) + \omega\}$ (the data that forms the experience E), where ω is, e.g., measurement error and noise, defining a loss function $l : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ (over the model's output and the target) that can be evaluated for each sample, e.g., the ℓ^2-norm. This way, the outcome of a deep learning algorithm would be a model $D_{\boldsymbol{\theta}^*}^{FN}$, such that

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{(\boldsymbol{x},\boldsymbol{y}) \in X} l(D_{\boldsymbol{\theta}}^{FN}(\boldsymbol{x}), \boldsymbol{y}).$$

Usually, though, given the data X and the model $D_{\boldsymbol{\theta}}^{FN}$, it is easy to define a *cost function* $J : \Theta \to \mathbb{R}$, which can be an aggregation of the per-sample loss function like $J(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x},\boldsymbol{y}) \in X} l(D_{\boldsymbol{\theta}}^{FN}(\boldsymbol{x}), \boldsymbol{y})$. Therefore, to train a model one can solve the optimization problem

$$\begin{aligned} \min_{\boldsymbol{\theta}} \quad & J(\boldsymbol{\theta}) \\ \text{s.t.} \quad & \boldsymbol{\theta} \in \Theta, \end{aligned}$$

where Θ is the set of feasible parameters for $D_{\boldsymbol{\theta}}^{FN}$.

The most common way to solve this optimization for deep learning models is to use the *gradient descent* algorithm. This method was first proposed by Cauchy in

the XIX century (LEMARECHAL, 2012) based on the definition of the gradient of a differentiable function. It is known that, given a differentiable function $f : A \to \mathbb{R}$ and $\boldsymbol{a} \in A$, if $\|\nabla f(\boldsymbol{a})\| \neq 0$, $\exists \gamma > 0$ such that $f(\boldsymbol{a} - \gamma \nabla f(\boldsymbol{a})) < f(\boldsymbol{a})$, that is, if one takes a small enough step in the opposite direction of the gradient, the function is certain to decrease. The scalar γ is called the *learning rate*, and defines the size of the step that is taken (GOODFELLOW; BENGIO; COURVILLE, 2016).

Following the notation of the previous example, if we assume that both the model and the cost function are differentiable, given parameters $\boldsymbol{\theta}_k$ such that $\|\nabla J(\boldsymbol{\theta}_k)\| \neq 0$, if we set

$$\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k - \gamma \nabla J(\boldsymbol{\theta}_k), \tag{3.2}$$

then, for a sufficiently small γ, we know that

$$J\left(\boldsymbol{\theta}_{k+1}\right) < J\left(\boldsymbol{\theta}_k\right),$$

i.e., the cost will decrease. Notice how intuitively it seems that if one takes enough steps like (3.2) with small enough γ, the parameters will converge to a point that is a local minimum of the cost function. In the context of the learning algorithm, every iteration in which the cost is computed on the whole training data and the parameters are updated following Equation (3.2) (or similar) is called an *epoch*.

Gradient descent usually provides poor convergence conditions (WOLFE, 1969), being considered unreliable and slow for many practical optimization problems. Still, gradient descent is known to work very well for deep learning. In this area, it has shown to achieve low values of the cost function fast enough to be useful, even if it does not find a local minimum (GOODFELLOW; BENGIO; COURVILLE, 2016)

3.3.2 Gradient Descent Variations

Over the years, many improvements on the gradient descent algorithms made it even more reliable and efficient for learning deep networks. In particular, the use of *momentum* on the update of the parameters has shown to accelerate significantly the learning process (SUTSKEVER et al., 2013). Momentum can be seen as computing a "velocity" vector over the parameter update steps, and taking this into account when performing the update. The classical way of doing this is to replace the update in Equation (3.2) by

$$\begin{aligned} v_{k+1} &\leftarrow \mu v_k - \gamma \nabla J(\boldsymbol{\theta}_k) \\ \boldsymbol{\theta}_{k+1} &\leftarrow \boldsymbol{\theta}_k + v_{k+1}, \end{aligned}$$

where v_k is the velocity at the k-th step, and μ is the momentum coefficient.

A major challenge of the gradient descent as a learning algorithm is that it introduces a parameter that is not learned in the process (those are called *hyperparameters*)

and that is crucial to finding a good model: the learning rate. Using momentum reduces a little the sensitivity of the results to the choice of the learning rate, but does so while introducing a new hyperparameter (the momentum coefficient μ). Furthermore, it is known that the cost function can be sensitive to some of the parameters while being numb to others (GOODFELLOW; BENGIO; COURVILLE, 2016).

In face of these challenges, strategies were proposed that use different learning rates for each parameter and automatically change these learning rates throughout the learning process. The simplest way of doing this is through the rationale that if the gradient with respect to a given parameter does not change sign (i.e., remains positive (or negative)) over the epochs, the cost is not very sensitive to changes in this parameter and, thus, its associated learning rate should increase; otherwise, it should decrease. This adaptive learning rate approach makes the training more stable.

Adam (KINGMA; BA, 2015) combined two gradient descent variations proposed by Duchi, Hazan, and Singer (2011) and Tieleman and Hinton (2012), encompassing both adaptive learning rates and momentum, and taking them one step further. After showing excellent empirical results over many application areas, Adam became one of the most commonly used algorithms and is seen as one of the best choices overall (AN. . . , 2017).

3.4 REGULARIZATION

One of the biggest challenges in deep learning is to get models that perform well also on samples that are not in the data provided during training, i.e., models that *generalize* well. Strategies designed to improve performance outside of the data seen during training at the cost of decreased performance in the training data are known as *regularization* (GOODFELLOW; BENGIO; COURVILLE, 2016).

One of the most common regularization practices is to penalize the magnitude of the parameters by adding a term to the cost function such as

$$J(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x},\boldsymbol{y})\in X} l(D_{\boldsymbol{\theta}}(\boldsymbol{x}), \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}),$$

where α is a hyperparameter that weighs the contribution of the new term and $\Omega : \Theta \to \mathbb{R}$ can be, e.g., the ℓ^1-norm. This can be seen as an incentive for the model to be built on simple features, in an effort to approximate the training data with the simplest model possible (GOODFELLOW; BENGIO; COURVILLE, 2016). One may ponder why this may be useful, when limiting the complexity of the model through hard constraints is rather easy (e.g., limiting the number of parameters, reducing Θ, etc.). Unfortunately, properly defining these hard constraints is not trivial, as finding the “simplest” model that is still able to approximate the target function can be hard.

In practice, complex models properly regularized almost always perform better than simpler models (GOODFELLOW; BENGIO; COURVILLE, 2016).

Another way to improve generalization is to use regularization to drive the learning towards models that present some desired characteristics. One of the most common of such characteristics is noise robustness. If one wants to train a model that is robust to noise in its input, one way to achieve this is to reduce the output's sensitivity. The output can be said less sensitive to variations of an input dimension if its derivative with respect to this input dimension has a small magnitude. This can be enforced through a regularization term on the gradients of the model's outputs (DRUCKER; LE CUN, 1992). E.g., given a model $D_{\boldsymbol{\theta}} : \mathbb{R}^n \to \mathbb{R}$, one can use a regularization term of the form

$$\Omega\left(\boldsymbol{\theta}\right) = \sum_{(\boldsymbol{x},y)\in X} \|\nabla D_{\boldsymbol{\theta}}\left(\boldsymbol{x}\right)\|_2.$$

Note, however, that to use gradient descent with this type of regularization, one must be able to compute second-order derivatives of $D_{\boldsymbol{\theta}}$.

3.5 BACK-PROPAGATION

Even from the most simplistic description of the gradient descent algorithm, as in Equation (3.2), it is clear that the challenge lies in computing the gradient of the cost function. Taking deep feedforward networks as an example, the analytical formula for this gradient can be derived without much effort. Yet, evaluating this formula can be quite expensive, given that, as already seen, deep learning models can have millions of parameters, thus making even the computation of a linear application non-trivial. The *back-propagation* algorithm (RUMELHART; HINTON, Geoffrey E.; WILLIAMS, 1986) provides a clever way of computing the gradient with respect to each parameter without great computational costs.

Back-propagation works based on the *chain-rule* for the derivatives. We first note that, from (3.2) and the definition of the cost function, ∇J can be reduced to evaluating the gradient of the loss function at several points. Therefore, if we want to compute the derivative of the cost function with respect to each of the parameters $\boldsymbol{\theta}^i$, $i = 1, \ldots, L$, we need to look at the derivative of the loss function with respect to these parameters. Now, by the chain-rule, let us take the case for $\boldsymbol{\theta}^L$ and see that

$$\begin{aligned}\frac{\partial l}{\partial \boldsymbol{\theta}^L} &= \nabla_{D_{\boldsymbol{\theta}}} l \frac{dD_{\boldsymbol{\theta}}}{d\boldsymbol{\theta}^L} \\ &= \nabla_{z^{[L]}} l \frac{df^{[L]}_{\boldsymbol{\theta}^L}}{d\boldsymbol{\theta}^L},\end{aligned}$$

where $\nabla_{z^{[L]}} l$ is the gradient of the loss function with respect to the model's output ($D_{\boldsymbol{\theta}}(x) = z^{[L]}$) and $\frac{df^{[L]}_{\boldsymbol{\theta}^L}}{d\boldsymbol{\theta}^L}$ is the Jacobian of the last layer of the model. Now, if we

consider the case for $\boldsymbol{\theta}^{L-1}$ and $\boldsymbol{\theta}^{L-2}$

$$\frac{\partial l}{\partial \boldsymbol{\theta}^{L-1}} = \nabla_{z^{[L]}} l \frac{df^{[L]}_{\boldsymbol{\theta}^L}}{dz[L-1]} \frac{df^{[L-1]}_{\boldsymbol{\theta}^{L-1}}}{d\boldsymbol{\theta}^{L-1}}$$
$$\frac{\partial l}{\partial \boldsymbol{\theta}^{L-2}} = \nabla_{z^{[L]}} l \frac{df^{[L]}_{\boldsymbol{\theta}^L}}{dz[L-1]} \frac{df^{[L-1]}_{\boldsymbol{\theta}^{L-1}}}{dz[L-2]} \frac{df^{[L-2]}_{\boldsymbol{\theta}^{L-2}}}{d\boldsymbol{\theta}^{L-2}},$$

it is easy to see repeating terms: the result of $\nabla_{z^{[L]}} l \frac{df^{[L]}_{\boldsymbol{\theta}^L}}{dz^{[L-1]}}$, necessary to compute $\frac{\partial l}{\partial \boldsymbol{\theta}^{L-1}}$, can be reused to compute $\frac{\partial l}{\partial \boldsymbol{\theta}^{L-2}}$ as well as the gradients of $\boldsymbol{\theta}^{L-3}, \ldots, \boldsymbol{\theta}^1$.

Back-propagation exploits this by computing the gradients with respect to each parameter "from left to right", that is, starting by computing the gradient of the outer-most component of the composition and storing the intermediate result. Applied to the equations above, the first operation would be to compute $\boldsymbol{u}^T \leftarrow \nabla_{z^{[L]}} l$, which would be used in the vector-Jacobian product

$$\boldsymbol{u}^T \frac{df^{[L]}_{\boldsymbol{\theta}^L}}{d\boldsymbol{\theta}^L}$$

to compute the gradient with respect to $\boldsymbol{\theta}^L$. Then, the intermediate result can be updated through the vector-Jacobian product $\boldsymbol{u}^T \leftarrow \boldsymbol{u}^T \frac{df^{[L]}_{\boldsymbol{\theta}^L}}{dz^{[L-1]}}$, which is useful to compute the gradient with respect to $\boldsymbol{\theta}^{L-1}$.

It is easy to see that this procedure can be repeated, propagating the gradient through each layer of the network until every parameter is covered, effectively reducing the evaluation of the gradient to computing several vector-Jacobian products. Furthermore, back-propagation can be also very memory efficient, as the chain-rule can be applied to each $f^{[i]}_{\boldsymbol{\theta}^i}$ and, thus, the stored values are limited by the layer size[5].

[5] The presented approach considers the $\boldsymbol{\theta}^i$ vectors as the parameters of interest, when in a practical application the gradients would be computed with respect to each of the $A^{[i]}$ and $\boldsymbol{b}^{[i]}$ parameters. Therefore, the Jacobian matrices computed would be limited by the sizes of each of these parameters. Finally, it is easy to see that the intermediate value stored $\boldsymbol{u}$ will have the same dimension as the next layer used in the vector-Jacobian product to update it.

4 PHYSICS-INFORMED LEARNING

Solving an IVP for ODEs using deep learning is not as straight-forward as it may seem. Indeed, it falls under the function approximation paradigm, in which we want to train a deep learning model to approximate a target function (which is the solution to the IVP). In this scenario, it is usual that either the target function is unknown or it is too complex to be useful, and, thus, only input-output samples are available to train the deep learning model. This is the case for most of the well-known successful applications of deep learning, such as those involving computer vision and the classification of images.

However, that is not the case for IVPs. When a solution for an IVP is desired, the target function is not known, but neither are the input-output pairs. Actually, generating such training data is essentially solving the IVP. Yet, the *dynamics* of the solution is usually well-known, in the form of an ODE. This chapter focuses on communicating an approach to harvest this knowledge efficiently and to use it to train deep feedforward networks that approximate the solution of IVPs. More specifically, the work of Raissi, Perdikaris, and Karniadakis (2019) is presented in a more limited formulation, targeting ODEs[1].

4.1 PROBLEM STATEMENT

Let us first recall the IVP. Given an ODE (written like in Equation (2.2)) and boundary conditions, we want to find a solution that satisfies both. More precisely, given $\mathcal{N} : \mathbb{R} \times \mathbb{R}^m \to \mathbb{R}^m$ and initial conditions $t_0 \in I \subset \mathbb{R}$, $\boldsymbol{y}_0 \subset \mathbb{R}^m$, we want to find $\boldsymbol{\phi} : I \to \mathbb{R}^m$ such that

$$\begin{aligned} \frac{d\boldsymbol{\phi}(t)}{dt} &= \mathcal{N}\left(t, \boldsymbol{\phi}(t)\right), t \in I \\ \boldsymbol{\phi}(t_0) &= \boldsymbol{y}_0. \end{aligned}$$

Solving this using deep feedforward networks means to train a model $D_{\boldsymbol{\theta}}^{FN} : I \to \mathbb{R}^m$, parametrized by $\boldsymbol{\theta} \in \Omega$, that satisfies the same conditions, i.e.,

$$\frac{dD_{\boldsymbol{\theta}}^{FN}(t)}{dt} = \mathcal{N}\left(t, D_{\boldsymbol{\theta}}^{FN}(t)\right), t \in I \tag{4.1}$$

$$D_{\boldsymbol{\theta}}^{FN}(t_0) = \boldsymbol{y}_0. \tag{4.2}$$

The naïve approach would be to construct a set $X = \{(t, \boldsymbol{y}) \in I \times \mathbb{R}^m : \boldsymbol{y} = \boldsymbol{\phi}(t)\}$ to be used as the experience for the learning algorithm. Then, with a properly constructed cost function and given that the model is complex enough, the model $D_{\boldsymbol{\theta}}^{FN}$ would approximate the target ϕ and, by consequence, satisfy Equation (4.1) and Equation (4.2). However, note how this approach assumes that ϕ is known, as it is required

[1] The original work focuses on partial differential equations, having ODEs as a particular case.

to construct a set X with more than just $(t_0, \boldsymbol{y}_0)$. Therefore, in many IVP setups, training a deep learning model this way would either be impossible or useless.

4.2 PHYSICS REGULARIZATION

The naïve approach described above is quite inefficient in that it does not use the information provided by the known $\mathcal{N}$ function. This is precisely the turning point for making deep learning a viable option in solving IVPs. The approach of Raissi, Perdikaris, and Karniadakis (2019) proposes to train the model using a regularization based on $\mathcal{N}$. This means to train $D_{\boldsymbol{\theta}}^{FN}$ to approximate $\boldsymbol{\phi}$ at the initial condition (since this is known by the definition), satisfying Equation (4.2), but also to train its Jacobian to approximate $\mathcal{N}$, satisfying Equation (4.1).

For this, let us define the singleton $X_b = \{(t_0, \boldsymbol{y}_0)\}$ and the set $X_{\mathcal{N}} \subset I$. Then, we can construct

$$J_b(\boldsymbol{\theta}) = \sum_{(t,\boldsymbol{y}) \in X_b} \| D_{\boldsymbol{\theta}}^{FN}(t) - \boldsymbol{y} \|_2^2 = \| D_{\boldsymbol{\theta}}^{FN}(t_0) - \boldsymbol{y}_0 \|_2^2,$$

which looks like a usual cost function, and

$$J_{\mathcal{N}}(\boldsymbol{\theta}) = \sum_{t \in X_{\mathcal{N}}} \left\| \frac{dD_{\boldsymbol{\theta}}^{FN}(t)}{dt} - \mathcal{N}\left(t, D_{\boldsymbol{\theta}}^{FN}(t)\right) \right\|_2^2.$$

Notice how the definition of $J_{\mathcal{N}}$ resembles a gradient regularization as discussed in Section 3.4. The difference is that instead of penalizing high derivatives, we want them to approach a desired value. Then, the cost function used to train the model is defined as

$$J(\boldsymbol{\theta}) = J_b(\boldsymbol{\theta}) + \lambda J_{\mathcal{N}}(\boldsymbol{\theta}),$$

where $\lambda \in \mathbb{R}^+$ is a scalar value to weight in the components of the loss function. Originally, Raissi, Perdikaris, and Karniadakis (2019) define this value as $\lambda = \frac{|X_b|}{|X_{\mathcal{N}}|}$, that is, proportional to the contribution of both sets in the training of the network. A neural network trained to learn a differential equation following a cost function as the one defined above is called a PINN.

The intuition of this approach is that J_b will guide the optimization so that Equation (4.2) is satisfied, while $J_{\mathcal{N}}$ will ponder it towards satisfying (4.1). Unfortunately, no theoretical guarantees have been published to prove this intuition. Nevertheless, the authors have provided plenty of empirical evidence together with a robustness analysis, indicating that with enough samples in $X_{\mathcal{N}}$ and a sufficiently complex model $D_{\boldsymbol{\theta}}$, a small error (e.g., $\| D_{\boldsymbol{\theta}}(t) - \boldsymbol{\phi}(t) \|$) can be achieved (RAISSI; PERDIKARIS; KARNIADAKIS, 2019). This result has also been achieved by others in different applications, validating the claims (NOAKOASTEEN et al., 2020; ZHANG; LIU; SUN, 2020; ARNOLD; KING, 2021; YUCESAN; VIANA, 2022).

Finally, note how this approach does not require that the target function is known. Actually, $X_{\mathcal{N}}$ can be constructed randomly by extracting samples of I, therefore, making deep learning an efficient approach to solving IVPs.

Example

Let us apply the approach presented above and train a PINN to solve an IVP for Newton's second law of motion. Recall that it can be modeled as a first-order ODE of the form

$$\frac{d\boldsymbol{y}(t)}{dt} = \begin{bmatrix} \frac{dy_1(t)}{dt} \\ \frac{dy_2(t)}{dt} \end{bmatrix} = \begin{bmatrix} y_2(t) \\ \frac{C}{M} \end{bmatrix},$$

assuming that the force applied to the object is a constant C. For simplicity, let us assume that $C = M = 1$. Finally, let us define the IVP through $t_0 = 0$ and $\boldsymbol{y}_0 = (0, 0)$, that is, at the initial time, the object stands still at the reference position. Also, the interval in which we are interested is $I = [0, 1]$.

Now, for the deep learning model, let us use a deep feedforward network with 2 hidden layers of size 10, i.e., our model is a function $D_{\boldsymbol{\theta}}^{FN} : \mathbb{R} \to \mathbb{R}^2$ such that

$$\begin{aligned} \boldsymbol{z}^{[0]} &= t \\ \boldsymbol{z}^{[i]} &= \boldsymbol{f}_{\boldsymbol{\theta}}^{[i]}(\boldsymbol{z}^{[i-1]}),\ i = 1, 2, 3 \\ D_{\boldsymbol{\theta}}^{FN}(t) &= \boldsymbol{z}^{[3]}, \end{aligned}$$

in which $\boldsymbol{f}_{\boldsymbol{\theta}}^{[1]} : \mathbb{R} \to \mathbb{R}^{10}$, $\boldsymbol{f}_{\boldsymbol{\theta}}^{[2]} : \mathbb{R}^{10} \to \mathbb{R}^{10}$, $\boldsymbol{f}_{\boldsymbol{\theta}}^{[3]} : \mathbb{R}^{10} \to \mathbb{R}^2$, and

$$\boldsymbol{f}_{\boldsymbol{\theta}}^{[i]}(\boldsymbol{z}) = \tanh\left(A^{[i]}\boldsymbol{z} + \boldsymbol{b}^{[i]}\right).$$

In the results shown in Figure 3, this model was implemented using PyTorch (PASZKE et al., 2019) and trained using Adam with $\gamma = 0.1$ and $\lambda = 0.1$. Running the algorithm for 1000 epochs took less than 2 seconds on a high-end, 16-core processor (no GPU was used). Notice how the PINN evolves over the epochs, achieving the same results of the numerical solver even though only the dynamics and the initial condition were experienced during training.

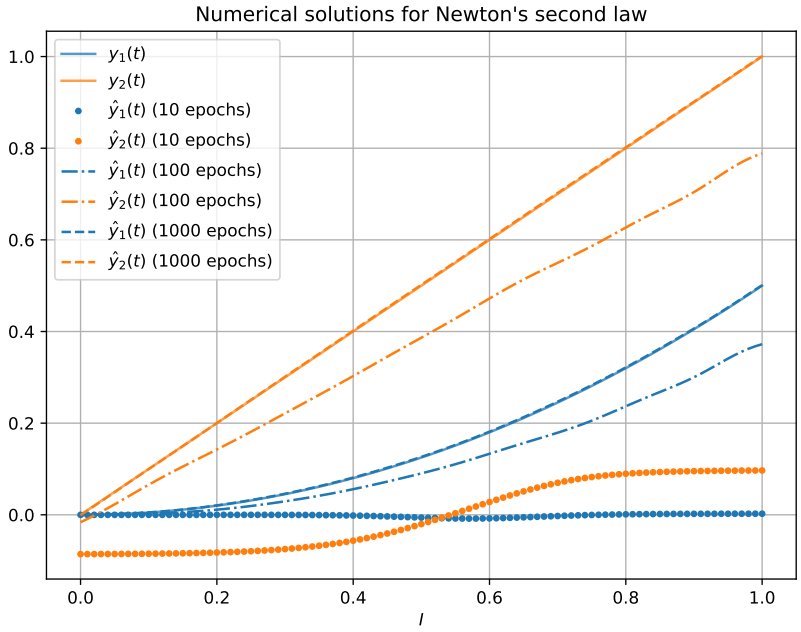


Figure 3 – Performance of a PINN in comparison to RK4 in solving an IVP of Newton's second law for $I = [0, 1]$. In the graphic, y_1 and y_2 are the results of the numerical solver, while $\hat{y}_1$ and $\hat{y}_2$ are the results of the PINN trained after different numbers of epochs. PINN results for 1000 epochs overlap RK4 results.

5 DEEP EQUILIBRIUM MODELS

This chapter's focus is to define the deep learning model that is the central investigation point of this work. DEQs have been proposed by Bai, Zico Kolter, and Koltun (2019) and Ghaoui et al. (2019), the latter naming them *implicit models*. In this chapter, we follow the notation of the former.

Furthermore, one of the greatest challenges in working with DEQs is that, by their implicit nature, they do not fit perfectly well with current deep learning tools. Therefore, to better understand the nuances and challenges that this family of models present during the experiments, a good share of attention is dedicated to the specificities of performing back propagation with DEQs.

5.1 INTRODUCTION AND DEFINITION

In Chapter 3, the intuition behind a deep learning model was introduced, that is, to model complex features through the composition of simple-yet-non-linear parametrized functions. Besides the network defined in Section 3.2 (which is the base for PINNs, as shown in Chapter 4), many other deep learning model architectures have been proposed over the years. Some of the architectures with the most surprising results involve composing the models with the same function applied multiple times, i.e., following the notation of Chapter 3, instead of defining the model as a composition similar to $D_{\boldsymbol{\theta}} = f_{\boldsymbol{\theta}}^{[L]} \circ \cdots \circ f_{\boldsymbol{\theta}}^{[1]}$, these architectures suggest a model similar to $D_{\boldsymbol{\theta}} = f_{\boldsymbol{\theta}}^{[1]} \circ \cdots \circ f_{\boldsymbol{\theta}}^{[1]}$. Inspired by this, Bai, Zico Kolter, and Koltun (2019) take a step further, defining the model with a (possibly) infinite stack of the same function, which was named DEQ.

For the definition of a DEQ $D_{\boldsymbol{\theta}}^{EQ}$, let us start with the definition of a deep learning model as proposed in Equation (3.1), but imagine it having an infinite number of layers (infinite depth). Of course, if each $f_{\boldsymbol{\theta}}^{[i]}$ is a different function (with different parameters), then this would be impossible to fit in memory. Therefore, let us assume that $f_{\boldsymbol{\theta}}^{[i]} = f_{\boldsymbol{\theta}}, \forall i$, i.e.,

$$
\begin{aligned}
z^{[0]} &= x \\
z^{[i]} &= f_{\boldsymbol{\theta}}(z^{[i-1]}), \forall i \geq 1,
\end{aligned}
$$

in which the output would be

$$
D_{\boldsymbol{\theta}}^{EQ}(x) = z^{\star} = z^{[\infty]}.
$$

If this iterative process converges, that is, if there is a number N such that $\forall i \geq N$, $z^{[i]} \approx z^{[i+1]}$, then the output $z^{\star} \approx z^{[N]}$ is well-defined, and it is also true that

$$
z^{\star} = f_{\boldsymbol{\theta}}\left(z^{\star}\right).
$$

One can say that $z^\star$ is an *equilibrium point* of $f_{\boldsymbol{\theta}}$. Therefore, the output of a well-behaved (i.e., one that respects the restrictions above presented) infinite-depth deep learning model can be computed by finding an equilibrium point.

The model proposed by Bai, Zico Kolter, and Koltun (2019) has a slight change in how it handles the input, feeding the input vector at each layer of the model. This means that the equilibrium function $\boldsymbol{f_\theta}$ has as arguments both an input vector $\boldsymbol{x}$ and a state vector $\boldsymbol{z}$. More precisely, we can say that a DEQ of the form

$$\begin{aligned} D_{\boldsymbol{\theta}}^{EQ} : \mathbb{R}^n &\longrightarrow \mathbb{R}^m \\ \boldsymbol{x} &\longmapsto D_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x}) = \boldsymbol{z}^\star \end{aligned}$$

defines its output as the equilibrium point of a function $\boldsymbol{f_\theta} : \mathbb{R}^{n+m} \to \mathbb{R}^m$ for a given input,

$$\boldsymbol{z}^\star = \boldsymbol{f_\theta}\left(\boldsymbol{x}, \boldsymbol{z}^\star\right). \tag{5.1}$$

5.2 FORWARD

The simplest way to perform the forward pass of a DEQ, i.e., to compute the output of the model given an input, is to iterate the application of the equilibrium function $\boldsymbol{f_\theta}$ until the current value is close enough to the previous. More specifically, given an input $\boldsymbol{x}$ and an initial guess $\boldsymbol{z}^{[0]}$, the procedure is to update the equilibrium guess $\boldsymbol{z}^{[i]}$ by

$$\boldsymbol{z}^{[i]} = \boldsymbol{f_\theta}(\boldsymbol{x}, \boldsymbol{z}^{[i-1]})$$

until $\|\boldsymbol{z}^{[i]} - \boldsymbol{z}^{[i-1]}\|$ is small enough. This approach is the *simple iteration* method (SÜLI; MAYERS, 2003). Even though this approach is very intuitive given our derivation of a DEQ from an infinite-depth model, it is quite limited. First because it can be quite slow, i.e., it can take many iterations until convergence is achieved, being very sensitive to the starting point. But mostly because this approach only finds equilibrium points if the function of interest is a contraction between the starting point and the equilibrium point (SÜLI; MAYERS, 2003).

This limitation can be easily visualized by trying to use the simple iteration method to find the equilibrium of $f(z) = 2z - 1$. The function clearly has an equilibrium at $z = 1$. Yet, at any starting point *except* the equilibrium, the simple iteration method will diverge.

Luckily, we know from Equation (5.1) that the equilibrium point, for a given input, is also the root of a function $\boldsymbol{g_x}(\boldsymbol{z}) = \boldsymbol{f_\theta}(\boldsymbol{x}, \boldsymbol{z}) - \boldsymbol{z}$. This means that using any root-finding algorithm on $\boldsymbol{g_x}$ yields $\boldsymbol{z}^\star$, the desired output. Perhaps the most classical root-finding algorithm is *Newton's method*, which proposes to iterate over the solution space given

$$\boldsymbol{z}^{[i+1]} = \boldsymbol{z}^{[i]} - \left(\frac{d\boldsymbol{f_\theta}(\boldsymbol{x}, \boldsymbol{z}^{[i]})}{d\boldsymbol{z}}\right)^{-1} \boldsymbol{f_\theta}(\boldsymbol{x}, \boldsymbol{z}^{[i]}),$$

in which $\frac{d}{d\boldsymbol{z}}\boldsymbol{f_\theta}(\boldsymbol{x}, \boldsymbol{z}^{[i]})$ represents the Jacobian of $\boldsymbol{f_\theta}$ with respect to $\boldsymbol{z}$[1]. Newton's method not only has guaranteed convergence for a broader class of functions in comparison to simple iteration, but also converges much faster (SÜLI; MAYERS, 2003).

5.2.1 Practical Considerations

Most modern algorithms that help us find the desired equilibrium point are either modifications of the simple iteration algorithm (e.g., Anderson Acceleration (WALKER; NI, 2011)) or modifications of Newton's method (e.g., Broyden's method (BROYDEN, 1965)). Nevertheless, all these methods require an initial guess $\boldsymbol{z}^{[0]}$ and a tolerance $\varepsilon > 0$. The initial guess, or starting point, is clearly necessary, as it is natural for iterative procedures, and is usual to find it as $\boldsymbol{z}^{[0]} \leftarrow \mathbf{0}$ by default. The tolerance is necessary to define a stopping condition for the algorithm, when the approximation for the equilibrium point is "good enough," i.e., if $\|\boldsymbol{z}^{[i]} - \boldsymbol{z}^{[i-1]}\| < \varepsilon$ then it is considered that the equilibrium has been reached. Furthermore, it is also usual to define a limit for the number of iterations, avoiding that the algorithms run for an indefinite amount of time.

5.2.2 Jacobian Regularization

Two common empirical findings of DEQ applications are that they are 1) unstable to architectural choices (BAI; KOLTUN; KOLTER, 2021) and 2) increasingly slower over training iterations (BAI; ZICO KOLTER; KOLTUN, 2019; WINSTON; KOLTER, 2020). This is a direct implication of the equilibrium-finding nature of the forward pass, which relies heavily on the behavior of $\boldsymbol{f_\theta}$. Intuitively, the complexity of $\boldsymbol{f_\theta}$, which depends heavily on the architecture and is expected to increase during training, makes it harder for the root-finding algorithm to converge.

In a very recent work, Bai, Koltun, and Kolter (2021) discussed how the Jacobian of $\boldsymbol{f_\theta}$ with respect to $\boldsymbol{z}$ is related to both problems. The authors propose, then, to penalize large values in this Jacobian during training and show how this increases robustness and convergence speed of DEQs, reducing training and inference times. More specifically, they propose to compute the Frobenius norm[2]of the Jacobian of $\boldsymbol{f_\theta}$ and add it as a regularization term to the cost function (see Sec. 3.4).

5.3 BACKWARD

It was shown in Sec. 3.5 that, in order to use a gradient descent algorithm to train a deep learning model, one must compute the derivatives of the cost function with

[1] To avoid the computational burden of inverting the Jacobian matrix, it is usual that the iteration focuses on solving $\frac{d\boldsymbol{f_\theta}(\boldsymbol{x}, \boldsymbol{z}^{[i]})}{d\boldsymbol{z}} \left(\boldsymbol{z}^{[i+1]} - \boldsymbol{z}^{[i]}\right) = -\boldsymbol{f_\theta}(\boldsymbol{x}, \boldsymbol{z}^{[i]})$ instead.

[2] The Frobenius norm of a matrix A can be written $\|A\|_F = \sqrt{\sum_{i,j=1}^{n} |a_{i,j}|^2}$.

respect to the model's parameters. In the case of DEQs, this computation is not straightforward. Given $D_{\boldsymbol{\theta}}^{EQ} : \mathbb{R}^n \to \mathbb{R}^m$ a DEQ as defined above, computing the derivative of a cost function $J : \Omega \to \mathbb{R}$ with respect to the parameters can be seen as

$$\nabla_{\boldsymbol{\theta}} J\left(\boldsymbol{\theta}\right) = \nabla_{\boldsymbol{z}^\star} J(D_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x})) \frac{dD_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x})}{d\boldsymbol{\theta}}.$$

At the same time, we know that the output of a model is given by an equilibrium point of $\boldsymbol{f_\theta}$, which is computed using a root finding method as

$$D_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x}) = RootFind(\boldsymbol{g}, \boldsymbol{z}^{[0]}),$$

where $\boldsymbol{g}$ is the function $\boldsymbol{g}(\boldsymbol{x}, \boldsymbol{z}) = \boldsymbol{f_\theta}(\boldsymbol{x}, \boldsymbol{z}) - \boldsymbol{z}$. Therefore, directly computing the derivatives of $D_{\boldsymbol{\theta}}^{EQ}$ requires the computation of the derivatives of the root-finding algorithm. Yet, not only root-finding algorithms are not usually differentiable, but even those that are differentiable may have an enormous computational cost to compute the derivatives. To illustrate the point, if we restrain ourselves to using the simple iteration method as the root-finding algorithm, we can apply the chain rule and decompose the derivative in computing the derivative of $\boldsymbol{f_\theta}$ for as many times as there were iterations until convergence, which can make practical applications impossible.

Luckily, we can exploit the fact that the output of the model ($D_{\boldsymbol{\theta}}^{EQ}\left(\boldsymbol{x}\right) = \boldsymbol{z}^\star$) is an equilibrium point of $\boldsymbol{f_\theta}$, as done by Bai, Zico Kolter, and Koltun (2019). This implies that, in a neighborhood of the input vector $\boldsymbol{x}$, $D_{\boldsymbol{\theta}}^{EQ}$ is a *parametrization* of $\boldsymbol{z}$ with respect to $\boldsymbol{x}$, i.e.,

$$\boldsymbol{f_\theta}\left(\boldsymbol{x}, D_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x})\right) - D_{\boldsymbol{\theta}}^{EQ}\left(\boldsymbol{x}\right) = \boldsymbol{0}$$

is true. Then, by the definition of $\boldsymbol{g}$, it is also true that

$$\boldsymbol{g}\left(\boldsymbol{x}, D_{\boldsymbol{\theta}}^{EQ}\left(\boldsymbol{x}\right)\right) = \boldsymbol{0}$$

holds in a neighborhood of the input. Therefore, by the *implicit function theorem* (applied to $\boldsymbol{g}$ with $D_{\boldsymbol{\theta}}^{EQ}$ as the parametrization), we can write

$$\frac{dD_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x})}{d\boldsymbol{\theta}} = -\left[\frac{d\boldsymbol{g}(\boldsymbol{x}, D_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x}))}{d\boldsymbol{z}}\right]^{-1} \frac{d\boldsymbol{g}(\boldsymbol{x}, D_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x}))}{d\boldsymbol{\theta}}. \tag{5.2}$$

Now, by the definition of $\boldsymbol{g}$ we know that

$$\frac{d\boldsymbol{g}(\boldsymbol{x}, \boldsymbol{z})}{d\boldsymbol{z}} = \frac{d\boldsymbol{f_\theta}(\boldsymbol{x}, \boldsymbol{z})}{d\boldsymbol{z}} - I$$

and

$$\frac{d\boldsymbol{g}(\boldsymbol{x}, \boldsymbol{z})}{d\boldsymbol{\theta}} = \frac{d\boldsymbol{f_\theta}(\boldsymbol{x}, \boldsymbol{z})}{d\boldsymbol{\theta}}.$$

Therefore, we can rewrite (5.2) as

$$\frac{dD_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x})}{d\boldsymbol{\theta}} = -\left[\frac{d\boldsymbol{f_\theta}(\boldsymbol{x}, D_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x}))}{d\boldsymbol{z}} - I\right]^{-1} \frac{d\boldsymbol{f_\theta}(\boldsymbol{x}, D_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x}))}{d\boldsymbol{\theta}}, \tag{5.3}$$

which allows us to compute the derivative of the DEQ model *regardless of how the equilibrium was computed*. This makes it possible for us to use *any* root-finding algorithm in the forward pass and still optimize the parameters of the model. Furthermore, the implicit function theorem can also be applied to find the derivatives of the DEQ model with respect to the input vector, enabling back-propagation even if the "infinite depth" is just a component of a larger deep learning model.

5.3.1 Implementation

One may notice that trading the computation of the derivative of the root-finding algorithm for the computation of a matrix inverse (the Jacobian of $\boldsymbol{f_\theta}$ with respect to $\boldsymbol{z}$) might not be a good idea. Luckily, we do not actually need to compute the Jacobian of $D_{\boldsymbol{\theta}}^{EQ}$ to perform back-propagation. As discussed in Section 3.5, all we need is to compute a vector-Jacobian product of the form $\boldsymbol{u}^T \frac{dD_{\boldsymbol{\theta}}^{EQ}(\boldsymbol{x})}{d\boldsymbol{\theta}}$, which, in turn, can also be reduced to computing other vector-Jacobian products, following Equation (5.3).

It is clear that the challenge lies in computing the vector-Jacobian product $\boldsymbol{u}^T \left[\frac{d\boldsymbol{f_\theta}}{d\boldsymbol{z}} - I\right]^{-1}$ efficiently, as a matrix inversion is oftentimes too costly. Yet, we can see that

$$
\begin{aligned}
\boldsymbol{v}^T &= \boldsymbol{u}^T \left[\frac{d\boldsymbol{f_\theta}}{d\boldsymbol{z}} - I\right]^{-1} \\
\implies \boldsymbol{u}^T &= \boldsymbol{v}^T \left[\frac{d\boldsymbol{f_\theta}}{d\boldsymbol{z}} - I\right] \\
&= \boldsymbol{v}^T \frac{d\boldsymbol{f_\theta}}{d\boldsymbol{z}} - \boldsymbol{v}^T \\
\implies \boldsymbol{v}^T &= \boldsymbol{v}^T \frac{d\boldsymbol{f_\theta}}{d\boldsymbol{z}} - \boldsymbol{u}^T. \qquad (*)
\end{aligned}
$$

The above equation tells us that, given $\boldsymbol{u}$ the result of the previous step, the challenging vector-Jacobian product can be computed by finding the equilibrium point of Equation $(*)$ on $\boldsymbol{v}$. Therefore, a root-finding algorithm can be used to compute the result of the challenging vector-Jacobian product, just like in the forward pass.

6 EXPERIMENTS AND RESULTS

In this chapter, we build on the theoretical foundations exposed above and propose a novel solution method for IVPs of ODEs. The main goal of the chapter is to validate the approach and explore its performance under different hyperparameter configurations. This is done through a series of experiments on the Van der Pol oscillator.

We benchmark our approach to PINNs, a well-established deep learning solution to this kind of problem. The comparison is made both in terms of approximation error and in solution speed. Overall, this chapter lays out the results for the discussion provided in Chapter 7.

6.1 PROBLEM DEFINITION

The Van der Pol oscillator (Sec. 2.4) was chosen as the ODE system for which an IVP will be solved. This system is known for having no analytical solution, thus becoming a benchmark for solvers. As well, Antonelo et al. (2021) have already studied a solution using PINNs, which is considered as a starting point and a reference for performance in the experiments.

More specifically, we define the first-order formulation of the Van der Pol oscillator (as presented in Equation (2.3)) as the ODE system, with $\mu = 1$. Then, the IVP is defined with initial condition $\boldsymbol{y}(0) = \boldsymbol{y}_0 = (0, 0.1)$, simulating a small perturbation to the system around the unstable equilibrium at the origin. The solution is desired for a horizon of 2 seconds. In this interval, the solution is expected to gravitate to a limit cycle, as illustrated in Figure 2.

6.1.1 Evaluation Metrics

As there is no analytical solution to the IVP at hand, the solutions will be evaluated in comparison to the approximation found using RK4. This reference was generated from 1000 points equally spaced in the solution interval $I = [0, 2]$, with a time step of 2 ms[1]. Then, a solution's approximation to the reference is measured through the integral of absolute error (IAE), which can be defined here as

$$IAE = \frac{1}{h} \sum_{i=1}^{1000} \|\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i\|_1,$$

where $\boldsymbol{y}_i$ are the points in the RK4 solution, $\hat{\boldsymbol{y}}_i$ are the points in the solution being evaluated, and $h = 0.002$ is the time step. Therefore, a solution is said to be suitable for the problem if it achieves a low IAE.

[1] Of course, this set does not include the initial condition, which is already given, meaning that the first point of evaluation is at $t = 0.002$.

Besides the quality of the approximation, the time required to achieve such an approximation is also of interest, as many applications (e.g., model predictive control) are highly time-dependent. The hardware resources used are valuable metrics, as the necessary computing power limits the range of equipment that can support a given solution. These will be auxiliary to the IAE in the analysis of the approximations.

6.2 PIDEQ

As already discussed in Section 4.1, solving IVPs is (mostly) only reasonable if using a physics-informed approach, that is, if "teaching" the model through the known dynamics instead of through actual samples of the target function. Therefore, a reasonable solution using DEQs must follow the same principle, which implies in a physics-informed training of DEQs (thus the naming physics-informed deep equilibrium model (PIDEQ)).

For the IVP defined above, we recall the definition of Section 5.1 and propose a DEQ similar to Ghaoui et al. (2019), that is, a model

$$\begin{aligned} D_{\boldsymbol{\theta}}^{EQ} : \mathbb{R} &\longrightarrow \mathbb{R}^2 \\ t &\longmapsto D_{\boldsymbol{\theta}}(t) = \hat{\boldsymbol{y}} \end{aligned}$$

such that

$$\begin{aligned} D_{\boldsymbol{\theta}}^{EQ}(t) &= C\boldsymbol{z}^\star \\ \boldsymbol{z}^\star &= \boldsymbol{f}_{\boldsymbol{\theta}}\left(t, \boldsymbol{z}^\star\right) \\ \boldsymbol{f}_{\boldsymbol{\theta}}\left(t, \boldsymbol{z}\right) &= \tanh\left(A\boldsymbol{z} + t\boldsymbol{a} + \boldsymbol{b}\right) \end{aligned} \quad , \tag{6.1}$$

in which the parameters $\boldsymbol{\theta}$ are a vectorization of the matrices and vectors, i.e., $\boldsymbol{\theta} = (A, C, \boldsymbol{a}, \boldsymbol{b})$, and the hyperbolic tangent function is applied to each element of the resulting vector.

Notice that this formulation is very close to the one used in Chapter 5, except for the linear transformation of the equilibrium point at the model's output. This implies that both forward and backward operations can be performed as previously shown, except that the term

$$\frac{dD_{\boldsymbol{\theta}}^{EQ}}{d\boldsymbol{z}^\star} = C$$

must be considered in the computation of the gradients. The advantage of this modification is that we can have $\boldsymbol{z}$ with an arbitrary dimension, that is, we can have an arbitrary *number of states*, which results in arbitrary representational power (GHAOUI et al., 2019).

The challenge in physics-informing a DEQ is to optimize a cost function on its derivatives. As it was shown in Section 5.3, Bai, Zico Kolter, and Koltun (2019) have proposed an efficient way to compute the first derivative of a DEQ with regard to either

its parameters or the input, which allows us to compute the cost function value. Yet, for an application of gradient descent with such cost function, it is required that the second derivatives of the model (with respect to the input and the parameters) are computable during training time. In practice, this implies in the computation of the derivative of the root-finding algorithm used to compute the first derivative, as seen in Section 5.3.1.[2] This restricts us to using differentiable root-finding algorithms to compute the first derivative, relying on automatic differentiation tools to compute the second derivative.

Finally, we propose to use a cost function for training a PIDEQ model of the form

$$J(\boldsymbol{\theta}) = J_b(\boldsymbol{\theta}) + \lambda J_{\mathcal{N}}(\boldsymbol{\theta}) + \kappa \left\| \frac{d\boldsymbol{f}_{\boldsymbol{\theta}}}{d\boldsymbol{z}} \right\|_F,$$

in which J_b, $J_{\mathcal{N}}$ and λ are as defined in Section 4.2, while $\left\| \frac{d\boldsymbol{f}_{\boldsymbol{\theta}}}{d\boldsymbol{z}} \right\|_F$ is the Frobenius norm of the Jacobian of the equilibrium function, as exposed in Sec. 5.2.2, a regularization term weighted by $\kappa \in \mathbb{R}_+$. This cost function ensures the physics-informed training from PINNs with the regularization term which was shown essential to DEQs.

6.3 TRAINING

Early experiments helped to define the design choices for the training algorithm and the hyperparameters. Adam (KINGMA; BA, 2015) was the optimizer of choice, with $\gamma = 0.001$ and default configuration. The cost-weighting coefficients were initially set as $\lambda = 0.1$ and $\kappa = 1.0$. The solver used for the forward pass of the model was the Anderson Acceleration (WALKER; NI, 2011), while best results were found using simple iteration to compute the backward pass. For both algorithms, the tolerance was set as $\varepsilon = 10^{-4}$ and they were limited to 200 iterations. The initial guess was always $\boldsymbol{z}_0 = \mathbf{0}$.

The set X_b, of course, contained only the initial condition, while $X_{\mathcal{N}}$ contained 10^5 random samples within I. The regularization term $\left\| \frac{d\boldsymbol{f}_{\boldsymbol{\theta}}}{d\boldsymbol{z}} \right\|_F$ was computed on the input values of both data sets.

6.4 EXPERIMENTS

The experiments reported below concern the training of the deep learning model proposed above to solve the IVP described in the first section of this chapter. The implementation was done using PyTorch (PASZKE et al., 2019) and is available at `https://github.com/brunompacheco/pideq`. All experiments were performed on a high-end computer, using an RTX 3060 GPU. A budget of 50000 epochs was chosen. 5 models with each configuration were trained, with different initial (random) weights.

[2] In theory, it could be possible to use the implicit function theorem once again to achieve an analytical formula for the second derivative, but this is left for future work.

6.4.1 Baseline Results

The baseline result for a PINN trained to solve an IVP of the Van der Pol oscillator comes from Antonelo et al. (2021). We replicate these results, that is, a traditional PINN with 4 hidden layers with 20 nodes each, following the same setup proposed above (except for the regularization term of the cost function). At the same time, Ghaoui et al. (2019) showed that a DEQ with as many states as there are hidden nodes in a given deep feedforward network, has *at least* as much representational power as that network. Then, our starting DEQ is a model with 80 states. The results for both models can be seen in Figure 4.

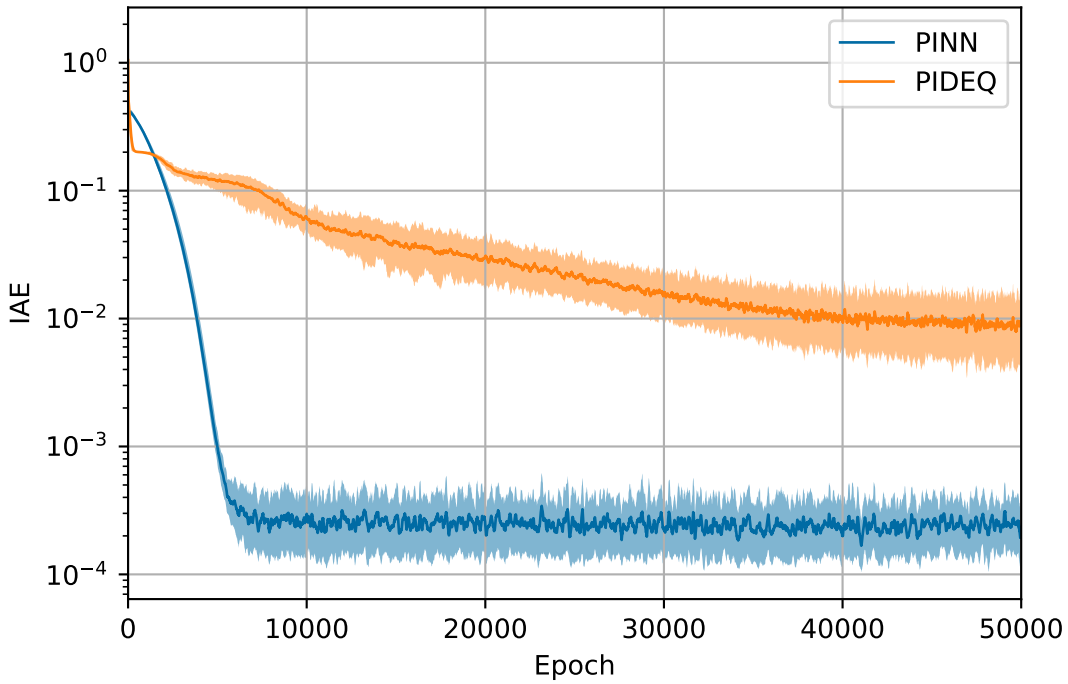
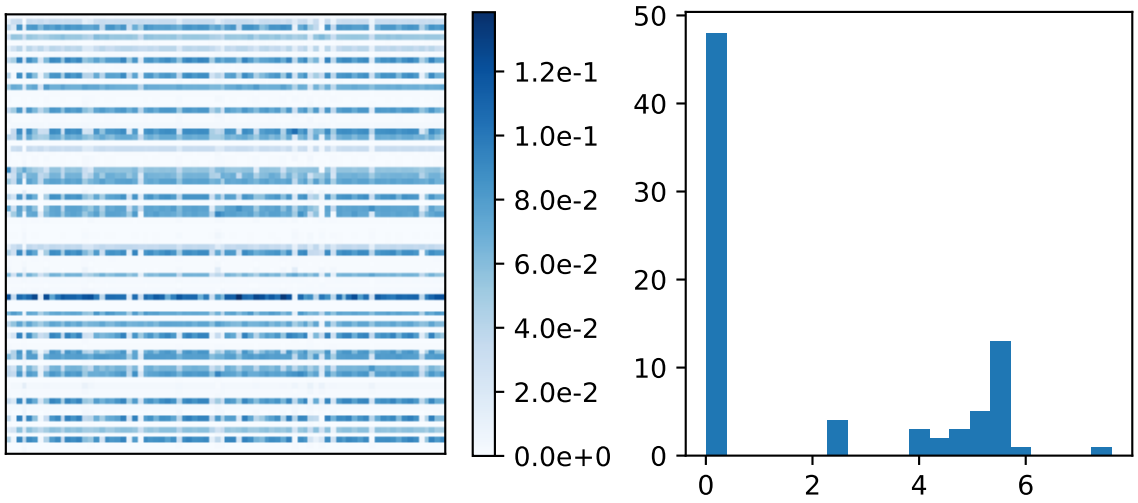


Figure 4 – Learning curve for the baseline models trained on the IVP for the Van der Pol oscillator. Solid lines are mean values ($n = 5$), shaded regions represent minimum and maximum values. For a better visualization, a moving average of 100 epochs was taken.

Given that the PINN was able to "learn" (i.e., to achieve a low IAE on) the task, we expect that the large PIDEQ model also is capable of learning. PIDEQs indeed achieved a low IAE, yet not as low as the PINNs. Furthermore, PIDEQs had a much slower convergence rate, while PINNs converged in under 10000 epochs. Besides being much more complex (in its structure) than the feedforward network, the PIDEQ also has many more parameters, in a total of 6802 against 1342. This results in a big difference in the effective training time: the PINNs took, on average, 11 ms for each epoch, while the PIDEQs took 209 ms.

6.4.2 Number of States

A deeper look at the A matrix of the baseline PIDEQs after training, as illustrated in Figure 5, shows that many of the rows are practically zero vectors, in comparison to the remaining. Having "empty" rows implies that the states associated with these rows are not effectively contributing to the results, that is, given the parameters of this model, a smaller model could be constructed with the same result.



(a) Magnitude of the elements of A. (b) Histogram of ℓ_1 norm of the rows of A.

Figure 5 – A matrix of the baseline PIDEQ after training. From all models trained, the one with median final performance was used to generate the graphics.

This analysis motivates the training of models with fewer states. The experiment was defined as an iterative procedure based on the intuition above. A new model was trained with fewer states, its A matrix was analyzed, and, if there were still empty rows in the A matrix, a new model was proposed with even fewer states. This procedure was repeated for models with 40, 20, 10 and 5 states. The models with 5 states showed no "empty rows" in the A matrix, as shown in Figure 7, but still a model with only two states was trained. The results can be seen in Figure 6.

The smaller models achieve even better results than the baseline and converge much faster, except for the smallest one, which was not able to learn within the training budget, reinforcing the intuition exposed above. The model with 5 states takes more epochs to converge, intuitively this can be attributed to it being close to its representational limit, but as it is much smaller (with only 52 parameters), it trains faster than the larger ones. Table 1 exposes the times for each model.

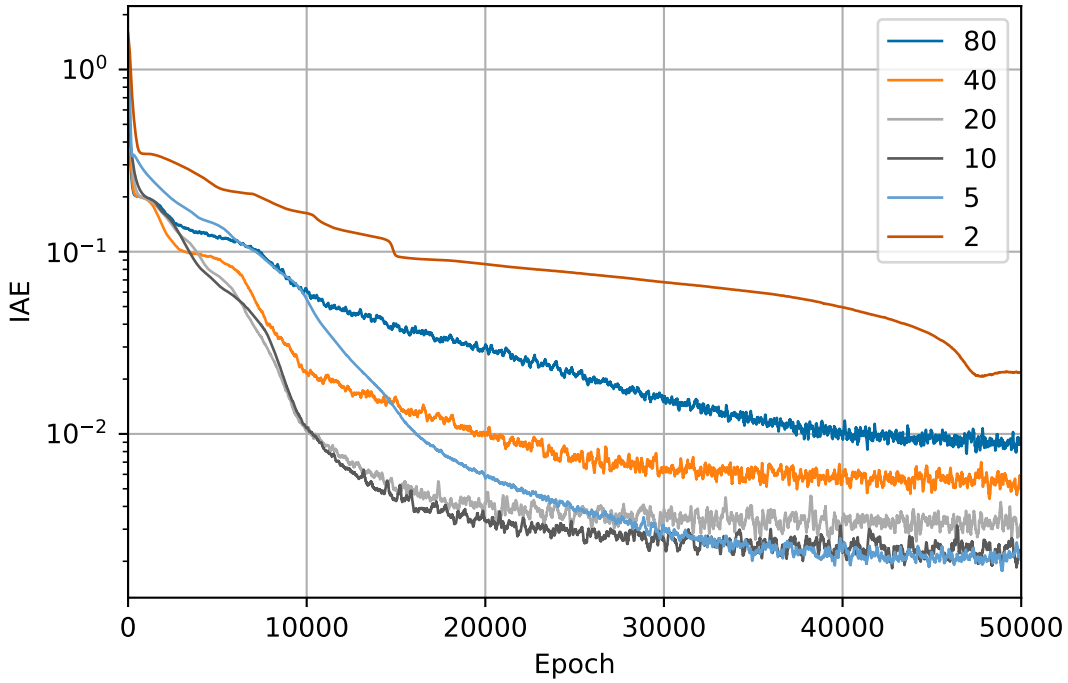


Figure 6 – Learning curve of the models trained with fewer states. Solid lines are mean values ($n = 5$). For a better visualization, a moving average of 100 epochs was taken.

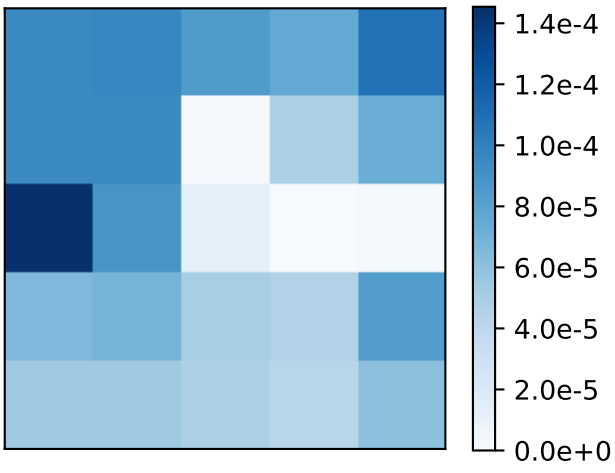


Figure 7 – A matrix of the PIDEQ with 5 states. From the 5 models trained, the one with median final performance was selected to generate the graphic.

6.4.3 Jacobian Regularization

Following the previous results, the impact of the Jacobian regularization term in the cost function is explored for the model with 5 states. From the work of Bai, Koltun, and Kolter (2021), it is expected that this regularization term will have an impact on the results. We verify this by training PIDEQ models without such term ($\kappa = 0$) and with a

Table 1 – Median training and validation times per epoch for PIDEQ models with different number of states.

Number of States	**Train** [ms]	**Validation** [ms]
80	99.1	2.1
40	51.2	1.7
20	37.1	1.2
10	29.7	1.1
5	26.0	1.2
2	24.8	1.2

reduced importance ($\kappa = 0.1$). The results are shown in Figure 8.

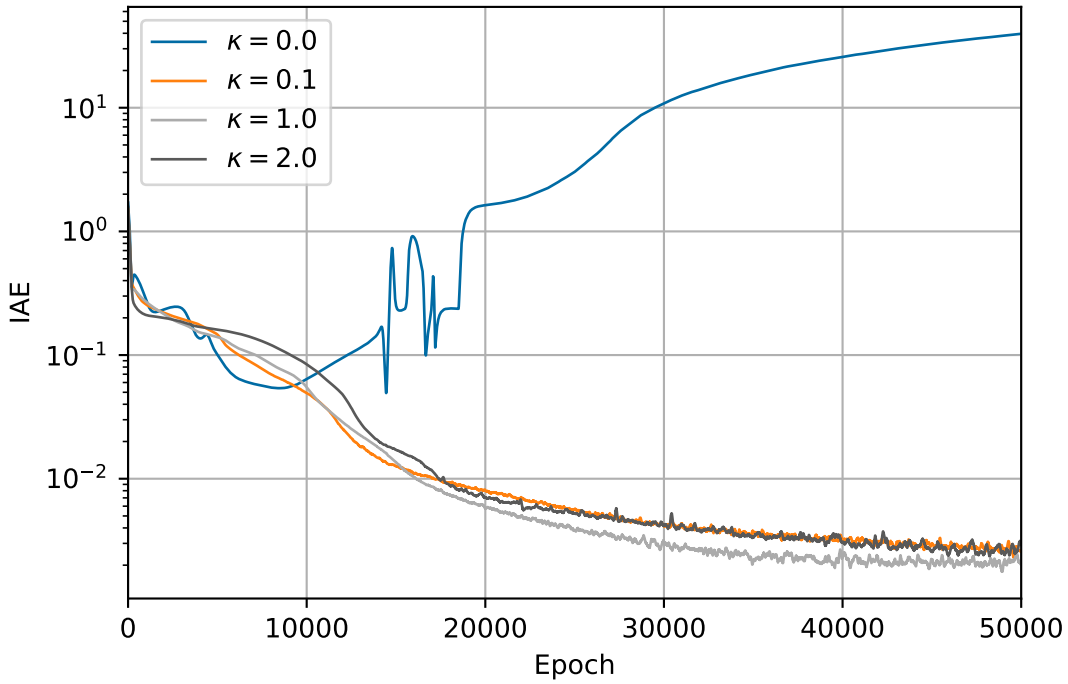


Figure 8 – Learning curve for PIDEQs with different κ values. Only one model was trained with $\kappa = 0$ because the training took over 30 times longer. Solid lines are mean values ($n = 5$). For a better visualization, a moving average of 100 epochs was taken.

The experiments confirm the theory in that training without the regularization was ineffective. Not using the regularization term resulted in models not learning the task and taking over 30 times longer to train. At the same time, increasing or decreasing the impact of the regularization term in the cost function resulted in worse models overall.

6.4.4 Solver

As discussed in Section 6.2, the solver used within the computation of the first derivative of PIDEQs must be differentiable, and, for the experiments, was fixed as the simple iteration method. Nevertheless, the impact of the solver used for the forward

pass can be explored. Since the derivative is computed implicitly, solver's impact in the inference and training speed is clearer. Yet, as the algorithm of different solvers define different paths through the domain to find an equilibrium point, different equilibria can be found given distinct solvers, which can change the performance of the model with respect to error levels.

In fact, this behavior is observed in the results of the experiments comparing Anderson's Acceleration with simple iteration and Broyden's method (BROYDEN, 1965) for the forward pass, as seen in Figure 9. Models using Broyden's method achieve a smaller error, but the iterations took much longer, with a median value of 212 ms in comparison to 27 ms using Anderson Acceleration. At the same time, using simple iteration resulted in a performance as good as using Anderson Acceleration, but with a median epoch time of 9 ms. Of course, as was discussed in Section 5.2, simple iteration is less reliable than the other two methods, yet, it seems to work well for the problem at hand.

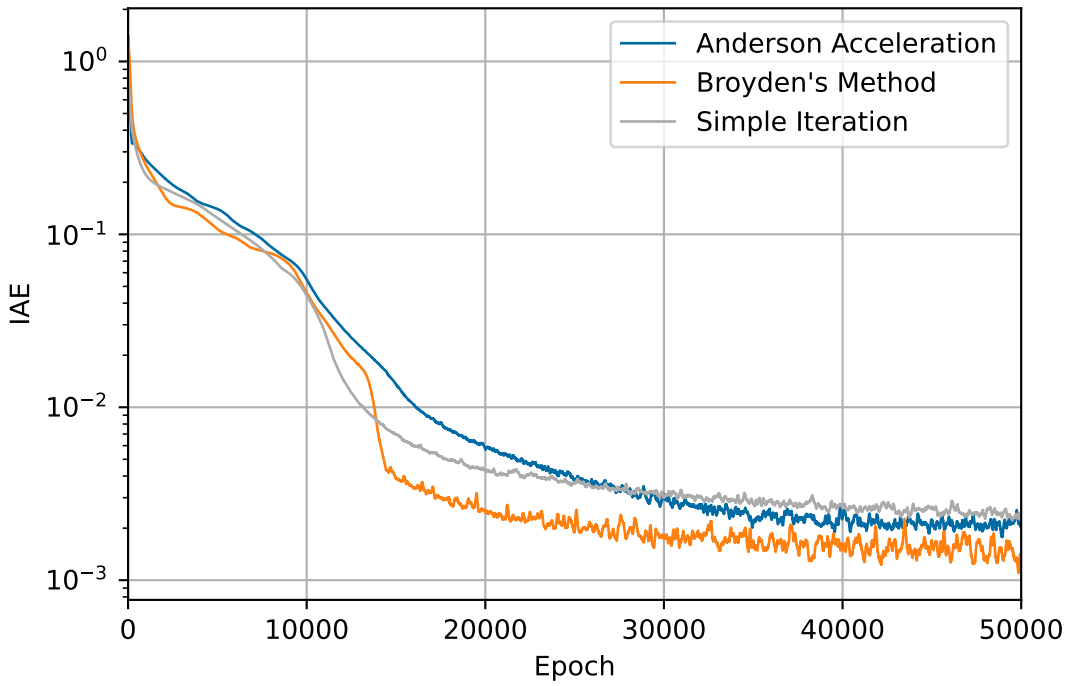


Figure 9 – Learning curve of PIDEQ models with 5 states using different solvers for the forward pass. Solid lines are mean values ($n = 5$). For a better visualization, a moving average of 100 epochs was taken.

6.4.5 Solver Tolerance

As the previous experiment showed, the simple iteration was the best approach, given the trade-off between prediction quality and training speed. Then, the ε parameter is explored next. Intuitively, it is expected that a smaller tolerance imply in a slower training but a smaller error. The results with models trained using the simple iteration al-

gorithm with $\varepsilon \in [10^{-2}, 10^{-4}, 10^{-6}]$, as exposed in Figure 10, confirm that the tolerance is indeed impactful, as increasing the "default" tolerance (of 10^{-4}) tends to increase the error of the trained models. At the same time, a decreased tolerance did not improve the performance within the training budget, possibly pointing that the default value is already sufficient for the problem.

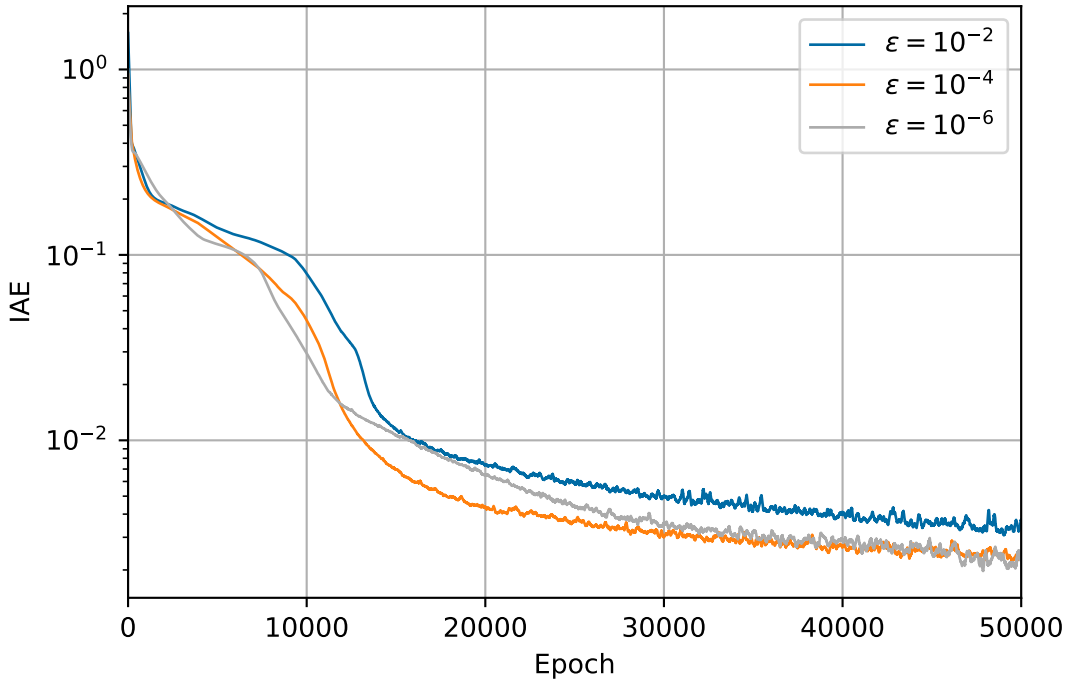


Figure 10 – Learning curve of PIDEQ models with 5 states when trained with different tolerances for the simple iteration solver. Solid lines are mean values ($n = 5$). For a better visualization, a moving average of 100 epochs was taken.

6.4.6 Final Model

The previous experiments showed how a much smaller a model can be used together with the simple iteration method, which provides a faster training. In comparison to the initial PIDEQ, there was a significant decrease in the error. Yet, the biggest impact was in training time, which decreased from 99 ms to 9 ms for each epoch (median value). The PIDEQ with only 5 states also has significantly fewer parameters (52), since the number of parameters grows quadratically with the number of states. A reduced number of parameters implies in less memory consumption and in greater explainability.

For a fair comparison, PINN models were trained with only 52 parameters, i.e., with only two hidden layers, each with dimension 5. The performance of the final model and the small PINN model can be assessed in Figure 11, in comparison to the baseline models. The final models can be assessed visually, against RK4's approximate solution, through the graphics in Figure 12.

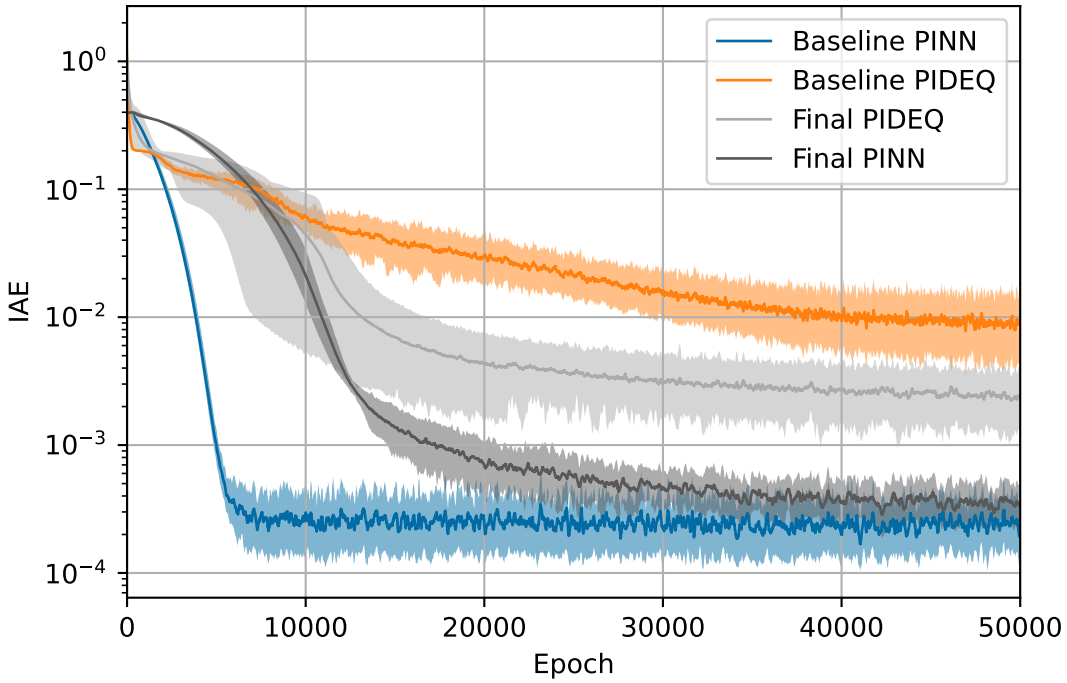


Figure 11 – Learning curve of the final models in comparison to the baselines. “Final PINN” are the small PINN models, with only 52 parameters. “Final PIDEQ” are the PIDEQ models with 5 states and using the simple iteration method as solver for the forward pass. Solid lines are mean values ($n = 5$), shaded regions represent minimum and maximum values. For a better visualization, a moving average of 100 epochs was taken.

Both final models take around the same number of epochs to converge, but the PINN models trained were able to achieve much smaller errors, almost as low as the baseline models. The small PINNs also have a smaller training time. A breakdown of the time per epoch spent by each approach can be seen in Figure 13. As expected, PIDEQs take longer to compute the output, which is a direct implication of computing the equilibrium and storing intermediate values to compute the derivatives. This, however, does not translate to inference, as both approaches take approximately the same to compute the validation results each epoch. Computing the total cost is a more expensive operation for PIDEQs. This extra cost comes mostly from the regularization term necessary to convergence, as seen in the experiments from Section 6.4.3. It is surprising, however, how the PIDEQs did not take longer than PINN models to back-propagate the cost and update the parameters..

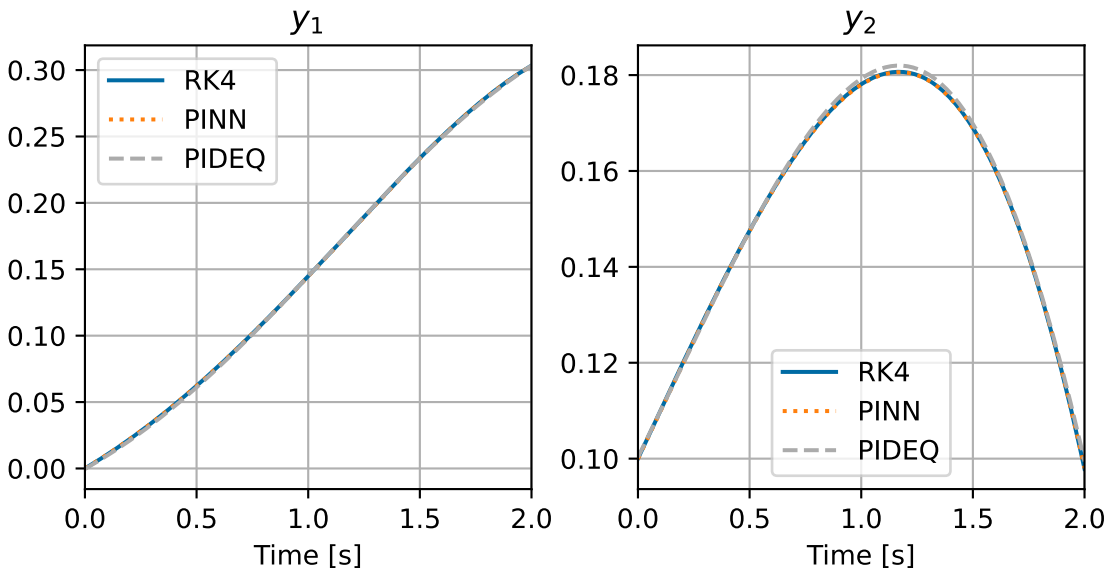


Figure 12 – Prediction of PINN and PIDEQ in comparison to the reference approximation resulting from RK4. Both models presented the median performance in the respective experiments.

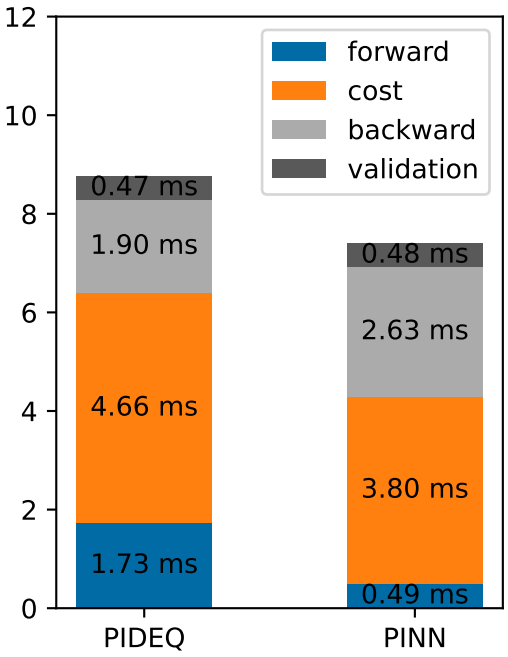


Figure 13 – Breakdown of median time per epoch for the final PIDEQ and the small PINN. “forward” represents the time necessary to compute the output given the input during the training time (with gradients computation enable). “cost” indicates the time necessary to compute the cost function, given the predictions. “backwards” is the time necessary to back-propagate the cost to the parameters.

7 CONCLUSION

An in-depth study was carried out with two novel approaches in the big area of deep learning: PINN and DEQ. Both provide a deeper connection between deep learning and other areas of knowledge, such as differential equations, optimization, and numerical analysis. PINNs provide an efficient approach to train deep learning models on problems involving physical phenomena. DEQs are a promising new architecture that can provide a larger representational power with a small number of parameters. This work proposed an application that combines both: using PIDEQs to solve IVPs of ODEs. For this, we successfully studied, implemented and tested several PIDEQ models.

To the best of our knowledge, this is the first study on physics regularization (or any gradient regularization) for DEQs. In fact, we have not found any published result reporting higher-order derivatives of these models, as Bai, Zico Kolter, and Koltun (2019) only proposed an analytical solution for the first derivative. The requirement for higher-order derivatives imposes limitations to the model, namely the use of a differentiable solver to compute the first derivative.

To validate the implementation of the model, we trained PIDEQs to solve IVPs of the Van der Pol oscillator, a well-known, ODE-governed system. The experimental results showed that differentiating through the backward-pass solver (used to compute the first derivative) did not have a big impact in the speed of training. Computing the equilibrium (forward pass) and the Jacobian regularization are the most costly operations in comparison to PINNs, at least for small PIDEQs. We hypothesize that larger models and more complex problems may result in harder-to-compute derivatives, as it may be harder to find equilibria efficiently. This would change not only the forward pass but also the Jacobian used by the backward pass, therefore, requiring more iterations of the differentiable solver (simple iteration method) or even a more robust solver.

Comparing PIDEQ results with PINN models showed that the former has a larger approximation error and slower training. Still, both presented very small errors in the proposed problems, to the point of being almost indistinguishable visually. These results indicate that the inner structure of DEQs do not provide significant advantage for learning to solve the problem at hand.

7.1 OUTLOOK

Given that DEQs approximate infinite-depth models, it is reasonable to imagine that they would be more effective in problems that benefit from deeper models, whereas in the problem considered in this work, even a shallow deep feedforward network was able to properly approximate the target function. Therefore, it is natural that a next step is to apply the proposed model to more complex problems. The four-tanks system

(JOHANSSON, 2000; GATZKE et al., 2000), which was better solved with deeper networks in the approach of Antonelo et al. (2021), could benefit from PIDEQs. Also, the proposed model could easily be generalized to partial differential equations, and then compared to the original PINN of Raissi, Perdikaris, and Karniadakis (2019).

A variation of the IVP as discussed here is to train a model that can provide approximate solutions for a set of initial conditions, as discussed in Antonelo et al. (2021) and Arnold and King (2021). This enables the efficient use of PINNs for control problems. Being able to train models for this problem with fewer parameters (such as PIDEQs) can be very useful for increasing explainability and, thus, robustness for the controller.

Lastly, another way to make PIDEQs more competitive is to improve their efficiency. In principle, the second (and any higher-order) derivative of DEQs can be implicitly computed, without the necessity of differentiating the solver, by using the implicit function theorem. This has the potential to significantly save training time, mainly for larger DEQs, by allowing the use of non-differentiable solvers for computing the first derivative of the output.

REFERENCES

AN overview of gradient descent optimization algorithms. [S.l.: s.n.], June 2017. Number: arXiv:1609.04747 arXiv:1609.04747 [cs]. DOI: 10.48550/arXiv.1609.04747. Available from: http://arxiv.org/abs/1609.04747. Visited on: 18 June 2022.

ANTONELO, Eric Aislan; CAMPONOGARA, Eduardo; SEMAN, Laio Oriel; REHBEIN DE SOUZA, Eduardo; PANAIOTI JORDANOU, Jean; HÜBNER, Fred. Physics-Informed Neural Nets for Control of Dynamical Systems, Apr. 2021. arXiv: 2104.02556. DOI: 10.48550/arxiv.2104.02556. Available from: https://arxiv.org/abs/2104.02556v2. Visited on: 5 Apr. 2022.

ARNOLD, Florian; KING, Rudibert. State–space modeling for control based on physics-informed neural networks. **Engineering Applications of Artificial Intelligence**, v. 101, p. 104195, May 2021. Publisher: Pergamon. ISSN 0952-1976. DOI: 10.1016/J.ENGAPPAI.2021.104195. Visited on: 10 Mar. 2022.

BAI, Shaojie; KOLTUN, Vladlen; KOLTER, J. Zico. Multiscale Deep Equilibrium Models. **Advances in Neural Information Processing Systems**, 2020-December, June 2020. arXiv: 2006.08656 Publisher: Neural information processing systems foundation. ISSN 10495258. DOI: 10.48550/arxiv.2006.08656. Available from: https://arxiv.org/abs/2006.08656v2. Visited on: 7 Apr. 2022.

BAI, Shaojie; KOLTUN, Vladlen; KOLTER, J. Zico. Stabilizing Equilibrium Models by Jacobian Regularization, June 2021. arXiv: 2106.14342. DOI: 10.48550/arxiv.2106.14342. Available from: https://arxiv.org/abs/2106.14342v1. Visited on: 16 May 2022.

BAI, Shaojie; ZICO KOLTER, J.; KOLTUN, Vladlen. Deep Equilibrium Models. **Advances in Neural Information Processing Systems**, v. 32, Sept. 2019. arXiv: 1909.01377 Publisher: Neural information processing systems foundation. ISSN 10495258. DOI: 10.48550/arxiv.1909.01377. Available from: https://arxiv.org/abs/1909.01377v2. Visited on: 7 Apr. 2022.

BENGIO, Yoshua; COURVILLE, Aaron; VINCENT, Pascal. Representation Learning: A Review and New Perspectives. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, v. 35, n. 8, p. 1798–1828, Aug. 2013. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence. ISSN 1939-3539. DOI: 10.1109/TPAMI.2013.50.

BROYDEN, C. G. A Class of Methods for Solving Nonlinear Simultaneous Equations. **Mathematics of Computation**, v. 19, p. 577–593, 1965.

CARTWRIGHT, Julyan H. E.; EGUÍLUZ, Víctor M.; HERNÁNDEZ-GARCÍA, Emilio; PIRO, Oreste. Dynamics of elastic excitable media. **International Journal of Bifurcation and Chaos**, v. 09, n. 11, p. 2197–2202, Nov. 1999. Publisher: World Scientific Publishing Co. ISSN 0218-1274. DOI: 10.1142/S0218127499001620. Available from: https://www.worldscientific.com/doi/abs/10.1142/S0218127499001620. Visited on: 23 June 2022.

CIREGAN, Dan; MEIER, Ueli; SCHMIDHUBER, Jürgen. Multi-column deep neural networks for image classification. In: 2012 IEEE Conference on Computer Vision and Pattern Recognition. [S.l.: s.n.], June 2012. P. 3642–3649. ISSN: 1063-6919. DOI: 10.1109/CVPR.2012.6248110.

DRUCKER, H.; LE CUN, Y. Improving generalization performance using double backpropagation. **IEEE Transactions on Neural Networks**, v. 3, n. 6, p. 991–997, Nov. 1992. Conference Name: IEEE Transactions on Neural Networks. ISSN 1941-0093. DOI: 10.1109/72.165600.

DUCHI, John; HAZAN, Elad; SINGER, Yoram. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. **Journal of Machine Learning Research**, v. 12, n. 61, p. 2121–2159, 2011. Available from: http://jmlr.org/papers/v12/duchi11a.html.

FITZHUGH, Richard. Impulses and Physiological States in Theoretical Models of Nerve Membrane. en. **Biophysical Journal**, v. 1, n. 6, p. 445–466, July 1961. ISSN 0006-3495. DOI: 10.1016/S0006-3495(61)86902-6. Available from: https://www.sciencedirect.com/science/article/pii/S0006349561869026. Visited on: 23 June 2022.

GATZKE, Edward P.; MEADOWS, Edward S.; WANG, Chung; DOYLE, Francis J. Model based control of a four-tank system. **Computers & Chemical Engineering**, v. 24, n. 2-7, p. 1503–1509, July 2000. Publisher: Pergamon. ISSN 0098-1354. DOI: 10.1016/S0098-1354(00)00555-X. Visited on: 11 Apr. 2022.

GHAOUI, Laurent El; GU, Fangda; TRAVACCA, Bertrand; ASKARI, Armin; TSAI, Alicia Y. Implicit Deep Learning. **SIAM Journal on Mathematics of Data**

Science, v. 3, n. 3, p. 930–958, Aug. 2019. arXiv: 1908.06315 Publisher: Society for Industrial & Applied Mathematics (SIAM). DOI: 10.48550/arxiv.1908.06315. Available from: https://arxiv.org/abs/1908.06315v4. Visited on: 5 Apr. 2022.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. **Deep Learning**. Ed. by Francis Bach. Cambridge, MA, USA: MIT Press, Nov. 2016. (Adaptive Computation and Machine Learning series). ISBN 978-0-262-03561-3.

GRIMSHAW, Roger. **Nonlinear ordinary differential equations**. Boca Raton: CRC Press, 1993. (Applied mathematics and engineering science texts, 2). ISBN 978-0-8493-8607-7.

HAIRER, Ernst; WANNER, Gerhard; NØRSETT, Syvert P. **Solving Ordinary Differential Equations I**. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. v. 8. (Springer Series in Computational Mathematics). ISBN 978-3-540-56670-0 978-3-540-78862-1. DOI: 10.1007/978-3-540-78862-1. Available from: http://link.springer.com/10.1007/978-3-540-78862-1. Visited on: 20 June 2022.

HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. Deep Residual Learning for Image Recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). [S.l.: s.n.], June 2016. P. 770–778. ISSN: 1063-6919. DOI: 10.1109/CVPR.2016.90.

HORNIK, Kurt; STINCHCOMBE, Maxwell; WHITE, Halbert. Multilayer feedforward networks are universal approximators. en. **Neural Networks**, v. 2, n. 5, p. 359–366, Jan. 1989. ISSN 0893-6080. DOI: 10.1016/0893-6080(89)90020-8. Available from: https://www.sciencedirect.com/science/article/pii/0893608089900208. Visited on: 15 June 2022.

ISERLES, Arieh. A first course in the numerical analysis of differential equations, second edition. **A First Course in the Numerical Analysis of Differential Equations, Second Edition**, p. 1–459, Jan. 2008. Publisher: Cambridge University Press ISBN: 9780511995569. DOI: 10.1017/CBO9780511995569. Available from: https://www.cambridge.org/core/books/first-course-in-the-numerical-analysis-of-differential-equations/2B4E05F5CFC58CFDC7BBBC6D1150661B. Visited on: 15 May 2022.

JOHANSSON, K.H. The quadruple-tank process: a multivariable laboratory process with an adjustable zero. **IEEE Transactions on Control Systems Technology**, v. 8,

n. 3, p. 456–465, May 2000. Conference Name: IEEE Transactions on Control Systems Technology. ISSN 1558-0865. DOI: 10.1109/87.845876.

KINGMA, Diederik P.; BA, Jimmy. Adam: A Method for Stochastic Optimization. In: BENGIO, Yoshua; LECUN, Yann (Eds.). **3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings**. [S.l.: s.n.], 2015. Available from: http://arxiv.org/abs/1412.6980.

KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks. In: ADVANCES in Neural Information Processing Systems. [S.l.]: Curran Associates, Inc., 2012. Available from: https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html. Visited on: 25 May 2022.

LEARY, L. **The Unity of Knowledge**. [S.l.]: Doubleday, 1955. (Bicentennial conference series). Available from: https://books.google.com.br/books?id=3rgtAAAAMAAJ.

LECUN, Yann; BENGIO, Yoshua; HINTON, Geoffrey. Deep learning. en. **Nature**, v. 521, n. 7553, p. 436–444, May 2015. Number: 7553 Publisher: Nature Publishing Group. ISSN 1476-4687. DOI: 10.1038/nature14539. Available from: https://www.nature.com/articles/nature14539. Visited on: 25 May 2022.

LEMARECHAL, Claude. Cauchy and the Gradient Method. In: DOCUMENTA Mathematica. [S.l.: s.n.], 2012. P. 251–254.

LESHNO, Moshe; LIN, Vladimir Ya.; PINKUS, Allan; SCHOCKEN, Shimon. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. en. **Neural Networks**, v. 6, n. 6, p. 861–867, Jan. 1993. ISSN 0893-6080. DOI: 10.1016/S0893-6080(05)80131-5. Available from: https://www.sciencedirect.com/science/article/pii/S0893608005801315. Visited on: 15 June 2022.

LUCERO, Jorge C.; SCHOENTGEN, Jean. Modeling vocal fold asymmetries with coupled van der Pol oscillators. **Proceedings of Meetings on Acoustics**, v. 19, n. 1, p. 060165, June 2013. Publisher: Acoustical Society of America. DOI: 10.1121/1.4798467. Available from: https://asa.scitation.org/doi/abs/10.1121/1.4798467. Visited on: 23 June 2022.

MITCHELL, Tom M. **Machine Learning**. New York: McGraw-Hill, 1997. (McGraw-Hill series in computer science). ISBN 978-0-07-042807-2.

NAGUMO, J.; ARIMOTO, S.; YOSHIZAWA, S. An Active Pulse Transmission Line Simulating Nerve Axon. **Proceedings of the IRE**, v. 50, n. 10, p. 2061–2070, Oct. 1962. Conference Name: Proceedings of the IRE. ISSN 2162-6634. DOI: 10.1109/JRPROC.1962.288235.

NOAKOASTEEN, Oameed; WANG, Shu; PENG, Zhen; CHRISTODOULOU, Christos. Physics-Informed Deep Neural Networks for Transient Electromagnetic Analysis. **IEEE Open Journal of Antennas and Propagation**, v. 1, n. 1, p. 404–412, 2020. Publisher: Institute of Electrical and Electronics Engineers Inc. ISSN 26376431. DOI: 10.1109/OJAP.2020.3013830. Visited on: 6 Apr. 2022.

PANAYOTOUNAKOS, D.e.; PANAYOTOUNAKOU, N.d.; VAKAKIS, A.f. On the lack of analytic solutions of the Van der Pol oscillator. de. **ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik**, v. 83, n. 9, p. 611–615, 2003. ISSN 1521-4001. DOI: 10.1002/zamm.200310040. Available from: https://onlinelibrary.wiley.com/doi/abs/10.1002/zamm.200310040. Visited on: 23 June 2022.

PASZKE, Adam et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: WALLACH, H.; LAROCHELLE, H.; BEYGELZIMER, A.; ALCHÉ-BUC, F. d'; FOX, E.; GARNETT, R. (Eds.). **Advances in Neural Information Processing Systems 32**. [S.l.]: Curran Associates, Inc., 2019. P. 8024–8035. Available from: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

RAISSI, M.; PERDIKARIS, P.; KARNIADAKIS, G. E. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. **Journal of Computational Physics**, v. 378, p. 686–707, Feb. 2019. Publisher: Academic Press. ISSN 0021-9991. DOI: 10.1016/J.JCP.2018.10.045. Visited on: 20 Mar. 2022.

ROSENBLATT, Frank. **The Perceptron — A Perceiving and Recognizing Automaton**. [S.l.], 1957.

RUMELHART, David E.; HINTON, Geoffrey E.; WILLIAMS, Ronald J. Learning representations by back-propagating errors. en. **Nature**, v. 323, n. 6088, p. 533–536, Oct. 1986. Number: 6088 Publisher: Nature Publishing Group. ISSN 1476-4687. DOI: 10.1038/323533a0. Available from: https://www.nature.com/articles/323533a0. Visited on: 19 June 2022.

SILVER, David et al. Mastering the game of Go with deep neural networks and tree search. en. **Nature**, v. 529, n. 7587, p. 484–489, Jan. 2016. Number: 7587 Publisher: Nature Publishing Group. ISSN 1476-4687. DOI: 10.1038/nature16961. Available from: https://www.nature.com/articles/nature16961. Visited on: 25 May 2022.

SIMMONS, George Finlay. **Differential equations with applications and historical notes**. Third edition. Boca Raton London New York: CRC Press Taylor & Francis Group, 2017. (Textbooks in mathematics). ISBN 978-1-4987-0259-1.

SÜLI, Endre; MAYERS, D. F. **An introduction to numerical analysis**. Cambridge ; New York: Cambridge University Press, 2003. OCLC: ocm50525488. ISBN 978-0-521-81026-5 978-0-521-00794-8.

SUTSKEVER, Ilya; MARTENS, James; DAHL, George; HINTON, Geoffrey. On the importance of initialization and momentum in deep learning. In: DASGUPTA, Sanjoy; MCALLESTER, David (Eds.). **Proceedings of the 30th International Conference on Machine Learning**. Atlanta, Georgia, USA: PMLR, June 2013. (Proceedings of Machine Learning Research), p. 1139–1147. Issue: 3. Available from: https://proceedings.mlr.press/v28/sutskever13.html.

TIELEMAN, T; HINTON, G. **Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude**. [S.l.: s.n.], 2012.

VAN DER POL, balthazar. A theory of the amplitude of free and forced triode vibrations. **Radio Review**, v. 1, p. 701–710, 754–762, 1920. Available from: https://cir.nii.ac.jp/crid/1573387450564694656.

WALKER, Homer F.; NI, Peng. Anderson Acceleration for Fixed-Point Iterations. en. **SIAM Journal on Numerical Analysis**, v. 49, n. 4, p. 1715–1735, Jan. 2011. ISSN 0036-1429, 1095-7170. DOI: 10.1137/10078356X. Available from: http://epubs.siam.org/doi/10.1137/10078356X. Visited on: 2 July 2022.

WINSTON, Ezra; KOLTER, J. Zico. Monotone operator equilibrium networks. In: ADVANCES in Neural Information Processing Systems. [S.l.]: Curran Associates, Inc., 2020. P. 10718–10728. Available from: https://proceedings.neurips.cc/paper/2020/hash/798d1c2813cbdf8bcdb388db0e32d496-Abstract.html. Visited on: 2 July 2022.

WOLFE, Philip. Convergence Conditions for Ascent Methods. **SIAM Review**, v. 11, n. 2, p. 226–235, Apr. 1969. Publisher: Society for Industrial and Applied Mathematics. ISSN 0036-1445. DOI: 10.1137/1011036. Available from: https://epubs.siam.org/doi/10.1137/1011036. Visited on: 17 June 2022.

YUCESAN, Yigit A.; VIANA, Felipe A.C. A hybrid physics-informed neural network for main bearing fatigue prognosis under grease quality variation. **Mechanical Systems and Signal Processing**, v. 171, p. 108875, May 2022. Publisher: Academic Press. ISSN 0888-3270. DOI: 10.1016/J.YMSSP.2022.108875. Visited on: 20 Mar. 2022.

ZHANG, Ruiyang; LIU, Yang; SUN, Hao. Physics-informed multi-LSTM networks for metamodeling of nonlinear structures. **Computer Methods in Applied Mechanics and Engineering**, v. 369, p. 113226, Sept. 2020. arXiv: 2002.10253 Publisher: North-Holland. ISSN 0045-7825. DOI: 10.1016/J.CMA.2020.113226. Visited on: 6 Apr. 2022.

ZILL, Dennis G. **A first course in differential equations with modeling applications**. Boston, MA: Brooks/Cole, Cengage Learning, 2013. OCLC: 755698615. ISBN 978-1-111-82705-2 978-1-133-49245-0.