The Pennsylvania State University
The Graduate School
College of Engineering

LEARNING CAPABILITIES OF NEURAL NETWORK AND KEPLERIAN DYNAMICS

A Thesis in
Aerospace Engineering
by
Damien Guého

© 2019 Damien Guého

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2019

The thesis of Damien Guého was reviewed and approved* by the following:

Puneet Singla
Associate Professor of Aerospace Engineering
Thesis Co-Advisor

Robert G. Melton
Professor of Aerospace Engineering
Thesis Co-Advisor

Amy R. Pritchett
Professor and Head of Aerospace Engineering

*Signatures are on file in the Graduate School.

Abstract

Machine learning and new artificial intelligence algorithms inspire the scientific community to explore and develop new approaches for discovery of scientific laws and governing equations for complex physical and nonlinear dynamical systems. The question on how well deep learning approaches can approximate a dynamic system given a set of input data is difficult to answer. Neural networks have garnered significant attention in the last decade but it is not clear how well these tools can learn the inherent characteristics of a dynamical model (such as conservation laws) from training data only. Considering the unperturbed Keplerian two-body problem, this work investigates the approximation and prediction capabilities of neural networks in learning dynamical system models in a purely recurrent model. Training neural networks with data from a single revolution around an orbit produces poor performance when predicting motion on subsequent revolutions. By incorporating deviations from constancy of angular momentum and total energy into the objective function for the neural network model, predictive performance improves significantly. Further improvements appear when a richer training data set (generated from a number of orbits with different orbital element values) is employed or with different structures such as residual or deep residual architectures. Furthermore, the effect of the mathematical representation (i.e. coordinate system) on the learning process is also investigated. From numerical results, it can be inferred that neural networks were able to better learn inherent dynamics characteristics in spherical coordinates without any apriori information than in a Cartesian coordinate system. It is shown that a simple architecture is able to learn the symmetry of the central force and reproduce the conservation of the constants of the motion.

Table of Contents

List of Figures **vi**

List of Tables **ix**

List of Symbols **xi**

Acknowledgments **xiii**

Chapter 1
Introduction **1**
1.1 Problem Significance and Motivation 1
1.2 Previous Research 2
1.3 Approach 3

Chapter 2
The Neural Network **4**
2.1 Role and Promises 4
2.2 General Description 5
2.2.1 Input, Output and Layers 5
2.2.2 Neurons and Activation Functions 6
2.2.3 Loss Function 9
2.2.4 Designing a Qualified Network 11
2.2.5 Training a Neural Network 12
2.3 The Residual Neural Network 15
2.3.1 Description of the Network 15
2.3.2 Mathematical description 17
2.3.3 Dimension and dimensionality 18
2.4 The Deep Residual Neural Network 18

Chapter 3
General Methodology **20**
3.1 The Two-body Problem 20
3.1.1 History 20
3.1.2 Dynamics and Mathematical Description in Cartesian Coordinates 21
3.1.3 General Solution of the Two-body Problem 22
3.1.4 Integrated Solution by the Neural Network 23
3.1.5 Runge-Kutta Integration Scheme with the Neural Network . 23
3.1.6 Constants of the Motion 25
3.1.7 The Loss Function: from basic expression to incorporating soft constraints 28
3.1.8 In Polar Coordinates 29
3.1.9 Computational Details 31
3.1.9.1 TensorFlow Library 31
3.1.9.2 Poliastro Library 31
3.1.9.3 Computing specifications 31

Chapter 4
Numerical Results and Discussion **33**
4.1 Method 33
4.2 Initial Neural Network Architecture and Study on the Training Set 34
4.3 Influence of the Soft Constraints in the Loss Function 44
4.4 Influence of the Architecture 51
4.5 Learning in Polar Coordinates 57

Chapter 5
Conclusion **62**
5.1 Conclusion of Research Results 62
5.2 Future Recommendations 63

Bibliography **65**

List of Figures

2.1 Generic neural network with a three-dimensional input layer, three hidden layers and a two-dimensional output layer 7

2.2 Graphical representation of a neuron 7

2.3 Different activation functions 9

2.4 Different activation functions 10

2.5 General Representation of a ResNet. The top network is represented without a wrapping layer whereas the bottom network is represented with a wrapping layer. 16

2.6 General Representation of a DeepResNet with a wrapping layer. . . 19

3.1 How the neural network is used: the upper part illustrates Eq. 3.9 with the neural network used to approximate the dynamics along with the Runge-Kutta fixed-size step algorithm while the bottom part is the classical generation of the true solution with the known dynamics and a Dormand-Prince integration algorithm. 24

3.2 4th order Runge-Kutta integration scheme using a neural network model to approximate the dynamics. 26

4.1 Evolution of the Loss function for different network architectures. . 36

4.2 Performance of NN over training data involving time prediction . . 37

4.3 Error induced by NN over training data involving time prediction . 38

4.4 Performance of NN over unknown data involving time prediction . . 39

4.5 Error induced by NN over unknown data involving time prediction . 40

4.6 Performance of NN over unknown data involving time predicton for a large dataset 42

4.7 Error induced by NN over unknown data involving time predicton for a large dataset 43

4.8 Error induced by NN over unknown data involving time predicton for a large dataset 44

4.9 Performance of NN over training data involving time prediction with soft constraints 46

4.10 Error induced by NN over training data involving time prediction with soft constraints 47

4.11 Performance of NN over unknown data involving time prediction with soft constraints 48

4.12 Error induced by NN over unknown data involving time prediction with soft constraints 49

4.13 Performance of NN over unknown data involving time prediction for a large dataset with soft constraints 50

4.14 Error induced by NN over unknown data involving time prediction for a large dataset with soft constraints 51

4.15 Evolution of the Loss function for different network architectures. . 53

4.16 Error induced by NN over unknown data involving time prediction for a large dataset with soft constraints for different structures . . . 55

4.17 Error induced by NN over unknown data involving time prediction for a large dataset with soft constraints for different structures . . . 56

4.18 Investigation on $\mathbf{g}_r$ 58

4.19 Investigation on $\mathbf{g}_\theta$ 59

List of Tables

3.1 Hardware and Software Computing Specifications 32

4.1 Orbital elements for the ten training orbits 41

4.2 NN architectures tested 52

4.3 Average errors over 3 orbits for the different NN structures considered 54

4.4 Average errors over 3 orbits for the different NN structures considered in Cartesian and polar coordinates with no penalty terms in the Loss function 60

4.5 Average errors over 3 orbits for the different NN structures considered in Cartesian and polar coordinates including penalty terms in the Loss function for training in Cartesian coordinates and initializing weigths for $\mathbf{g}_\theta(r, \theta)$ at 0 61

List of Symbols

ML	Machine Learning
AI	Artificial Intelligence
NN	Neural Network
$\mathcal{M}$	Neural Network Mapping
$\mathcal{L}$	Residual Mapping
$\mathcal{S}$	Training Set
$\mathbf{x}$	Input
$\mathbf{y}$	Output
$\tilde{\mathbf{y}}$	Predicted Output by the Neural Network
$\alpha, \boldsymbol{\alpha}$	Set of Parameters of the Neural Network
ϕ	Activation Function
$\mathbf{w}, \mathbf{W}$	Weights
$b, \mathbf{b}$	Biases
L	Loss Function
$\mathcal{O}$	Optimization Problem to solve in order to train the Network
$\eta, \gamma, \beta_1, \beta_2$	Learning rates or decay rates
ε	Smoothing term

$\mathbf{g}_i(j)$	Gradient of the objective function with respect to the parameter $\boldsymbol{\alpha}(j)$ at time step i
$\mathbf{G}_i(j)$	Sum of the squares of the gradients of the objective function with respect to the parameter $\boldsymbol{\alpha}(j)$ at time step i
$\mathbf{m}_i$	Estimate for the first moment
$\mathbf{v}_i$	Estimate for the second moment
RB	Residual Block
$\mathbf{h}$	Angular Momentum
e	Total Energy

Acknowledgments

I would like to first thank Dr. Puneet Singla and Dr. Robert Melton for their mentorship over these past two years; it has been a real pleasure to work with them. They encouraged, directed me and brought new challenges, which contributed to the completion of this work. They believed in me like nobody else very soon and gave me strong support. I am looking forward to our continued collaborations in the future.

I would like to thank the College of Engineering and the Department of Aerospace Engineering at The Pennsylvania State University as well for providing me with an assistantship to support my first year of graduate education. I am thankful to Dr. Morris and Lindsay Moist who spent invaluable time to ensure a smooth transition from France to the United States. I would like to thank Dr. Amy Pritchett for her warm welcome when I arrived in the department in August 2017.

I am also deeply thankful to all the faculty and International Relations people at École Centrale de Lyon in France. They showed great assistance, encouragement and invaluable advice. A very special thanks goes to Lisa Mihami and the Director, Frank Debouck. I extend my gratitude to members of the Challengers team with whom I have shared priceless memories.

I am also grateful to all the people I met in Penn State in general who helped me live a great life far from home. Special thanks to Mike who has become a valuable partner for lunch among other things. With the number of projects we have done together, I am happy that you can finally produce generalized algorithms. I thank all my lab partners Zach, David, David, Matt, Amit, Haxi, Jason and my other friends in State College Mitansh, Valentin, Pierre, Clément, Nicolas, Auriane, Valentin, Aurore, Mehdi and Line.

A very special thanks goes to my family who have been encouraging me and supporting me to pursue my dreams for as long as I can remember. Nobody has been more important to me in the pursuit of this project than the members of my family. My parents Emmanuel and Isabelle are my ultimate role models. They assisted and encouraged me for every step in this modern world and supported every decision I made. My first brother Quentin, has always been a major source of encouragement and is undoubtedly someone I can share my achievements with. Yohan, my second brother, is very talented and is someone I can look up to despite his lesser age. I know for sure I can rely on them in any circumstance. My uncle Christophe is my number one supporter and has always been a great source of inspiration with my grand-parents. All together, they form the best support structure I could ever ask for.

This work was partially supported through AFOSR grant number FA9550-17-1-0088. The contents are those of the author(s) and do not necessarily represent the official views of, nor an endorsement, by AFOSR or the U.S. Government.

Chapter 1 Introduction

1.1 Problem Significance and Motivation

Artificial Intelligence (AI), and deep learning approaches in particular, have recently proven very effective in certain application domains. However, a key challenge is how well AI methodologies can generalize beyond the narrow set of data they are initially trained on. As AI and deep learning are increasingly being applied to predict the behavior of complex nonlinear dynamic systems where the system state is not fully observable and the amount of training data is limited, the question of their generalizability becomes even more important. Approximation capabilities of state-of-the-art machine learning approaches (particularly deep learning) in capturing the underlying physical characteristics of a dynamical system remain poorly understood. Although these methods have been used successfully to predict the behavior of a dynamical system for a short time period (couple of time step ahead), their performance deteriorate for forecasting dynamical system behavior for a long time period. This is due to the fact that these algorithms are unable to learn underlying physical features (or characteristics) of the system. For example, it is desired that a "properly" trained machine learning based weather prediction models capture the flow features (e.g. circulation) or to reproduce conservation principles of a Hamiltonian system. Thus, it is obscure whether machine learning tools can become a valued partner in scientific discovery and technology development.

Today's methodologies are mainly applied in a clustering type of analysis of big data; human scientists must handle most of the model-centric aspects of the physics, often relying on human experience, heuristics, and already established domain-

specific methodologies. Specifically, identifying additional high value features and quantifying confidence of predictions outside of the training space all lie outside the capabilities of most state-of-the-art AI approaches and fall into the domain of human experts. As science attempts to understand ever more complex systems, and as data sets become larger and more difficult to process, new AI architectures and methods have to to generate deep insights based on physics, mathematics, and context-aware representations that succinctly capture relevant dynamics and behaviors of the complex systems under study and can generalize across domains. Data-driven approaches to establish dynamic system models have been used for centuries, especially in astrodynamics, where sometimes only input-output observations are available. Kepler's laws, Newton's laws of motion and Newton's gravitational law were developed with critical reliance on observational data. The advent of ubiquitous computing solutions has led to unprecedented breakthroughs in pattern analysis and machine intelligence. Parallel developments in sensing technologies, microelectronics and embedded systems permit the acquisition of high precision data from physical systems to enable Machine Learning (ML) approaches.

1.2 Previous Research

Identification and control of nonlinear dynamical systems using neural networks were introduced by Narendra and Parthasarathy in 1990 [1] based on observed simulation results. With rapid progress, neural networks for identification of unknown dynamical systems [2] have been subjected to a broad and comprehensive consideration for long-term prediction capabilities. There are different types of machine learning, with supervised learning, reinforcement learning, and unsupervised learning [3–5]. Machine learning methods have shown great capabilities for a wide range of applications such as improving orbit prediction accuracy through supervised machine learning [7].

While it may appear that the proliferation of data and coupled embedded systems are slated to supplant physical models and their associated insights with data driven models, the stark reality is far from that view point. The main reason is that efforts are concentrated on developing efficient learning algorithms to fit a fixed structure model to a large amount of data rather than finding the optimal model structure. The special process of dynamic system identification is to apply reverse

engineering to experimental data-sets (measurements) to eventually characterize a physical system with a mathematical model.
To address this problem, the system identification community has sought a reliable methodology to derive a mathematical model capturing the main characteritics of dynamic systems. With their remarkable ability to derive meaning from complicated or imprecise data [1], neural networks (NNs) can be used to extract patterns and detect trends that are too complex to be noticed by humans or other computer techniques. However, the ability of the NN to succeed is directly related to its specific structure [3], [4]. Some problems might be too complicated for simple structures or at least structures that require a significant computational capability. Recently, outstanding results for speech and image recognition have been achieved by some state-of-the-art NNs [5] [6], which comes from their unique learning capabilities. While model-fitting picks the model that best describes the data (among linear models, linear multivariate models, nonlinear generalized models, etc.), dynamic system identification using neural networks tailors a unique non-linear model based on the measured data [7] [8].

1.3 Approach

In this work, we study the quintessential question about approximation as well as prediction capabilities of ML tools, especially with neural networks. We consider the unperturbed two-body problem to investigate the approximation and interpolation capabilities of NN-based methods. In this scenario, the underlying nonlinear dynamics will be realized by a multi-layer neural network. Different variants of NNs will be used to investigate the approximation capabilities of NN-based algorithms. We investigate whether the trained NN model can reproduce known constants of the motion (e.g. energy and angular momentum). It should be emphasized that this work does not advocate the use of any ML tools to predict orbit states of resident space objects. This study is an attempt to understand the approximation and prediction capabilities of a multi-layer neural network approach. The two-body problem is used here as an example due to its rich history and this is a first attempt to understand how incorporation of prior knowledge about system dynamics such as conservation laws affects the learning of a neural network.

Chapter 2
The Neural Network

2.1 Role and Promises

Nature has inspired many inventions. Velcro band hook was inspired by the burdock plant while the adhesive feet of the gecko might have influenced the development of scotch tape. It would seem only logical, then, to look at the human brain's architecture for motive on how to build an intelligent machine. The first step toward artificial neural networks came in 1943 when W. McCulloch, a neurophysiologist, and W. Pitts, a mathematician, wrote an essay on how neurons might work. In their paper [9], they presented a general computational model of how biological neurons in animal cerebral cortexes might work to perform complex calculations based on propositional logic. One approach focused on biological processes in the brain while the other focused on the application of neural networks to artificial intelligence. The early successes of such types of networks led to the widespread belief that humans would soon have access to truly intelligent machines, in the sense of human intelligence. Between the 1960s and the 1980s, it became clear that the original promise would go unfulfilled mainly because of a lack of computational power. However, early in the 1980s, better techniques (for the most part in the training algorithms) and a tremendous increase in computing capacities in the 1990s made it possible to use neural networks for practical applications. Recent development introduced powerful neural network structures along with improved training algorithms capable of remarkable achievements that no other machine learning technique could hope to match. Over the past few decades, neural networks have emerged as a powerful set of tools in pattern classification, signal processing

as well as dynamical system modeling and control. When traditional modeling is difficult to achieve, networks models are frequently able to learn behavior and are able to generalize to a certain extent. Typically, a neural network consists of an organized structure composed of processing units. An appropriate number of these computational nodes may be able to capture a given system's complex input-output mapping and the number of these nodes essentially determines the degrees of freedom of the non-parametric model. According to Kolmogorov's theorem [10], any continuous function from an input subspace to an appropriate output subspace can be approximated by a two-layer neural network with finite number of nodes.

Kolmogorov's Theorem. *Let $f(\mathbf{x})$ be a continuous function defined on a unit hypercube $[0,1]^n$, then there exist simple functions ϕ_i and ψ_{ij} such that $f(\mathbf{x})$ can be represented in the following form:*

$$f(\mathbf{x}) = \sum_{i=1}^{2n+1} \phi_i \left(\sum_{j=1}^{d} \psi_{ij}(x_j) \right). \tag{2.1}$$

While neural networks are frequently described using network architecture terminology and diagrams, the reality is that any neural network results in a set of parametric interpolation functions representing the input-output behavior. Like any curve-fitting approach, neural networks must be approached with attention to whether or not the approximation architecture is a good choice for the problem at hand. According to Cover [3] and Kolmogorov's theorems, multi-layer neural networks can serve as universal approximators, but in actuality, they offer no guarantee on accuracy in practice for a reasonable dimensionality (global and distributed approximation can be at the expense of high parametric dimensionality).

Cover's Theorem. *A complex pattern classification problem or input-output problem cast in a high-dimensional space is more likely to be approximately linearly separable than in a low-dimensional space.*

2.2 General Description

2.2.1 Input, Output and Layers

A neural network can be seen as a complex nonlinear mapping between some given input and output data. Mathematically speaking, if E and F are two topological

spaces, a neural network is a mapping $\mathcal{M} : E \to F$ such that

$$\mathcal{M} : \mathbf{x} \mapsto \mathbf{y} = \mathcal{M}(\mathbf{x}), \tag{2.2}$$

where $\mathbf{x}$ is the input and $\mathbf{y}$ the output of the neural network. The mapping $\mathcal{M}$ is generally non linear and is function of a set of parameters α:

$$\mathcal{M} = \mathcal{M}_\alpha. \tag{2.3}$$

Along with the specific structure of the mapping $\mathcal{M}$, the set of parameters α defines a neural network uniquely. Based upon two components (the processing elements called neurons or perceptrons, and the connection between them called links or connections), a neural network is generally organized in layers. The input layer is built with neurons that receive data from outside the network, the output layer with neurons whose outputs are used externally, and the hidden layers with neurons that receive and produce data internally. Typically, a neural network consists of one input and one output layer which represent the input and the output of the overall network, respectively, and one or more hidden layers. Figure 2.1 presents the general structure of a simple neural network. Highly complex computations can be performed by a vast network composed of fairly simple neurons. The next section develops in detail this concept of artificial neurons.

2.2.2 Neurons and Activation Functions

A neuron is an elementary unit that plays the role of a mathematical function. The inputs of each neuron are generally numbers, each input connection being associated with a weight. The neuron receives the weighted inputs, sums them, adds a bias and passes them through a nonlinear function known as an activation function (or transfer function). This thresholding function (inspired from logic gates in threshold logic) is bounded, differentiable and often monotonically increasing and continuous. For a real-valued network, if $\mathbf{x} \in \mathbb{R}^n$ denotes the input vector, $\mathbf{w} \in \mathbb{R}^n$ the weight vector, $b \in \mathbb{R}$ a bias, the output of the neuron is $y \in \mathbb{R}$:

$$y = \phi\left(\mathbf{w}^T\mathbf{x} + b\right), \tag{2.4}$$

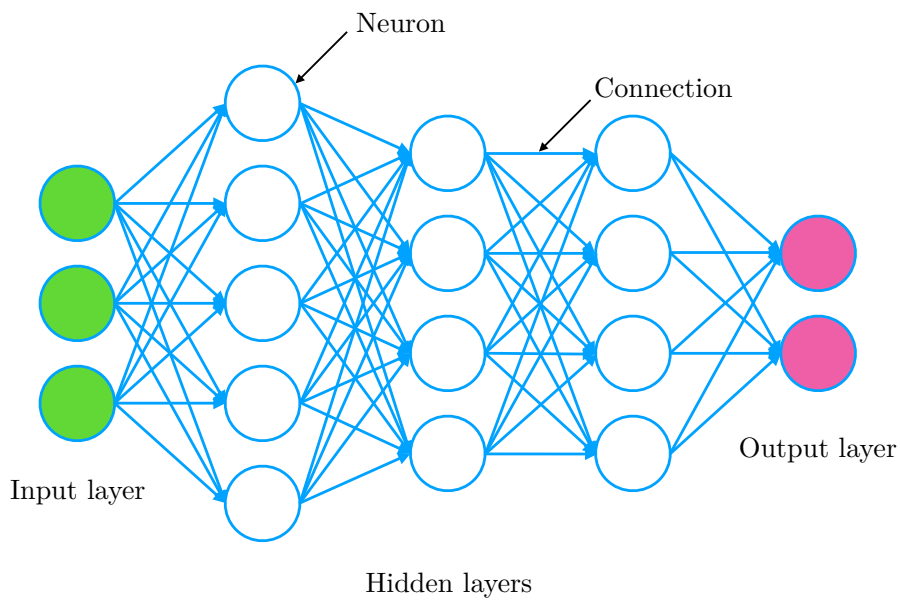


Figure 2.1: Generic neural network with a three-dimensional input layer, three hidden layers and a two-dimensional output layer

where ϕ is the nonlinear activation function. Figure 2.2 presents a graphical representation of a neuron where $n = 3$ and

$$\mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix}^T \qquad \mathbf{w} = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix}^T. \tag{2.5}$$

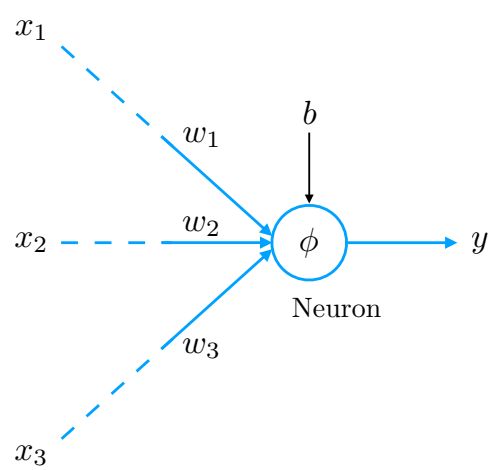


Figure 2.2: Graphical representation of a neuron

Figure 2.4 presents a short list of activation functions that are usually used. It should be mentioned that ϕ is directly part of the structure of the network $\mathcal{M}$ while $\mathbf{w}$ and b are a subset of the set of parameters α of $\mathcal{M}_\alpha$:

$$\{\mathbf{w}, b\} \subset \alpha. \tag{2.6}$$

This model of a neuron is called the perceptron and is one of the most common architectures. Used alone, it is the simplest network structure. If duplicated and organized in consecutive layers as in Figure 2.1, it can constitute a very effective network.

Looking at two consecutive layers $l-1$ and l, the weight vectors and the bias for each neuron in layer l are stacked in a matrix $\mathbf{W}^l$ and a vector $\mathbf{b}^l$ respectively. If the layers $l-1$ and l are comprised of p and q neurons respectively (meaning that the output of layer $l-1$ $\mathbf{y}^{l-1} \in \mathbb{R}^p$ and the output of layer l $\mathbf{y}^l \in \mathbb{R}^q$), the layer operator $\Psi^l : \mathbb{R}^p \to \mathbb{R}^q$ is defined as

$$\Psi^l : \mathbf{x}^l \mapsto \begin{bmatrix} \phi\left({\mathbf{w}_1^l}^T \mathbf{x}_1^l + b_1\right) & \phi\left({\mathbf{w}_2^l}^T \mathbf{x}_2^l + b_2\right) & \cdots & \phi\left({\mathbf{w}_q^l}^T \mathbf{x}_q^l + b_q\right) \end{bmatrix}^T \tag{2.7}$$

where

$$\mathbf{W}^l = \begin{bmatrix} \mathbf{w}_1^l & \mathbf{w}_1^l & \cdots & \mathbf{w}_q^l \end{bmatrix} \qquad \mathbf{b}^l = \begin{bmatrix} b_1^l & b_2^l & \cdots & b_q^l \end{bmatrix}^T. \tag{2.8}$$

A similar expression is

$$\mathbf{y}^l = \Psi^l\left(\mathbf{x}^l\right) = \phi\left({\mathbf{W}^l}^T \mathbf{x}^l + \mathbf{b}^l\right) \tag{2.9}$$

where the function ϕ is applied to the q elements of ${\mathbf{W}^l}^T \mathbf{x}^l + \mathbf{b}^l$.

Finally, the general output $\mathbf{y}$ of a network with d layers can be written as

$$\mathbf{y} = \mathcal{M}_\alpha(\mathbf{x}) = \left(\prod_{i=1}^{d} \Psi^i\right)(\mathbf{x}) := \Psi^d \circ \Psi^{d-1} \circ \cdots \circ \Psi^1(\mathbf{x}), \tag{2.10}$$

and is called a Feed-Forward neural network (also called fully-connected neural network or muti-layer perceptron). The full set of parameters α is

$$\alpha = \left\{\mathbf{W}^1, \mathbf{W}^2, \cdots, \mathbf{W}^d, \mathbf{b}^1, \mathbf{b}^2, \cdots, \mathbf{b}^d\right\} \tag{2.11}$$

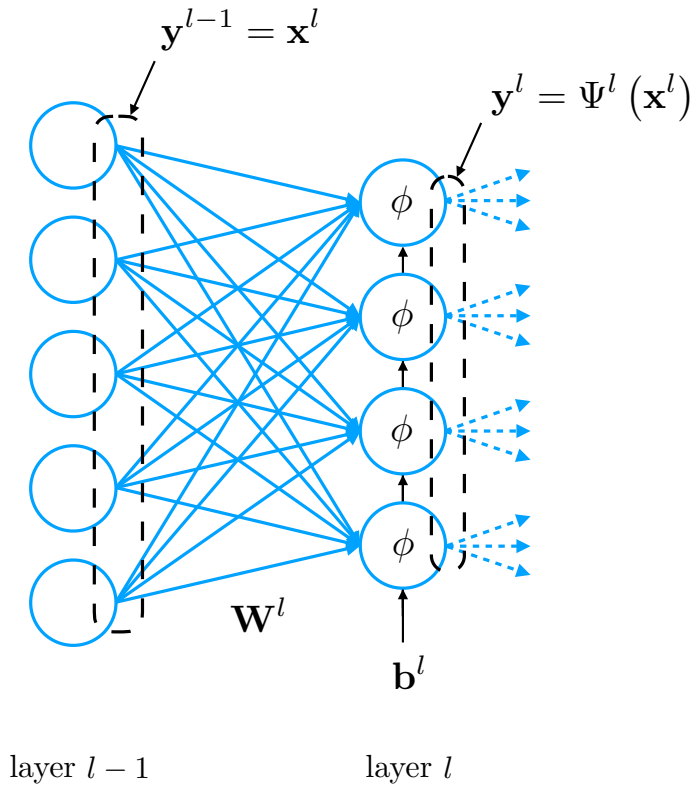


Figure 2.3: Different activation functions

As a mapping operator, measuring the efficiency of a network means measuring how well it is capable of mapping a set of inputs to a set of outputs. These concepts and the metric associated with it are discussed in the next section.

2.2.3 Loss Function

For $N \in \mathbb{N}^*$, a training data set $\mathcal{S}$ of size N is a set of pairs $(\mathbf{x}_k, \mathbf{y}_k)_{k\in[\![1,N]\!]}$ where $\mathbf{x}_k \in \mathbb{R}^n$ serves as input for the network and $\mathbf{y}_k \in \mathbb{R}^m$ is considered as the objective output of the network. $\mathbf{y}_k$ serves as a reference value (or target value) and is considered to be the truth. The value $\tilde{\mathbf{y}}_k \in \mathbb{R}^m$ is the quantity produced by the network. This is the output of the network. From Equation 2.2:

$$\tilde{\mathbf{y}}_k = \mathcal{M}_\alpha(\mathbf{x}_k). \tag{2.12}$$

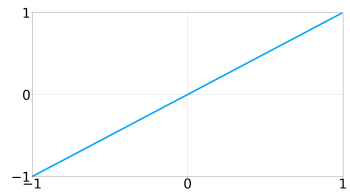
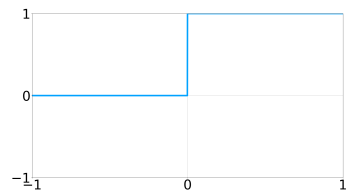
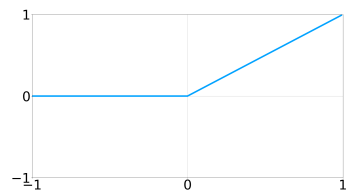
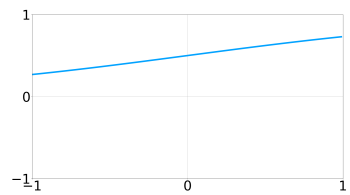
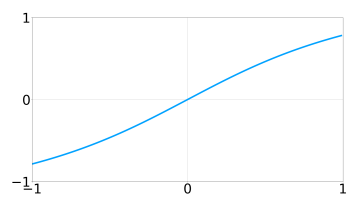
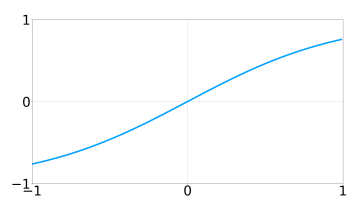
Linear	$\phi(x) = x$	
Heaviside	$\phi(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$	
ReLU	$\phi(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$	
Sigmoid	$\phi(x) = \dfrac{1}{1 + e^{-x}}$	
tan-1	$\phi(x) = \tan^{-1}(x)$	
tanh	$\phi(x) = \tanh(x)$	

Figure 2.4: Different activation functions

To measure the efficiency of the network based on the training data set $\mathcal{S}$, it is imperative to confront the output $\tilde{\mathbf{y}}_k$ of the network to the true value $\mathbf{y}_k$. For each sample $k \in [\![1, N]\!]$ in the training set, the most common metric used is the mean square error (MSE):

$$MSE(k) = \frac{1}{m} \sum_{i=1}^{m} \left(y_k(i) - \tilde{y}_k(i)\right)^2 = \frac{1}{m} \left|\left|\mathbf{y}_k - \tilde{\mathbf{y}}_k\right|\right|_2^2 . \tag{2.13}$$

Finally, the sum of the MSE over the whole training set $\mathcal{S}$ is called the Loss:

$$L = L(\mathcal{M}_\alpha, \mathcal{S}). \tag{2.14}$$

The Loss function illustrates the proficiency of the network to map two different data sets (namely $\{\mathbf{x}_k\}$ and $\{\mathbf{y}_k\}$, $k \in [\![1, N]\!]$):

$$L(\mathcal{M}_\alpha, \mathcal{S}) = \sum_{k=1}^{N} MSE(k). \tag{2.15}$$

The smaller the Loss, the better the network performs in mapping the input data set $\{\mathbf{x}_k\}$ to the targeted values $\{\mathbf{y}_k\}$, $k \in [\![1, N]\!]$. While the given training data set $\mathcal{S}$ remains unchanged for a given application, Equation 2.14 shows that the Loss is then only dependent on the network architecture $\mathcal{M}$ and network parameters α. The next two sections explain how to minimize the Loss while identifying an appropriate structure for $\mathcal{M}$ and finding the best set of parameter α.

2.2.4 Designing a Qualified Network

Choosing a correct architecture for a neural network is the subject of active research. Creating a neural network architecture means proposing a network topology for $\mathcal{M}$. Deciding how the neurons are interconnected therefore is a difficult problem. Even for a simple multi-layer perceptron as presented Figure 2.1, one can change the number of layers, the number of neurons per layer, the type of activation function to use in each neuron and other parameters. The question of how to choose a correct structure for a given problem has been addressed multiple times but there is no pragmatic technique that describe how to design an efficient neural network. It is also uncertain whether there exists a unique optimal model for a given problem

and there is different consensus regarding the impact on performance from adding additional hidden layers or increasing the number of neurons. However, designing a competent network architecture is relatively easy from the moment one has access to sufficient computation capabilities to test distinct configurations. For many problems, one can just begin with a single hidden layer. It has actually been shown that a single-layer perceptron can model even the most complex functions provided it has enough neurons. On the other hand, deeper networks can model complex functions using exponentially fewer neurons, making them much faster to train.

2.2.5 Training a Neural Network

Once the structure of the neural network has been decided, one can try to minimize the Loss function L by varying the set of parameter α which the network $\mathcal{M}$ is dependent on. Since $\mathcal{M}$ is a large and complex nonlinear operator, minimizing L is equivalent to solving a large and complex nonlinear optimization problem $\mathcal{O}$:

$$\mathcal{O} : \min_{\alpha} L(\mathcal{M}_{\alpha}, \mathcal{S}). \tag{2.16}$$

Training a neural network consiss of finding a proper set of parameters α that will yield a minimum value for L. But the flexibility of neural networks is also one of their main drawbacks: there are many parameters to adjust. Although the optimization community has studied the general problem of optimizing nonlinear functions for many years, the multilayer neural networks do not represent a typical optimization problem. Gradient descent and its various variants such as conjugate gradient are often used for optimization of smooth nonlinear functions. However, many of these optimizers do not perform very well for the training of multi-layer neural networks as they have a tendency to get stuck in local minimum due to a fixed learning rate. Furthermore, many of these optimizers do not parallelize to run on GPUs or a distributed network and hence are computationally intensive for large networks. For the training of multi-layer neural networks, different variants of gradient descent algorithms have been developed which have an adaptive learning rate to avoid local minima and plateaus in the Loss function. The most commonly used optimizers are: Adagrad [11], Adadelta optimizer [12], Adam optimizer [13], FtrlOptimizer [14], RMSprop [15] and NADAM [16]. These algorithms employ momentum based methods and/or compute averages of gradients to adjust the

learning rate to avoid local minima [17].
In this section, all the parameters are stacked in a single vector $\boldsymbol{\alpha}$:

$$\alpha \rightarrow \boldsymbol{\alpha}. \tag{2.17}$$

Considering an random initialization for $\boldsymbol{\alpha}_0$, the classical gradient descent scheme is

$$\boldsymbol{\alpha}_{i+1} = \boldsymbol{\alpha}_i - \eta \frac{\partial L}{\partial \boldsymbol{\alpha}}\bigg|_{\boldsymbol{\alpha}=\boldsymbol{\alpha}_i}. \tag{2.18}$$

Some optimizers make use of momentum that helps accelerate the gradient descent in the relevant direction and dampens oscillations by adding a fraction γ of the update vector of the past step to the current update vector:

$$\mathbf{v}_i = \gamma \mathbf{v}_{i-1} + \eta \frac{\partial L}{\partial \boldsymbol{\alpha}}\bigg|_{\boldsymbol{\alpha}=\boldsymbol{\alpha}_i} \tag{2.19}$$

$$\boldsymbol{\alpha}_{i+1} = \boldsymbol{\alpha}_i - \mathbf{v}_i. \tag{2.20}$$

The momentum term increases for dimensions whose gradients point in the same direction and reduces updates for dimensions whose gradients change direction. As a result, the gain is in general faster convergence and reduced oscillation.
Adagrad adapts the learning rate to the parameters, performing larger updates for infrequent parameters and smaller updates for frequent parameters. If $\mathbf{g}_i(j)$ is the gradient of the objective function with respect to the parameter $\boldsymbol{\alpha}(j)$ at time step i:

$$\mathbf{g}_i(j) = \frac{\partial L}{\partial \boldsymbol{\alpha}}\bigg|_{\boldsymbol{\alpha}=\boldsymbol{\alpha}_i}(j), \tag{2.21}$$

the update for the parameter $\boldsymbol{\alpha}(j)$ becomes:

$$\boldsymbol{\alpha}_{i+1}(j) = \boldsymbol{\alpha}_i(j) - \frac{\eta}{\sqrt{\mathbf{G}_i(j) + \varepsilon}} \mathbf{g}_i(j), \tag{2.22}$$

where ε is a smoothing term to avoid singularities (usually on the order of magnitude of 10^{-8}) and $\mathbf{G}_i$ is a vector where element j is the sum of the squares of the gradients with respect to $\boldsymbol{\alpha}(j)$ up to time step i:

$$\mathbf{G}_i(j) = \sum_{k=1}^{i} \mathbf{g}_k(j). \tag{2.23}$$

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate while its main weakness is the accumulation of the squared gradients in the denominator: since every added term is positive, the accumulated sum keeps growing during training which causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.
Instead of inefficiently storing previous squared gradients, Adadelta defines a running average $E[\mathbf{g}_i(j)^2]$ that depends only on the previous average and the current gradient:

$$E[\mathbf{g}_i(j)^2] = \gamma E[\mathbf{g}_{i-1}(j)^2] + (1-\gamma)\mathbf{g}_i(j)^2. \tag{2.24}$$

The update for the parameter $\boldsymbol{\alpha}(j)$ becomes:

$$\boldsymbol{\alpha}_{i+1}(j) = \boldsymbol{\alpha}_i(j) - \frac{\eta}{\sqrt{E[\mathbf{g}_i(j)^2] + \varepsilon}}\mathbf{g}_i(j), \tag{2.25}$$

Finally, the Adaptive Moment Estimation algorithm (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients, Adam also keeps an exponentially decaying average of past gradients:

$$\mathbf{m}_i = \beta_1 \mathbf{m}_{i-1} + (1-\beta_1)\mathbf{g}_i \tag{2.26}$$

$$\mathbf{v}_i = \beta_2 \mathbf{v}_{i-1} + (1-\beta_2)\mathbf{g}_i \tag{2.27}$$

$\mathbf{m}_i$ and $\mathbf{v}_i$ are estimates for the first moment (the mean) and second moment (the uncentered variance) of the gradients respectively (hence the name of the method). The update equation is then

$$\boldsymbol{\alpha}_{i+1} = \boldsymbol{\alpha}_i - \frac{\eta}{\sqrt{E[\mathbf{v}_i] + \varepsilon}}E[\mathbf{m}_i]. \tag{2.28}$$

Empirically, Adam works well in practice and compares favorably to other adaptive learning-method algorithms. All of these optimizers are available with Google's TensorFlow package and have been optimized for parallel processing.

2.3 The Residual Neural Network

Residual neural networks (ResNet), originally developed for image classification seem to require fewer parameters for enhanced accuracy [18], [19], [20], [21] and are built upon two principles, which are modularity and residual learning. The first idea behind ResNet is that they are primitively small networks that can be repeated (or stacked) to increase the depth of the network. A deeper ResNet is built by simply repeating this original module. The question of how to choose the structure (often equivalent to the number of layers) is a difficult problem. Typically, one wants to add a new layer only if some benefit results. One way to ensure this additional layer allows the network to learn something new is to also provide the input without any transformation to the output of this extra layer. This essentially drives the new layer to learn something different from what the input has already encoded. For some fixed-point problem, if the approximated value is x_c such that $\psi(x_c) \simeq x_c$ and a true value $x_0 = \psi(x_0)$, the error is $x_0 - x_c$. We call the residual the quantity $\psi(x_0) - \psi(x_c) = x_0 - \psi(x_c)$. The residual is the error in the result. In the ResNet description, instead of directly fitting a desired underlying mapping (the original mapping), we explicitly let these layers fit a residual mapping. If the underlying mapping is denoted $\mathcal{M}$, we let the ResNet fit the residual mapping

$$\mathcal{L}(\mathbf{x}) := \mathcal{M}(\mathbf{x}) - \mathbf{x}. \tag{2.29}$$

2.3.1 Description of the Network

A ResNet of depth d is a network composed of d residual blocks, each of them performing an operation. These operations are generally identical among the blocks, may be very simple or can be built upon a more complex scheme and involve several sub-operations. Given a layer $k < d$ (or a block), the particularity of the ResNet is to provide the output $\mathbf{x}^{k-1}$ of layer $k-1$ (or input $\mathbf{x}^{k-1}$ of layer k) without any transformation to the output $\mathbf{x}^k$ of this layer. Figure 2.5 (network on the top) depicts the general representation of a ResNet.

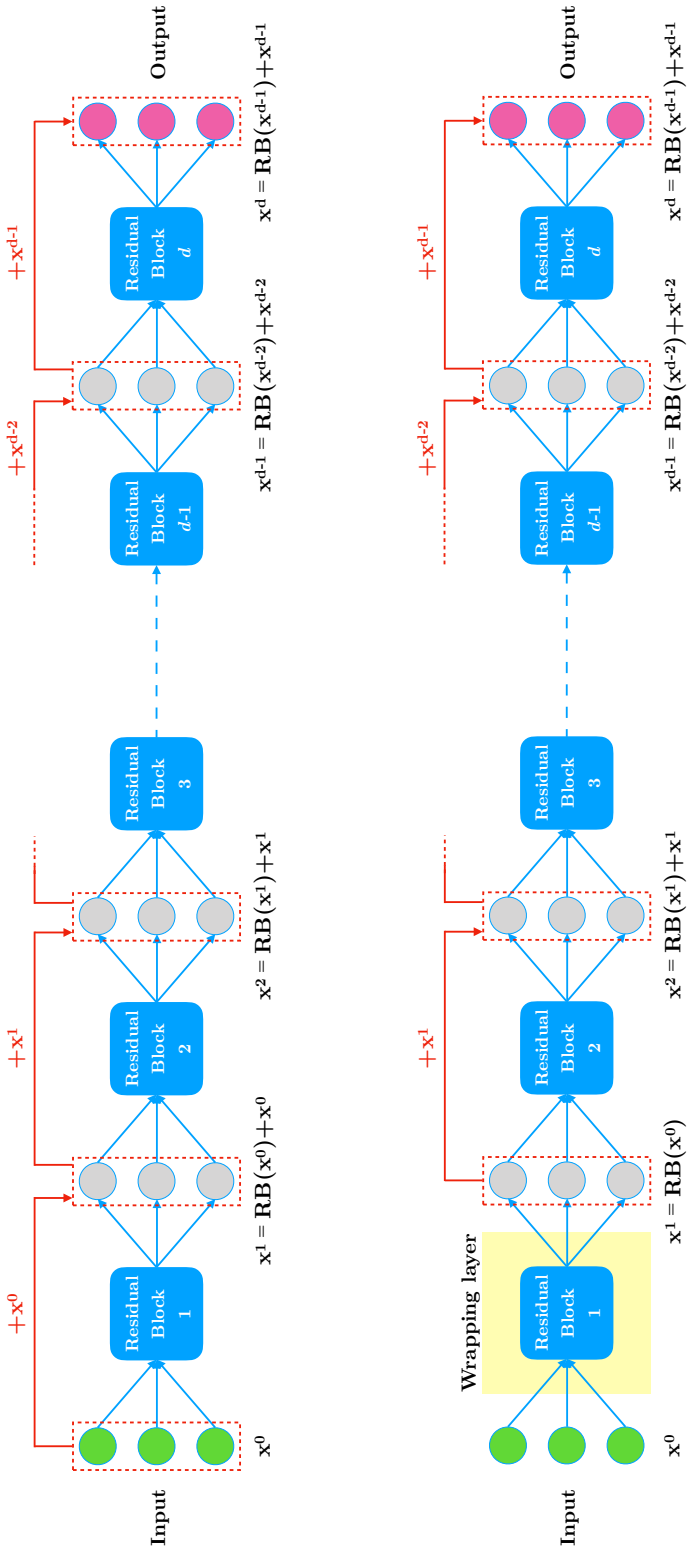


Figure 2.5: General Representation of a ResNet. The top network is represented without a wrapping layer whereas the bottom network is represented with a wrapping layer.

2.3.2 Mathematical description

Consider $E \subset \mathbb{R}^n$ a normed vector space and $\mathbf{RB}$: $E \to \mathbb{R}^n$ a real-valued function, regular enough. If $\mathbf{RB}$ is introduced in the residual scheme of Figure 2.5, any $\mathbf{x}^k$, $0 \leq k \leq d$, can be written as

$$\mathbf{x}^k = (\mathbf{Id} + \mathbf{RB})^k(\mathbf{x}^0), \tag{2.30}$$

where for $1 \leq k \leq d$,

$$(\mathbf{Id}+\mathbf{RB})^k = \prod_{i=1}^{k}(\mathbf{Id}+\mathbf{RB}) := \underbrace{(\mathbf{Id} + \mathbf{RB}) \circ (\mathbf{Id} + \mathbf{RB}) \circ \cdots \circ (\mathbf{Id} + \mathbf{RB})}_{k \text{ times}}, \tag{2.31}$$

and $(\mathbf{Id} + \mathbf{RB})^0 = \mathbf{Id}$. The residual block operator $\mathbf{RB}$ is simply the operator Ψ introduced Equation 2.7. The output $\mathbf{y}$ of the ResNet is:

$$\mathbf{y} = \mathbf{x}^d = \left(\prod_{i=1}^{d} \Psi + \mathbf{Id}\right)(\mathbf{x}^0) = \left(\prod_{i=1}^{d} \mathbf{RB} + \mathbf{Id}\right)(\mathbf{x}^0). \tag{2.32}$$

Furthermore, for any k, $0 \leq k \leq d-1$:

$$\mathbf{x}^{k+1} - \mathbf{x}^k = \mathbf{RB}(\mathbf{x}^k), \tag{2.33}$$

and, for any arbitrary $h > 0$:

$$\frac{\mathbf{x}^{k+1} - \mathbf{x}^k}{h} = h^{-1}\mathbf{RB}(\mathbf{x}^k) \Leftrightarrow \dot{\mathbf{x}}^k = \tilde{\mathbf{RB}}(\mathbf{x}^k), \tag{2.34}$$

with $\tilde{\mathbf{RB}} := h^{-1}\mathbf{RB}$ and using Euler discretization. This offers a new perspective: while FF NNs provide a complex nonlinear mapping between input $\mathbf{x}^0$ and output $\mathbf{x}^d$, ResNets offer a mapping built upon a differential equation between layers [22]. Therefore, topological spaces accessible from input $\mathbf{x}^0$ are different depending on the choice of the network.

2.3.3 Dimension and dimensionality

The network on the top Figure 2.5 is a general representation of a ResNet where the dimensionality of the vector $\mathbf{x}^k$, $0 \leq k \leq d$, remains constant across the network and where the dimension of $\mathbf{RB}(\mathbf{x}^k)$ is of the same dimension as $\mathbf{x}^k$. As a result, input and output have the same dimension and dimensionality. This usually does not cause any issue as a majority of ResNets applications are dedicated to image recognition. The ResNet is used to extract features from an original image and input and output are images of the same size; consequently dimension and dimensionality are fixed. Using a ResNet when dimension and/or dimensionality of the input is different than the ones of the output requires a slight modification in the ResNet structure. Hence, a wrapping layer is introduced. The general representation of a ResNet with a wrapping layer is also presented in Figure 2.5 (bottom network). This wrapping layer, which can be placed anywhere in the network has the role of accomodating and matching the dimension and/or dimensionality of the input with the ones of the output. However, it is computationally beneficial to introduce it as a first layer if the input's dimension is lower than the output.

2.4 The Deep Residual Neural Network

Derived from the Residual neural network, the Deep Residual neural network (DeepResNet) is a special neural network structure that combines both the FF and ResNet architectures. Starting from a basic ResNet, the DeepResNet has the special feature of having several residual blocks per layer. Similarly as in the FF, the input is sent to the first residual blocks that constitute the first layer. The layers can be seen as several residual blocks stacked on top of each other as presented in Figure 2.6 or more simply as a classical hidden layer in a FF neural network. However, the difference with a FF network is that the output of the previous layer is sent without any transformation to the the output of the next layer. The DeepResNet is then very similar to a FF with the special feature of the ResNet. It should be noted that the last skip connection that sends the output of layer $d-1$ to the output of layer d (e.g. the output of the network) has to be wrapped to respect the dimensionnality. Again, a wrapping layer is introduced to respect the dimension of the quantities considered.

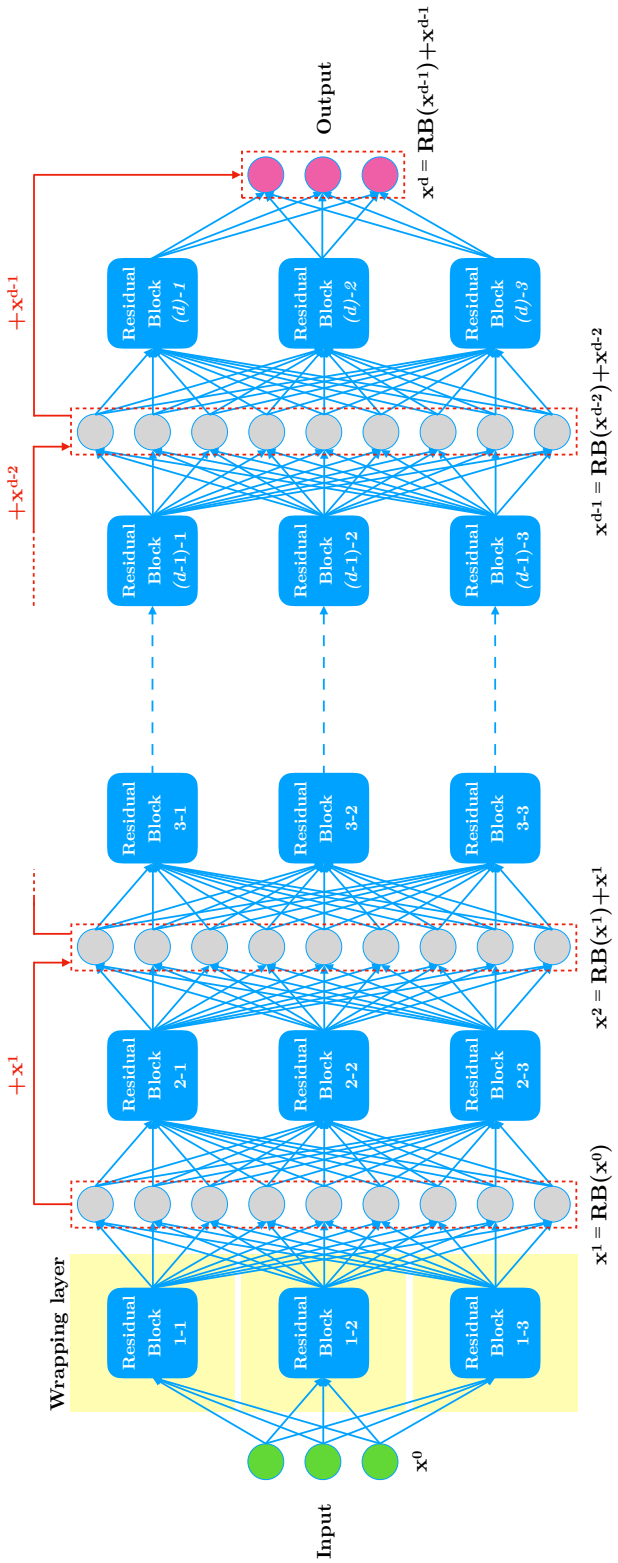


Figure 2.6: General Representation of a DeepResNet with a wrapping layer.

Chapter 3
General Methodology

3.1 The Two-body Problem

3.1.1 History

Identification and control of nonlinear dynamical systems using neural networks were introduced by Narendra and Parthasarathy in 1990 [1] based on observed simulation results. With rapid progress, neural networks for identification of unknown dynamical systems [2] have been subjected to a broad and comprehensive consideration for long-term prediction capabilities. Data driven models have been used for centuries in dynamical system modeling especially in astrodynamics, where sometimes only input-output observations are available. Kepler's laws, Newton's laws of motion and Newton's gravitational law were developed with critical reliance on observational data. One of the most important problems in classical mechanics is the well known two-body problem, which describes the motion of two point particles (or two uniform spherical masses) subject to a central force. Since Johannes Kepler first formulated the laws that describe planetary motion early in the 17th century, many scientists endeavored to solve for the equation of motion of the planets. Newton worked on the special case of the three-body problem and realized that any longitude on Earth could be determined knowing the Moon's position. Laplace and Poincaré, based on Euler's work, dealt with stability issues using series expansions just as in the time of Newton and Leibniz and the discovery of the calculus. While the general process to solve the two-body problem in the three spatial dimensions takes advantage of some of the most essential techniques in classical mechanics (decomposition of the dynamics, use of relative motion, the symmetries, reduction

of the order of the system with conservation laws), celestial mechanics are the experimental laboratory for the discovery of new mathematics. Based on Giuseppe Piazzi's observations of Ceres (1801) [23], Gauss calculated the orbit of Ceres originally using only three points. With some approximations, Gauss found inverse square law and initiated the theory of least squares and the premises of data driven models. Due to its rich history and its conservative properties, the unperturbed two-body problem is chosen to investigate the approximation and interpolation capabilities of NN-based methods. The objective is to see if NNs are able to learn the inverse square law of the gravity field and hence predict motion of a satellite in a purely predictive manner.

3.1.2 Dynamics and Mathematical Description in Cartesian Coordinates

Let $\mathbf{r}_1$ and $\mathbf{r}_2$ be the position vector of two bodies, and m_1 and m_2 be their mass. If $\mathbf{r} = \mathbf{r}_2 - \mathbf{r}_1$ is the relative position vector between the two bodies, the dynamics of the two-body problem are given by

$$\ddot{\mathbf{r}} = -\frac{\mu}{r^3}\mathbf{r}, \tag{3.1}$$

with $\mu = G(m_1 + m_2)$ and G is the universal gravitational constant. In an inertial reference frame and using Cartesian coordinates, with $\mathbf{r} = \begin{bmatrix} x & y & z \end{bmatrix}^T$ and $r = \sqrt{x^2 + y^2 + z^2}$, (3.1) can be written as

$$\begin{aligned} \ddot{x} &= -\frac{\mu x}{r^3}, \\ \ddot{y} &= -\frac{\mu y}{r^3}, \\ \ddot{z} &= -\frac{\mu z}{r^3}. \end{aligned} \tag{3.2}$$

From (3.1) and (3.2), we define the function $\mathbf{f} : \mathbb{R}^3 \to \mathbb{R}^3$ as

$$\mathbf{f} : \mathbf{r} \mapsto \ddot{\mathbf{r}} \Leftrightarrow \mathbf{f} : \begin{bmatrix} x \\ y \\ z \end{bmatrix} \mapsto \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = -\frac{\mu}{r^3} \begin{bmatrix} x \\ y \\ z \end{bmatrix}. \tag{3.3}$$

3.1.3 General Solution of the Two-body Problem

The idea is to investigate whether a certain model of neural network has the potential to approximate the function $\mathbf{f}$ introduced in (3.3) and therefore to determinate if such a model has the capacity to interpret and portray the underlying dynamics embedded in some data set.

Consider the initial value problem as follows:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{F}(\mathbf{x}, t); \\ \mathbf{x}(t_0) &= \mathbf{x}_0, \end{aligned} \tag{3.4}$$

where $\mathbf{x} = \begin{bmatrix} \mathbf{r} & \dot{\mathbf{r}} \end{bmatrix}^T \in \mathbb{R}^6$ is the unknown state vector of time t that is to be approximated. $\mathbf{F} : \mathbb{R}^6 \to \mathbb{R}^6$ contains the dynamics $\mathbf{f}$ of the two-body problem in the six-dimensional space of position and velocity. A solution of (3.4) is

$$\mathbf{x}(t) = \mathbf{x}_0 + \int_{t_0}^{t} \mathbf{F}(\mathbf{x}(\tau))\mathrm{d}\tau, \tag{3.5}$$

where a numerical integrator is used to compute the second term of (3.5).
One can reformulate the two-body problem in Cartesian coordinates and write the dynamics using a pseudo-matrix form

$$\mathbf{F}(\cdot) = \begin{bmatrix} \mathbf{0}_{3\times3}(\cdot) & \mathbf{I}_{3\times3}(\cdot) \\ \mathbf{f}(\cdot) & \mathbf{0}_{3\times3}(\cdot) \end{bmatrix} \tag{3.6}$$

such that

$$\dot{\mathbf{x}} = \mathbf{F}(\mathbf{x}, t) \Leftrightarrow \begin{bmatrix} \dot{\mathbf{r}} \\ \ddot{\mathbf{r}} \end{bmatrix} = \begin{bmatrix} \mathbf{0}_{3\times3}(\cdot) & \mathbf{I}_{3\times3}(\cdot) \\ \mathbf{f}(\cdot) & \mathbf{0}_{3\times3}(\cdot) \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \dot{\mathbf{r}} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{r}} \\ \mathbf{f}(\mathbf{r}) \end{bmatrix}, \tag{3.7}$$

where the matrix product has to be seen as a composition.

3.1.4 Integrated Solution by the Neural Network

Because the dynamics are to be approximated by a neural network, (3.7) becomes

$$\dot{\mathbf{x}} = \tilde{\mathbf{F}}(\mathbf{x}, t) \Leftrightarrow \begin{bmatrix} \dot{\mathbf{r}} \\ \ddot{\mathbf{r}} \end{bmatrix} = \begin{bmatrix} \mathbf{0}_{3\times3}(\cdot) & \mathbf{I}_{3\times3}(\cdot) \\ \mathbf{NN}(\cdot) & \mathbf{0}_{3\times3}(\cdot) \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \dot{\mathbf{r}} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{r}} \\ \mathbf{NN}(\mathbf{r}) \end{bmatrix}, \tag{3.8}$$

where $\mathbf{f}$ has been replaced by the neural network model ($\mathbf{NN}$).
Consider now $\tilde{\mathbf{F}} : \mathbb{R}^6 \to \mathbb{R}^6$ that contains a neural network to approximate the dynamics instead of the function $\mathbf{f}$ from (3.3). The neural network replaces the function $\mathbf{f}$. Denoted with a tilde $\tilde{\mathbf{x}}(t)$ to indicate that the solution comes from a neural network approximation, (3.5) becomes

$$\tilde{\mathbf{x}}(t) = \mathbf{x}_0 + \int_{t_0}^{t} \tilde{\mathbf{F}}(\tilde{\mathbf{x}}(\tau))\mathrm{d}\tau. \tag{3.9}$$

While the neural network takes over the dynamical model, a well chosen integrator will then have the role to finally integrate the general solution. The integrator implemented is a fourth order Runge-Kutta fixed step-size algorithm. Subsequently, the accuracy of the neural network is measured by calculating the difference between the integrated solution using the neural network model, $\tilde{\mathbf{x}}(t)$, and the true solution, $\mathbf{x}$, with the true dynamics $\mathbf{F}$ and a Dormand-Prince integration algorithm. Figure 3.1 displays how the solution from the neural network and the true solution are generated to compute the Loss function.

3.1.5 Runge-Kutta Integration Scheme with the Neural Network

For any step-size $h > 0$ and $\forall n \in \mathbb{N}$, we define $\mathbf{x}_{n+1}$ as the RK4 (Runge-Kutta 4th-order) approximation of $\mathbf{x}(t_{n+1})$ by

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \frac{1}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right), \tag{3.10}$$

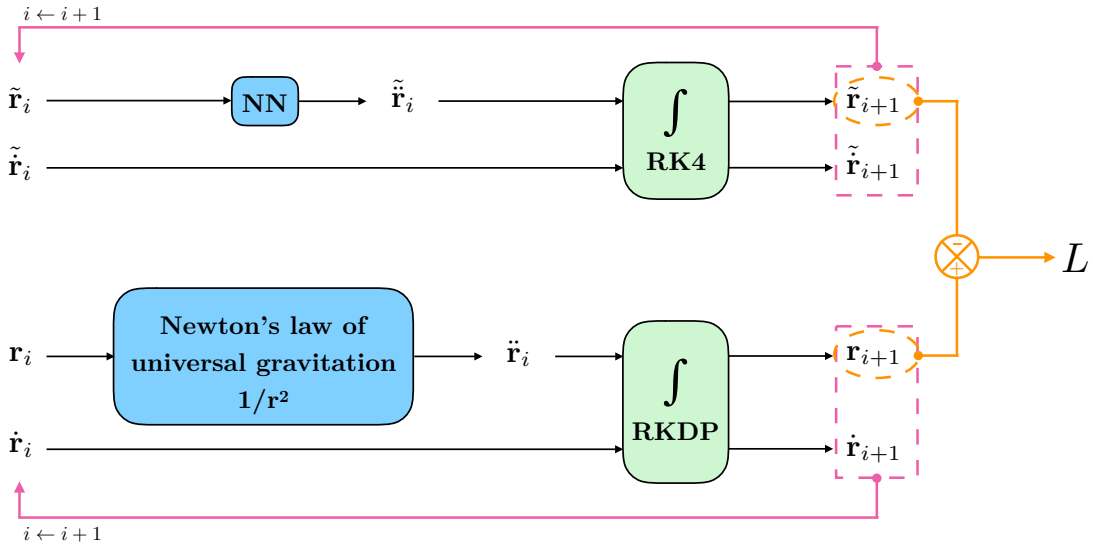


Figure 3.1: How the neural network is used: the upper part illustrates Eq. 3.9 with the neural network used to approximate the dynamics along with the Runge-Kutta fixed-size step algorithm while the bottom part is the classical generation of the true solution with the known dynamics and a Dormand-Prince integration algorithm.

with

$$
\begin{aligned}
\mathbf{k}_1 &= h\mathbf{F}(\mathbf{x}_n, t_n); \\
\mathbf{k}_2 &= h\mathbf{F}\left(\mathbf{x}_n + \frac{\mathbf{k}_1}{2}, t_n + \frac{h}{2}\right); \\
\mathbf{k}_3 &= h\mathbf{F}\left(\mathbf{x}_n + \frac{\mathbf{k}_2}{2}, t_n + \frac{h}{2}\right); \\
\mathbf{k}_4 &= h\mathbf{F}\left(\mathbf{x}_n + \mathbf{k}_3, t_n + h\right).
\end{aligned}
\tag{3.11}
$$

If a neural network model is used to replace the true dynamics, $\mathbf{F}$ is replaced by $\tilde{\mathbf{F}}$ and using 3.8 it is possible to define the pairs $(\mathbf{p}_i, \mathbf{q}_i)_{1 \leq i \leq 4} \in \mathbb{R}^3 \times \mathbb{R}^3$ as

$$
\begin{aligned}
\mathbf{p}_1 &= h\dot{\mathbf{r}}_n; \\
\mathbf{q}_1 &= h\mathbf{NN}(\mathbf{r}_n); \\
\mathbf{p}_2 &= h\left(\frac{\mathbf{q}_1}{2} + \dot{\mathbf{r}}_n\right); \\
\mathbf{q}_2 &= h\mathbf{NN}\left(\frac{\mathbf{p}_1}{2} + \mathbf{r}_n\right); \\
\mathbf{p}_3 &= h\left(\frac{\mathbf{q}_2}{2} + \dot{\mathbf{r}}_n\right); \\
\mathbf{q}_3 &= h\mathbf{NN}\left(\frac{\mathbf{p}_2}{2} + \mathbf{r}_n\right); \\
\mathbf{p}_4 &= h\left(\mathbf{q}_3 + \dot{\mathbf{r}}_n\right); \\
\mathbf{q}_4 &= h\mathbf{NN}\left(\mathbf{p}_3 + \mathbf{r}_n\right).
\end{aligned}
\tag{3.12}
$$

for the Runge-Kunta integration scheme. Finally, the RK4 approximation of $\mathbf{x}(t_{n+1})$ using the neural network model as a substitute for the dynamics (written with a tilde $\tilde{\mathbf{x}}(t_{n+1})$ to specify that it comes from an approximation) is

$$
\tilde{\mathbf{x}}(t_{n+1}) = \tilde{\mathbf{x}}_{n+1} = \tilde{\mathbf{x}}_n + \frac{1}{6}\left(\begin{bmatrix}\mathbf{p}_1\\ \mathbf{q}_1\end{bmatrix} + 2\begin{bmatrix}\mathbf{p}_2\\ \mathbf{q}_2\end{bmatrix} + 2\begin{bmatrix}\mathbf{p}_3\\ \mathbf{q}_3\end{bmatrix} + \begin{bmatrix}\mathbf{p}_4\\ \mathbf{q}_4\end{bmatrix}\right). \tag{3.13}
$$

Figure 3.2 presents an expanded view of the Runge-Kutta integration scheme implemented along with the neural network. As a fourth-order integrator, the Runge-Kutta algorithm makes use of the neural network at each integration step four times.

3.1.6 Constants of the Motion

Quite apart from whether a specific structure of NN can learn the inherent dynamics of the two-body problem, it is critical to assess the capabilities of such a model to recover the constants of the motion, namely the angular momentum and total energy.

Let be $\mathbf{h}_{2/1}$ the angular momentum of body m_2 relative to m_1 is

$$
\mathbf{h}_{2/1} = \mathbf{r} \times m_2\dot{\mathbf{r}} \tag{3.14}
$$

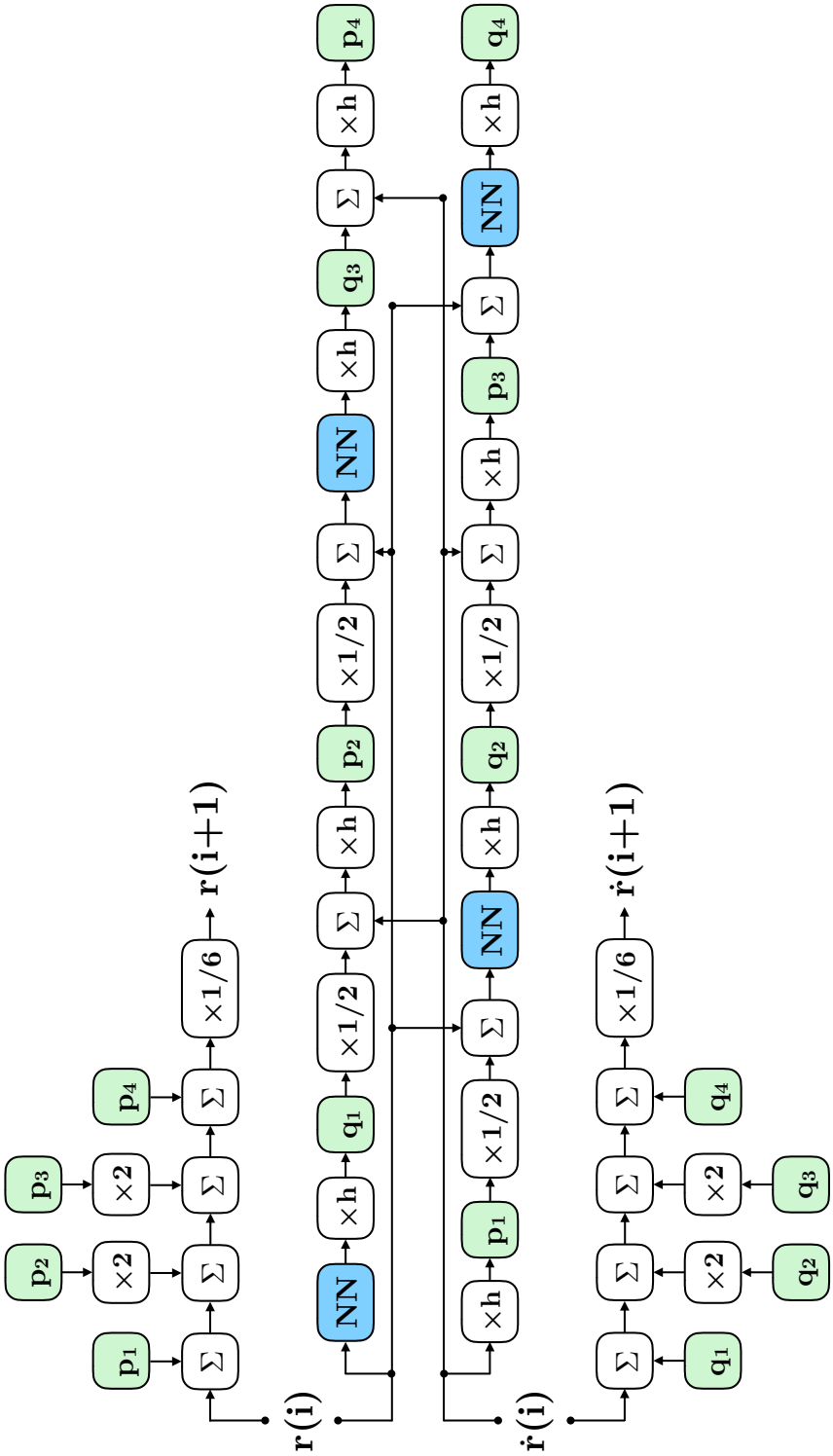


Figure 3.2: 4th order Runge-Kutta integration scheme using a neural network model to approximate the dynamics.

where $\dot{\mathbf{r}}$ is the relative velocity of m_2 with respect to m_1. Let be $\mathbf{h} = \mathbf{h}_{2/1}/m_2$ the specific relative angular momentum ($\mathbf{h}$ is the relative angular momentum of m_2 per unit mass):

$$\mathbf{h} = \mathbf{r} \times \dot{\mathbf{r}}. \tag{3.15}$$

Taking the time derivative of $\mathbf{h}$ yields

$$\frac{\mathrm{d}\mathbf{h}}{\mathrm{d}t} = \frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} \times \dot{\mathbf{r}} + \mathbf{r} \times \frac{\mathrm{d}\dot{\mathbf{r}}}{\mathrm{d}t} = \dot{\mathbf{r}} \times \dot{\mathbf{r}} + \mathbf{r} \times \ddot{\mathbf{r}} = \mathbf{0} - \frac{\mu}{r^3}\mathbf{r} \times \mathbf{r} = \mathbf{0}. \tag{3.16}$$

Therefore, angular momentum is conserved. By conservation of angular momentum, the path of m_2 relative to m_1 lies in a single plane.
Furthermore, from 3.1,

$$\ddot{\mathbf{r}} \cdot \dot{\mathbf{r}} = \frac{\mu}{r^3}\mathbf{r} \cdot \dot{\mathbf{r}}. \tag{3.17}$$

For the left-hand side,

$$\ddot{\mathbf{r}} \cdot \dot{\mathbf{r}} = \frac{1}{2}\frac{\mathrm{d}(\dot{\mathbf{r}} \cdot \dot{\mathbf{r}})}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{v^2}{2}\right). \tag{3.18}$$

For the right-hand side,

$$\frac{\mu}{r^3}\mathbf{r} \cdot \dot{\mathbf{r}} = \frac{\mu}{r^3} r\dot{r} = -\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mu}{r}\right). \tag{3.19}$$

Substituting in 3.17,

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{v^2}{2} - \frac{\mu}{r}\right) = 0 \quad \text{or} \quad \frac{\mathrm{d}e}{\mathrm{d}t} = 0, \tag{3.20}$$

where

$$e = \frac{v^2}{2} - \frac{\mu}{r} \tag{3.21}$$

is the total mechanical energy per unit mass and is constant at all points of the trajectory.
Historically, the angular momentum is known as the vector $\mathbf{C}_3$ and is the normal to the "invariable plane" defined by Laplace and the total mechanical energy is known as the scalar C_4 and is the tenth constant of the motion. The equations of motion of a system of n bodies consist of $3n$ second-order differential equations which are equivalent to $6n$ first-order differential equations. Thus $6n$ integrals of the motion

are required to solve for the general absolute motion of a n-body system. Since only 10 constants of the motion exist, the motion of a system of two-bodies cannot be determined in closed form, hence the numerical integration.

3.1.7 The Loss Function: from basic expression to incorporating soft constraints

Overall, since the Loss is an indicator of the fitness capabilities of the network, it will compare the true value of the states with the approximations provided by the network. According to 2.13 and 2.15, for a training set $\mathcal{S}$ of size N, the Loss function is

$$L = \sum_{k=1}^{N} \left(\frac{1}{3} \left|\left|\mathbf{r_k} - \tilde{\mathbf{r}}_k\right|\right|_2^2 \right). \tag{3.22}$$

When approximating the dynamics of the unperturbed two-body problem (e.g. conservative system), the constants of the motion can be seen as constraints. Constraints can be either hard constraints, which set some conditions for the model that are required to be satisfied, or soft constraints, which results in penalizing the Loss function if a deviation from constancy is observed. The unperturbed two-body problem is a constrained problem and the phase of training the neural network by taking into account the constants of the motion as soft constraints becomes a constraint optimization problem. In order to evaluate the prediction accuracy gained by considering these constraints, we provide the network with the context of inherent dynamics by incorporating a penalty term in the Loss function corresponding to violations of conservation of angular momentum as well as total energy. Given an initial state $\mathbf{x}_0 = \begin{bmatrix} \mathbf{r}_0 & \mathbf{v}_0 \end{bmatrix}^T$, we define initial angular momentum, $\mathbf{h}_0 = \mathbf{r}_0 \times \mathbf{v}_0$ and initial energy, $e_0 = v_0^2/r_0 - \mu/r_0$. Now, the Loss function can be modified as follows to account for conservation of angular momentum and energy:

$$L = \sum_{k=1}^{N} \left(\frac{1}{3} \left|\left|\mathbf{r_k} - \tilde{\mathbf{r}}_k\right|\right|_2^2 + \underbrace{\frac{1}{3} \left|\left|\mathbf{h}_0 - \tilde{\mathbf{h}}_k\right|\right|_2^2 + \left|\left|e_0 - \tilde{e}_k\right|\right|_2^2}_{\text{soft constraints}} \right). \tag{3.23}$$

Note that inclusion of angular momentum and total energy in the Loss function is equivalent to have pre-conditioning terms.

3.1.8 In Polar Coordinates

There is no doubt that the ability of NNs to learn the dynamic behavior also depends upon the mathematical representation of the physical system. This is due to the fact that nonlinearity is not an inherent attribute of a physical system, but rather dependent upon the mathematical description of the system's behavior [25]. Ideally, one has an infinity of coordinate choices to represent the same physical system. In the study of celestial mechanics, the quest to find a judicious coordinate system led to the development of various canonical transformations. To study the effect of mathematical representation on a NN's accuracy, it is relevant to consider the approximation of two-body dynamics in polar coordinates, where the external force field is only in the radial direction. The objective is to see if NNs accurately identify this hidden mathematical structure from orbital observations. From Eq. 3.1, the dynamics of the two-body problem expressed in polar coordinates is

$$\begin{aligned} \ddot{r} &= -\frac{\mu}{r^2} + r\dot{\theta}^2, \\ \ddot{\theta} &= -\frac{2\dot{r}\dot{\theta}}{r}. \end{aligned} \tag{3.24}$$

From Eq. 3.24, we define the function $\mathbf{g} : \mathbb{R}^2 \to \mathbb{R}^2$ as

$$\mathbf{g} : \begin{bmatrix} r \\ \theta \end{bmatrix} \mapsto \begin{bmatrix} \mathbf{g}_r(r,\theta) \\ \mathbf{g}_\theta(r,\theta) \end{bmatrix} = \begin{bmatrix} -\mu/r^2 \\ 0 \end{bmatrix}. \tag{3.25}$$

that contains the dynamics of the central gravitational force in polar coordinates. One can reformulate and write the full dynamics equations using a pseudo-matrix form

$$\mathbf{G}(\cdot) = \begin{bmatrix} \mathbf{0}_{2\times 2}(\cdot) & \mathbf{I}_{2\times 2}(\cdot) \\ \mathbf{g}(\cdot) & \mathbf{0}_{2\times 2}(\cdot) \end{bmatrix} \tag{3.26}$$

such that if $\mathbf{x} = \begin{bmatrix} r & \theta & \dot{r} & \dot{\theta} \end{bmatrix}^T$, then

$$\dot{\mathbf{x}} = \mathbf{G}(\mathbf{x}) + \mathbf{h}(\mathbf{x}) \Leftrightarrow \begin{bmatrix} \dot{r} \\ \dot{\theta} \\ \ddot{r} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \mathbf{0}_{2\times2}(\cdot) & \mathbf{I}_{2\times2}(\cdot) \\ \mathbf{g}(\cdot) & \mathbf{0}_{2\times2}(\cdot) \end{bmatrix} \begin{bmatrix} r \\ \theta \\ \dot{r} \\ \dot{\theta} \end{bmatrix} + \mathbf{h}(\mathbf{x}) = \begin{bmatrix} \dot{r} \\ \dot{\theta} \\ -\mu/r^2 \\ 0 \end{bmatrix} + \mathbf{h}(\mathbf{x}), \tag{3.27}$$

where the matrix product has to be seen as a composition and $\mathbf{h} : \mathbb{R}^4 \to \mathbb{R}^2$

$$\mathbf{h} : \begin{bmatrix} r \\ \theta \\ \dot{r} \\ \dot{\theta} \end{bmatrix} \mapsto \begin{bmatrix} r\dot{\theta}^2 \\ -\dfrac{2\dot{r}\dot{\theta}}{r} \end{bmatrix} \tag{3.28}$$

takes into account the remaining terms in the radial and circumferential acceleration components. Similarly, if the dynamics of the central force to be approximated by a neural network, (3.27) becomes

$$\dot{\mathbf{x}} = \tilde{\mathbf{G}}(\mathbf{x}) + \mathbf{h}(\mathbf{x}) \Leftrightarrow \begin{bmatrix} \dot{r} \\ \dot{\theta} \\ \ddot{r} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \mathbf{0}_{2\times2}(\cdot) & \mathbf{I}_{2\times2}(\cdot) \\ \mathbf{NN}(\cdot) & \mathbf{0}_{2\times2}(\cdot) \end{bmatrix} \begin{bmatrix} r \\ \theta \\ \dot{r} \\ \dot{\theta} \end{bmatrix} + \mathbf{h}(\mathbf{x}) \tag{3.29}$$

where $\mathbf{g}$ has been replaced by the neural network model ($\mathbf{NN}$). Therefore, the idea is to investigate whether the part of the dynamics coming from the central force, e.g. $\mathbf{g}$, can be replaced by a NN. More precisely, it is important to examine:

1. whether the second component $\mathbf{g}_\theta(r, \theta)$ is set to 0 or not,
2. in which extent the first component $\mathbf{g}_r(r, \theta)$ is independent from θ.

Note that $\mathbf{g}_\theta(r, \theta) = 0$ corresponds directly to a constant value for angular momentum and introduce a direct preconditioner on the network parameters α.

3.1.9 Computational Details

3.1.9.1 TensorFlow Library

Researchers have been implementing software libraries to facilitate the construction of neural network architectures for decades. Until the last few years, these systems were mostly special purpose and only used within an academic group. This lack of standardized, industrial-scalable software made it difficult for non-experts to make use of machine learning tools. However, the situation has changed dramatically over the last few years. Google implemented the DistBelief system in 2012 and made use of it to construct and deploy many simpler deep learning architectures. The advent of DistBelief, and similar packages such as Caffe, Theano, Torch and Keras, MxNet and so on have widely spurred industry adoption. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, TensorFlow [24] is an open source software library based on a strong computer algebra system allowing high performance numerical computation thanks to a high level of abstraction. Tensorflow uses the concept of tensors as the fundamental underlying primitive powering deep-learning systems. This focus on tensors distinguishes these packages from systems such as DistBelief or Caffe which don't allow the same flexibility for building sophisticated models.

3.1.9.2 Poliastro Library

Poliastro is an open source collection of Python functions useful in astrodynamics and orbital mechanics, focusing on interplanetary applications. It provides a simple and intuitive API and handles physical quantities with units. Poliastro handles analytical and numerical orbit propagation, initial orbit determination, maneuvers computation, trajectory plotting and much more. Poliastro works on recent versions of Python and is released under the MIT license, hence allowing commercial use of the library.

3.1.9.3 Computing specifications

Table 3.1 gathers the hardware and software computing specifications.

Table 3.1: Hardware and Software Computing Specifications

Hardware	**Dual Xeon E5-2695v4**	2.1 Ghz
	Cores	36
	Threads	72
	RAM	192 Gb
Software	**Python**	v3.7
	Tensorflow	v1.10.0
	Poliastro	v0.11.0

Chapter 4 Numerical Results and Discussion

4.1 Method

The final objective is to investigate the approximation capabilities of the two-body Keplerian dynamics with a multi-layer neural network coupled with a fourth order Runge-Kutta integrator. From what has been discussed previously, these approximation capabilities depend on the framework chosen. This work considers the influence of the training set, the expression of the Loss function, the design of the network and the choice of problem coordinates to explore the learning capabilities of neural networks.

It is crucial to specify that all the NNs used for training and then system identification are used in a recurrent manner. Indeed, during the identification phase after training, only one true initial condition is given to the NN model at initial time t_0. Prediction at time t_1 is directly used as input for the NN to provide a prediction at time t_2 and so on. All the structures considered are used in this recurrent scheme and a true solution is never injected afterwards to correct the NN model prediction sequence.

Furthermore, it is very common and highly recommended to work with scaled data when using NNs. First, weights and biases are randomly initialized using a standard Gaussian distribution with zero mean and unit variance. Second, nonlinear activation functions inside the networks are mostly bounded between -1 and 1; thus the values before the last layer are between these bounds. It is then a common practice

to have input and output data around these values. It would be inconceivable to hope for some output orders of magnitude higher than those of the network itself. Hence, all the lengths are in Length Unit (LU) which corresponds to the Earth radius. Time Unit (TU) is calculated such that $\mu = 1$ LU^3/TU^{-2}. This offers the comfort to use input close to 1 and output between -1 and 1.

4.2 Initial Neural Network Architecture and Study on the Training Set

This first study focuses on the impact of the training set and first involves a training with a reference near-circular Low-Earth Orbit (LEO) with semi-major axis and eccentricity

$$a = 7172490\text{m} \quad \text{and} \quad e = 0.0011. \tag{4.1}$$

The initial conditions for the reference orbit in Cartesian space are given as follows:

$$x_0 = 757700\text{m},\ y_0 = 5222607\text{m},\ z_0 = 4851500\text{m}; \tag{4.2}$$

$$\dot{x}_0 = 2213.21\text{m}\,\text{s}^{-1},\ \dot{y}_0 = 4678.34\text{m}\,\text{s}^{-1},\ \dot{z}_0 = -5371.30\text{m}\,\text{s}^{-1}. \tag{4.3}$$

A RK4 integrator with 0.01s time step is used to predict position and velocity vectors over an orbit while using a NN model for two-body dynamics approximation and true initial conditions. True orbit data is assumed to be available every 0.01s to compute the Loss function. The Loss function for this study is the one given without any soft constraint:

$$L = \sum_{k=1}^{N} \left(\frac{1}{3} \left|\left|\mathbf{r_k} - \tilde{\mathbf{r}}_k\right|\right|_2^2 \right). \tag{4.4}$$

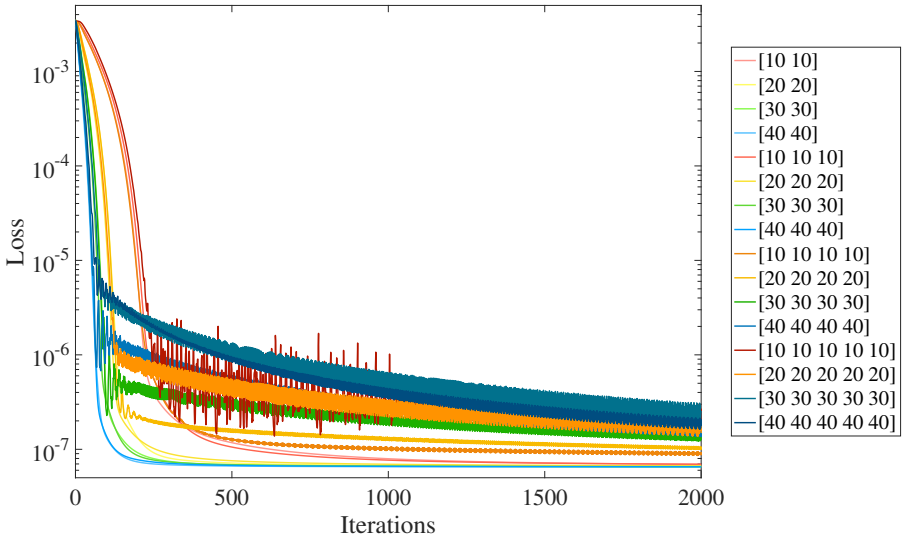
To assess the effect of different NN architectures (layers and neurons per layer) on the approximation accuracy, we initiate the study by considering sixteen distinct configurations with two to five layers and the number of neurons varying from ten to forty per hidden layer. For two thousand iterations (or epochs e.g. the number of times the entire training data set will flow through the network), we examine closely the progression of the Loss function and more particularly the monotonicity, its final value and the rate of convergence. Figure 4.1(a) presents the

evolution of the Loss function for the sixteen different configurations. Although all the architectures manage to provide an acceptable network, the Loss function does not converge for all considered NN architectures, and varies from configuration to configuration. Networks with four layers or more (eight configuration total as seen in Figure 4.1(b)) rapidly start oscillating. This non-monotonicity implies that the optimizer is not able to update the weights at each iteration to continuously decrease the overall value of the Loss function. This can happen if there are too many parameters to optimize: some updated weights or bias values result in a smaller Loss while others are responsible for an increase. This analysis allows us to draw two conclusions:

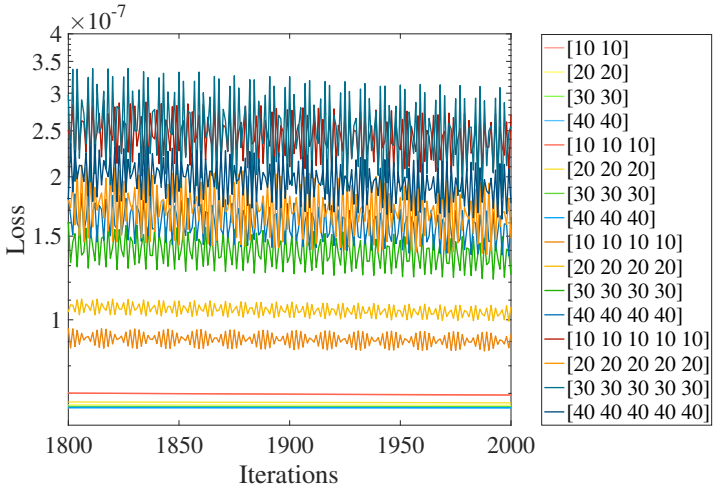
1. more layers and/or more neurons per layer don't always mean better accuracy;
2. optimizers with neural networks that are too deep may not converge monotonically (even if they do not diverge), and end up with a fuzzy behavior.

Based upon these results, we use a NN with three layers and thirty neurons per hidden layer.

Figure 4.2(a) shows the evolution of the Loss function for three layers with thirty neurons per layer FF [30-30-30] trained with orbital data for 1.6 orbit time period. It is apparent that the Loss function reaches a plateau around 3×10^{-8} after approximately 600 epochs. Figure 4.2(b) shows the true and NN approximated reference orbit plots for 4 revolutions. Figures 4.3(a) and 4.3(b) show the norm of the approximation error for position and velocity variables, respectively. Similarly, Figure 4.3(c) shows the plot of error in reproducing constants of the motion, i.e., angular momentum and total energy. The vertical red line corresponds to the end of the training time period. From these plots, it is clear that although the approximation error is small for the training time period, the prediction accuracy of the NN approximated orbit dynamic model deteriorates rapidly over time when predicting subsequent revolutions.

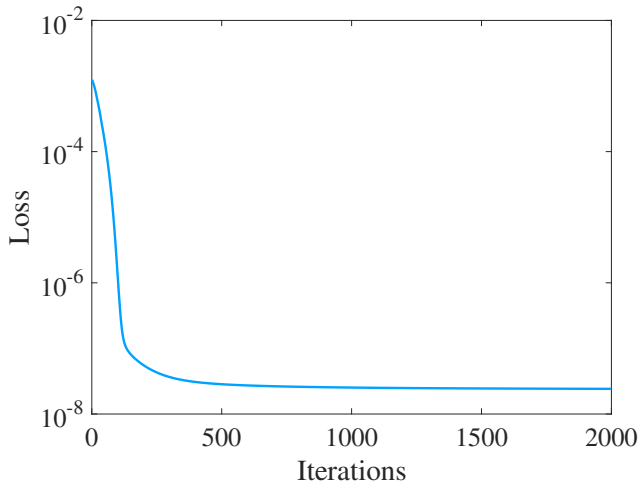


(a) Loss Function Evolution

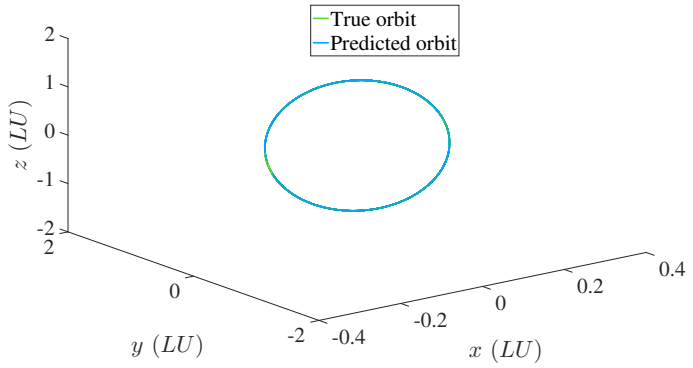


(b) Loss Function Evolution (Zoomed Version between epochs 1800 and 2000)

Figure 4.1: Evolution of the Loss function for different network architectures.

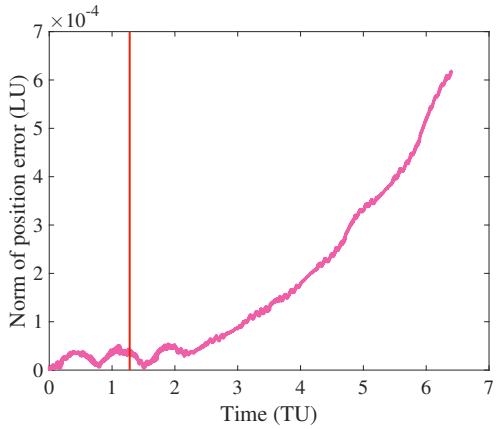


(a) Loss Function Evolution

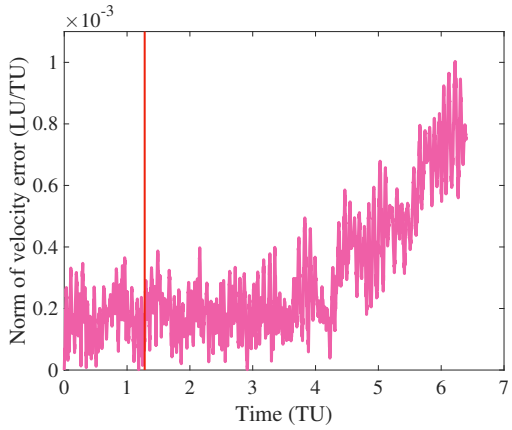


(b) True and NN Approximated Orbits

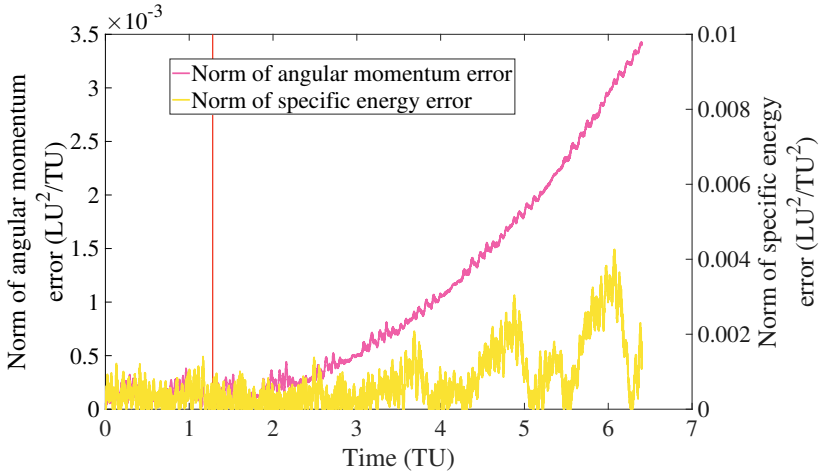
Figure 4.2: Performance of NN over training data involving time prediction



(a) Norm of position error vs. time



(b) Norm of velocity error vs. time



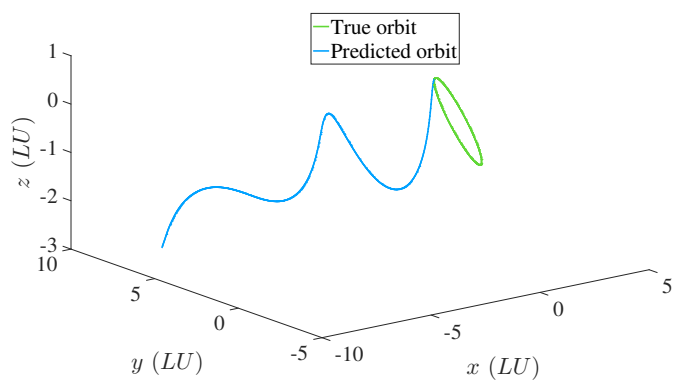
(c) Constant of the Motion Violation

Figure 4.3: Error induced by NN over training data involving time prediction

The accuracy of the same NN-based model is even worse when it is tasked to predict the motion of another orbit with semi-major axis and eccentricity

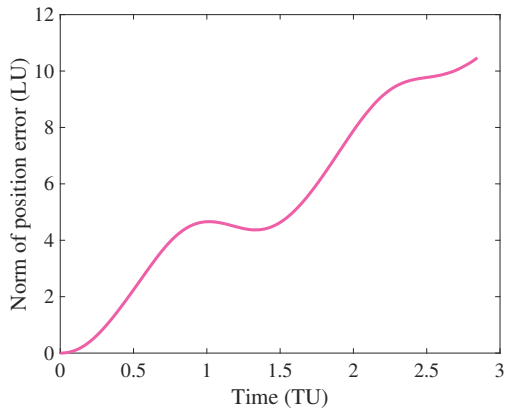
$$a = 7791108\text{m} \quad \text{and} \quad e = 0.0982, \tag{4.5}$$

(cf. Figure 4.4 and 4.5). From these results, we can conclude that the NN approximated dynamic model is unable to learn the inherent characteristics of Keplerian dynamics, i.e., conservation of energy and angular momentum and hence resulting in poor prediction capabilities.

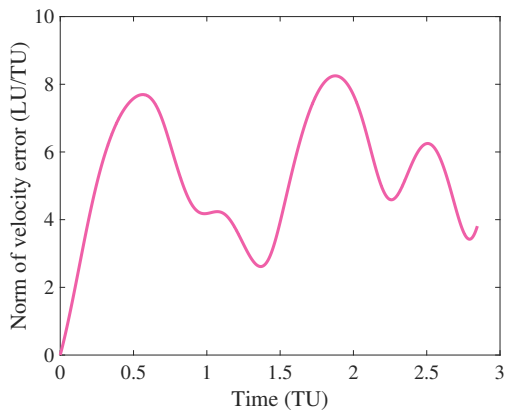


(a) True and NN Approximated Orbits

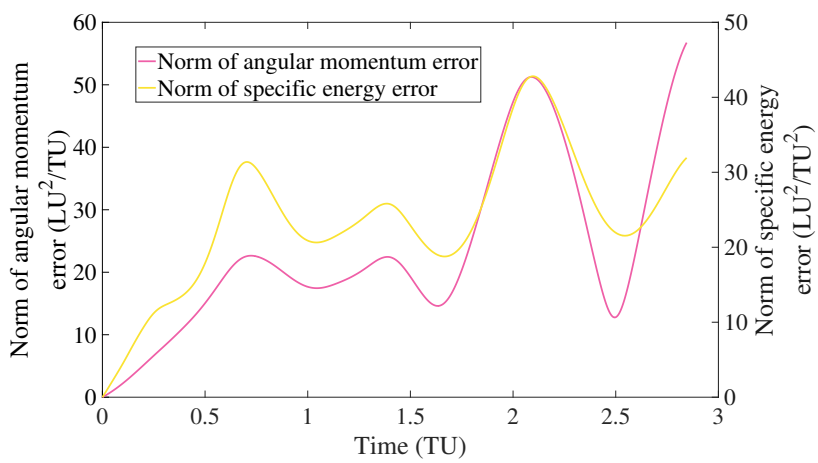
Figure 4.4: Performance of NN over unknown data involving time prediction



(a) Norm of position error vs. time



(b) Norm of velocity error vs. time



(c) Constant of the Motion Violation

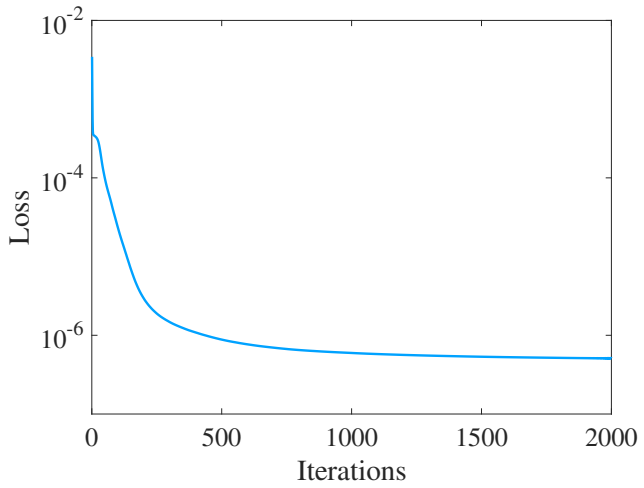
Figure 4.5: Error induced by NN over unknown data involving time prediction

To further improve the accuracy of the NN-approximated orbit model, we consider the training data comprising one revolution for ten different orbits. The data gathered from the ten orbits are stacked in a single set to produce a large training array consisting essentially of inputs with ten different initial conditions. Table 4.1 below lists the orbital elements for all 10 orbits.

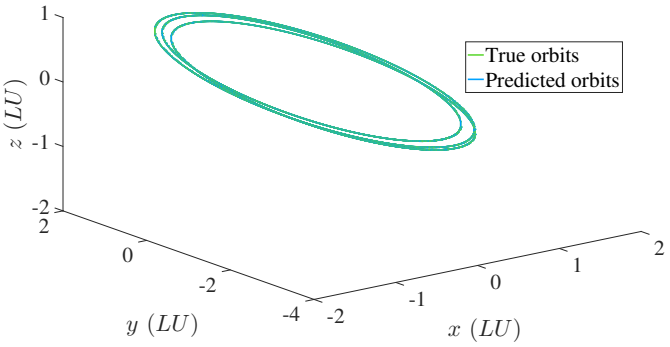
Table 4.1: Orbital elements for the ten training orbits

Semi-major axis a	10000km $\leq a \leq$ 14500km, $\Delta a = 500$km
Eccentricity e	$e = 0.2$
Inclination i	$i = \pi/6$
RAAN Ω	$\Omega = \pi/3$
Argument of perigee ω	$\omega = \pi/4$
True anomaly θ	$\theta = \pi/2$

It should be noted that the orbital elements considered correspond to all 10 orbits being coplanar. Also, for validation and testing, we consider three more orbits with the same constraints in the orbital elements as listed in Table 4.1. Figure 4.6(a) shows the plot of Loss function evolution during the NN training. As expected, the Loss function decreases continuously and converges to approximately 5×10^{-7} after 600 epochs. Because of the very large set of data, the value of the Loss function is consequently larger than the converged Loss function values in the previous test case. Figures 4.7(a) and 4.7(b) show the approximation error for position and velocity states for three orbits not included in the training. In position, the absolute error between the neural network prediction and the true value does not surpass 3×10^{-7} LU with a mean below 10^{-7} LU. Similarly, the norm of velocity approximation error does not exceed 1.2×10^{-6} LU/TU with a mean close to 2×10^{-7} LU/TU. Finally, Figures 4.8(a) and 4.8(b) show the norm of error in conservation of angular momentum and energy, respectively. The better performance of the NN in learning the orbit dynamics as compared to the previous Test Case can be attributed to the richness of the training data set.

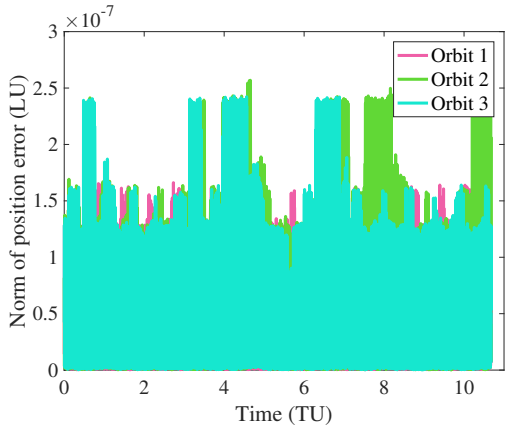


(a) Loss Function Evolution

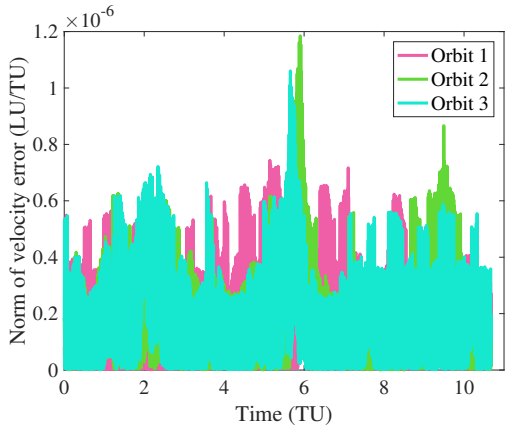


(b) True and NN Approximated Orbits

Figure 4.6: Performance of NN over unknown data involving time predicton for a large dataset

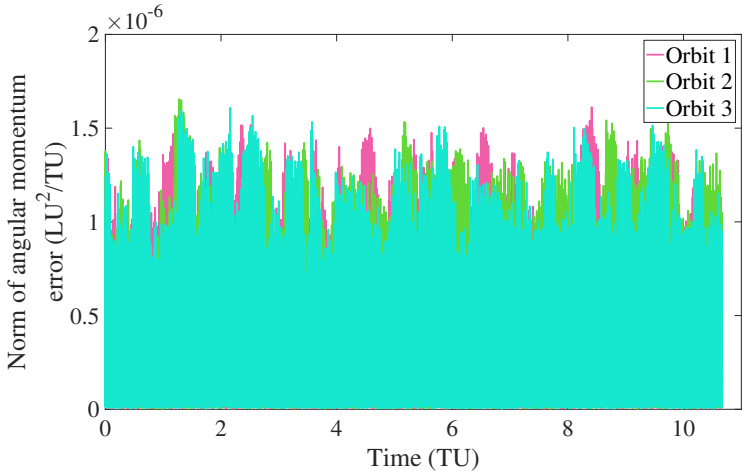


(a) Norm of position error vs. time

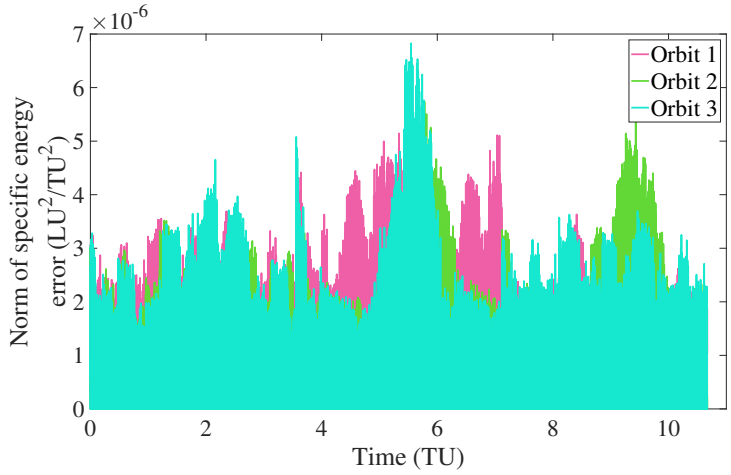


(b) Norm of velocity error vs. time

Figure 4.7: Error induced by NN over unknown data involving time predicton for a large dataset



(a) Constant of the Motion Violation



(b) Constant of the Motion Violation

Figure 4.8: Error induced by NN over unknown data involving time predicton for a large dataset

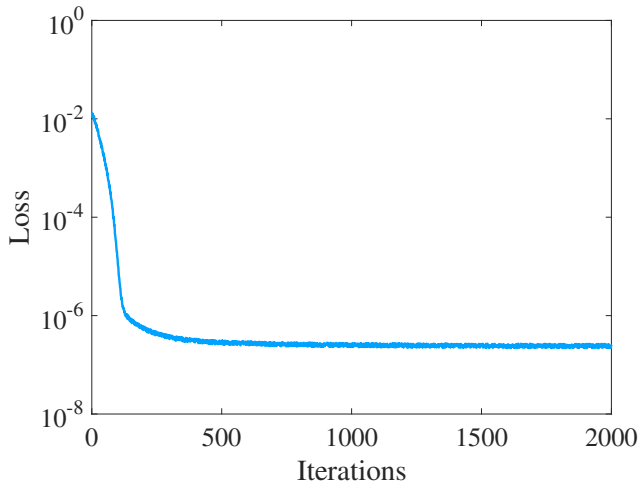
4.3 Influence of the Soft Constraints in the Loss Function

As an indicator of the fitness capabilities of the network, the Loss function compares the true value of the state with the approximation provided by the network during the training. Additionally, to measure how well the conservative motion is preserved,

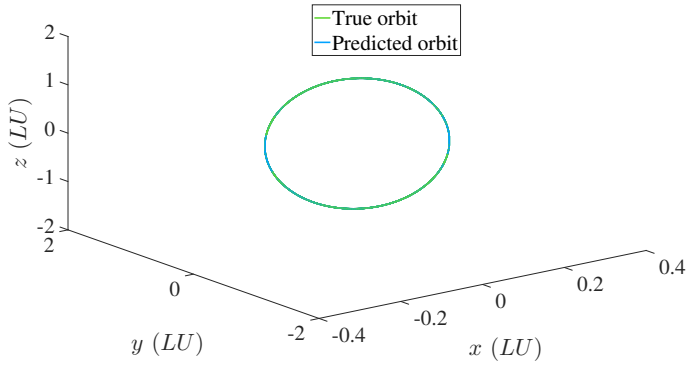
a penalty term resulting in violation of constants of the motion is introduced as a soft constraint inside the Loss function. As a result, the Loss is:

$$L = \sum_{k=1}^{N} \left(\frac{1}{3} ||\mathbf{r_k} - \tilde{\mathbf{r}}_k||_2^2 + \underbrace{\frac{1}{3} \left|\left| \mathbf{h}_0 - \tilde{\mathbf{h}}_k \right|\right|_2^2 + ||e_0 - \tilde{e}_k||_2^2}_{\text{soft constraints}} \right). \tag{4.6}$$

The evolution of the Loss function is shown in Figure 4.9(a). Notice that the converged value of the Loss function is larger than the converged Loss function in earlier training due to addition of the new terms $||\mathbf{h}_0 - \tilde{\mathbf{h}}_k||_2^2$ and $||e_0 - \tilde{e}_k||_2^2$. However, the larger value of penalty function does not necessarily mean a poor approximation capability of the NN model. Figures 4.10(a) and 4.10(b) show the norm of the approximation error for position and velocity variables, respectively. The vertical red line still corresponds to the end of the training data set. Although the approximation accuracy decreases with time prediction, the prediction accuracy of the NN learned model has increased by an order of magnitude as compared to results presented in Figures 4.3(a) and 4.3(b). This increase in prediction accuracy can be attributed to inclusion of constants of the motion during the training of NN. This observation is once again confirmed by the plot of constant of the motion violation in Figure 4.10(c). From these results, we can conclude that inclusion of a penalty term in the Loss function corresponding to violation of constants of the motion helps with NN training. However, including constants of the motion as soft constraints in the Loss function does not improve significantly the NN-based model accuracy when the prediction is performed on an unknown orbit. Figure 4.11 and 4.12 prove that this enhanced training has not produced acceptable prediction capabilities for the NN model.

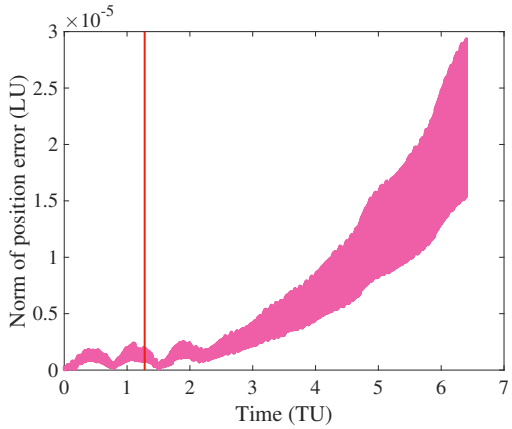


(a) Loss Function Evolution

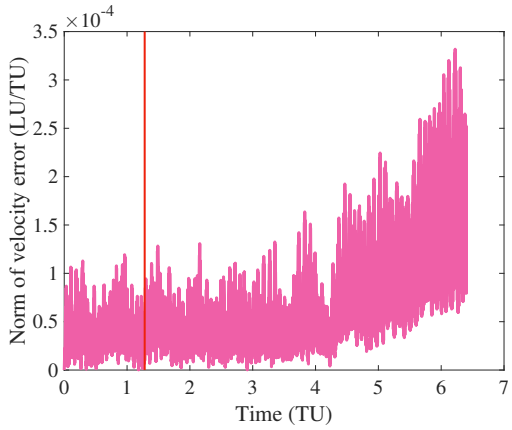


(b) True and NN Approximated Orbits

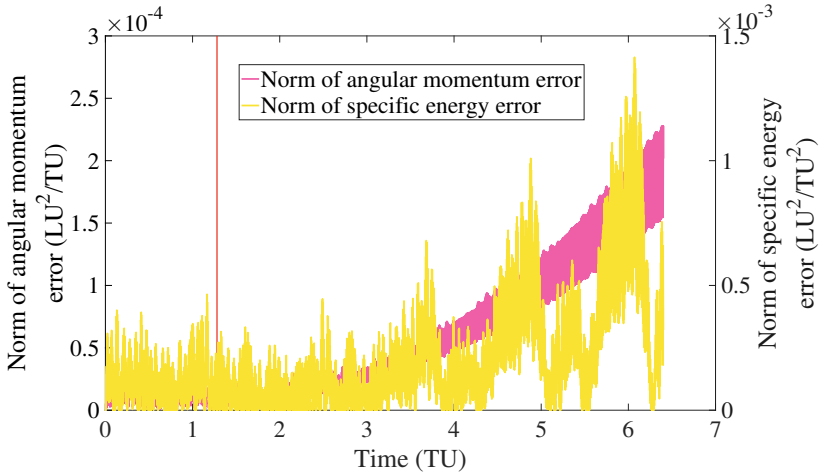
Figure 4.9: Performance of NN over training data involving time prediction with soft constraints



(a) Norm of position error vs. time

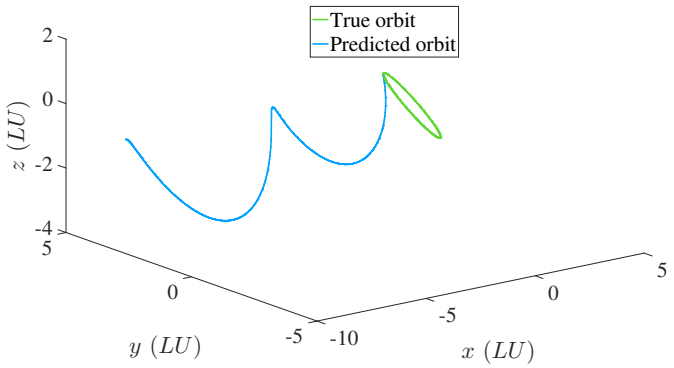


(b) Norm of velocity error vs. time



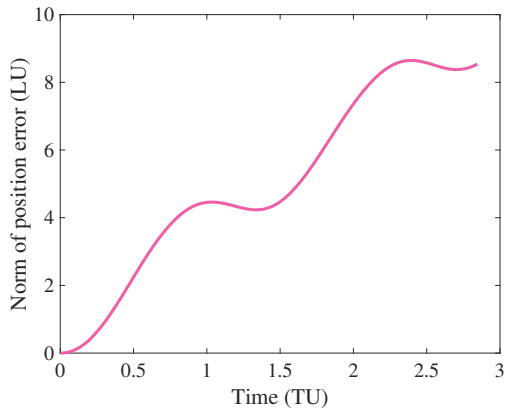
(c) Constant of the Motion Violation

Figure 4.10: Error induced by NN over training data involving time prediction with soft constraints

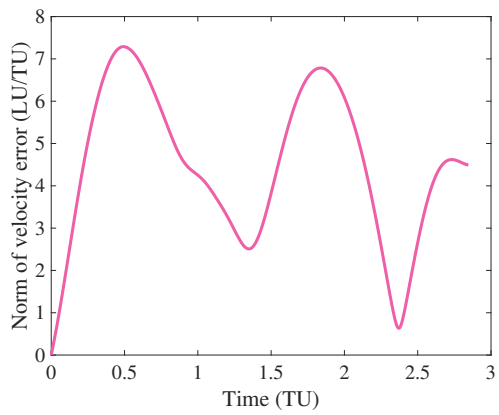


(a) True and NN Approximated Orbits

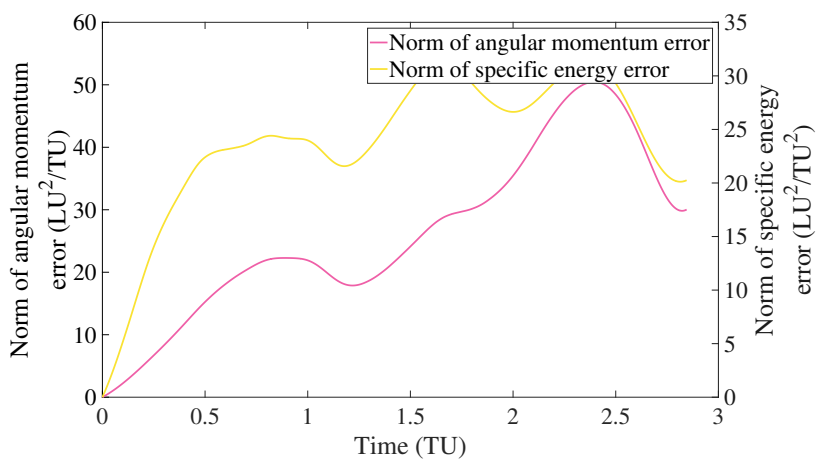
Figure 4.11: Performance of NN over unknown data involving time prediction with soft constraints



(a) Norm of position error vs. time



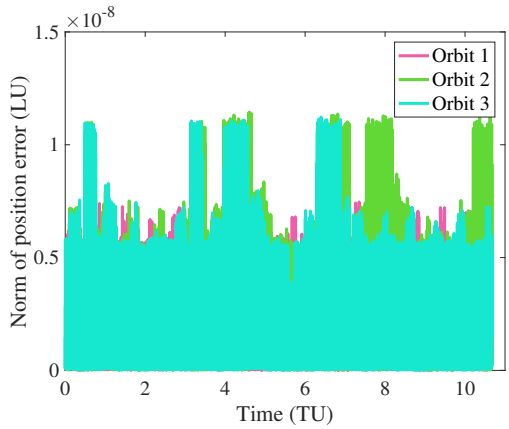
(b) Norm of velocity error vs. time



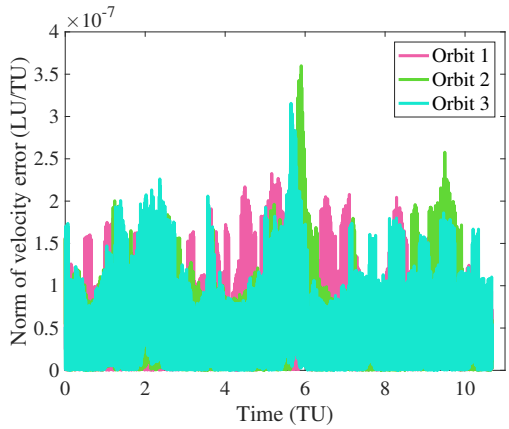
(c) Constant of the Motion Violation

Figure 4.12: Error induced by NN over unknown data involving time prediction with soft constraints

In addition, Figures 4.13 and 4.14 shows the enhanced training by including constraints violation in the Loss function. Similarly as before, the precision accuracy is one order of magnitude better. In position, the absolute error between the neural network prediction and the true value does not exceed 1.5×10^{-8} LU with a mean below 4×10^{-9} LU. The norm of velocity approximation error does not exceed 4×10^{-7} LU/TU with a mean around 5×10^{-8} LU/TU. Because the integration error is close to 10^{-10} LU in position and 10^{-9} LU/TU in velocity, these plots confirm for the first time the ability of the NN model to learn the orbit dynamics.

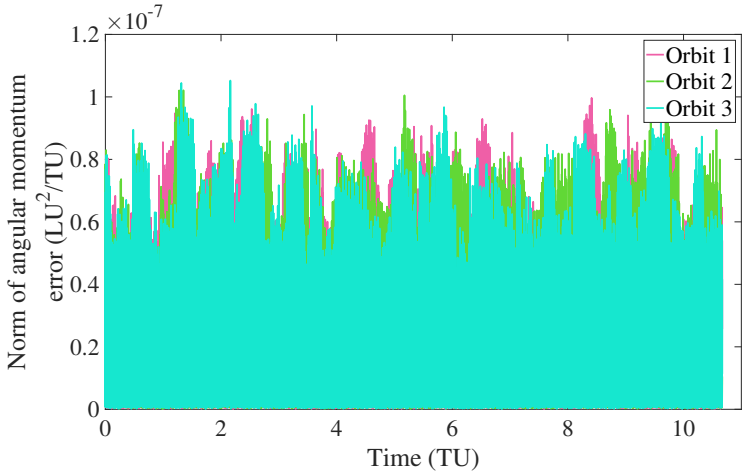


(a) Norm of position error vs. time

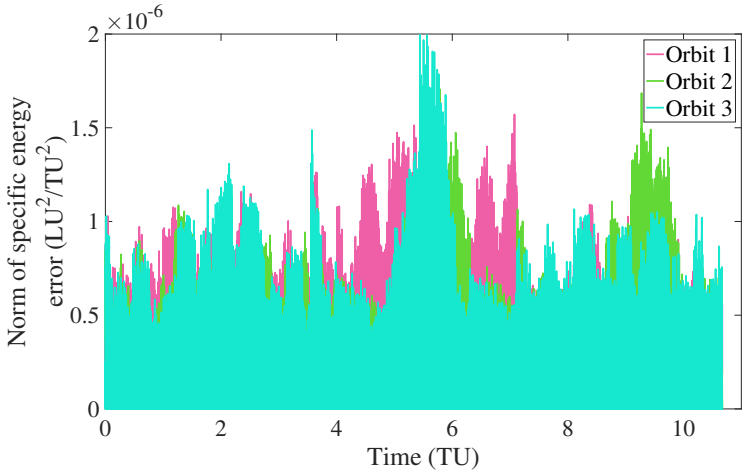


(b) Norm of velocity error vs. time

Figure 4.13: Performance of NN over unknown data involving time prediction for a large dataset with soft constraints



(a) Constant of the Motion Violation



(b) Constant of the Motion Violation

Figure 4.14: Error induced by NN over unknown data involving time prediction for a large dataset with soft constraints

4.4 Influence of the Architecture

This section considers three different NN architectures: Feed-Forward (FF), Residual Neural Network (ResNet) and Deep Residual Neural Network (DeepResNet). The capabilities of three ResNet and two DeepResNet architectures will be compared with the FF [30 30 30] approximation capabilities from the previous sections. Table 4.2 presents the six different NN architectures used to investigate the approximation

and interpolation capabilities of NN-based methods.

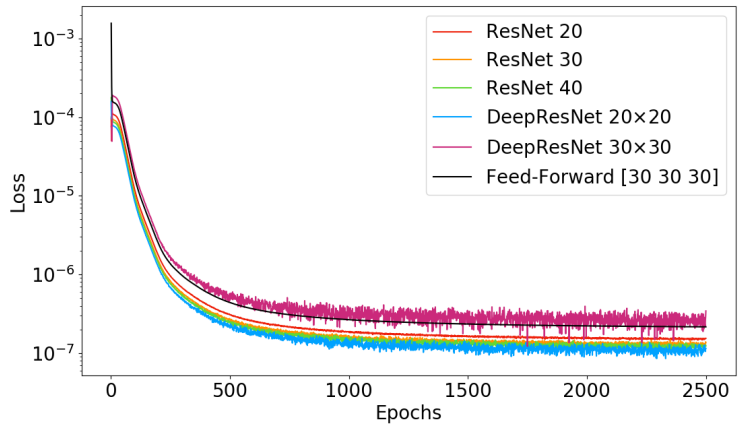
Table 4.2: NN architectures tested

Feed-Forward	ResNet	DeepResNet
3 layers with 30 neurons per layers	20 Residual Blocks 30 Residual Blocks 40 Residual Blocks	20 layers with 20 neurons 30 layers with 30 neurons

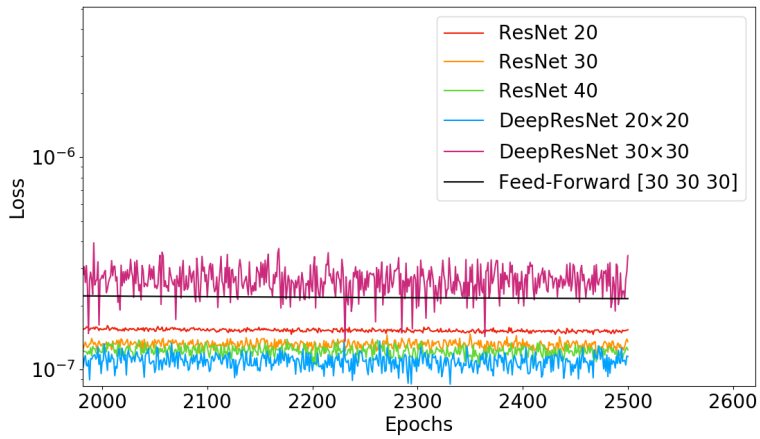
The ResNet and DeepResNet structures used in this section are those presented in Chapter 2. Both networks are implemented with a single wrapping layer in order to take into account the change of dimension of the output (acceleration) with respect to the input (position). However, the wrapping layer preserves the dimensionality as both input and output are of dimension three.

The evolution of the Loss function for these six networks can be seen Figure 4.15(a) with a zoom between epochs 2000 and 2500 on Figure 4.15(b). From Figure 4.15(a), it is clear that the Loss function is monotically decreasing only for the FF neural network to reach a steady state value of approximatively 2×10^{-7}. The Loss function for ResNets 20, 30, 40 is globally decreasing but oscillations can be seen early in the training phase. After 2000 epochs, the Loss is stabilized just above 10^{-7}; Loss for ResNet 20 being slightly higher than Loss for ResNet 30 again slightly higher than Loss for ResNet 40. DeepResNet 20×20 achieves the smaller Loss among all the networks, reaching 10^{-7}, but still with some oscillations. On the other hand, DeepResNet 30×30 presents greater amplitude for oscillations combined with the highest Loss among all the networks above 2×10^{-7}. Hence, we will focus on three structures that produce the best performance on the training data set: Feed-Forward [30 30 30], ResNet 40 and DeepResNet 20×20.

Figures 4.16(a), 4.16(c) and 4.16(e) show the norm of the approximation error for position for the three test orbits not included in the training (each orbit being separated by a vertical line) for the three NN structures. Considering the FF NN, the absolute error between the neural network prediction and the true value does not surpass 1.4×10^{-8} LU with a mean at 4.24×10^{-9} LU. For ResNet 40 and DeepResNet 20×20, the absolute error in position is below 10^{-8} LU with a mean at 1.99×10^{-9} LU and 1.38×10^{-9} LU respectively. Figures 4.16(b), 4.16(d) and 4.16(f) show the norm of the approximation error for velocity for the three test orbits for the



(a) Loss Function Evolution



(b) Loss Function Evolution (Zoomed Version between epochs 1800 and 2000)

Figure 4.15: Evolution of the Loss function for different network architectures.

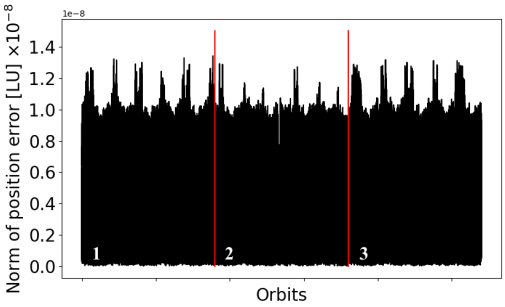
three NN structures. The FF NN achieves an overall absolute error below 4×10^{-7} LU/TU with a mean at 9.24×10^{-8} LU/TU. For ResNet 40, the absolute error in velocity stays below 2.5×10^{-7} LU/TU with a mean at 4.29×10^{-8} LU/TU. Finally, DeepResNet 20×20 does not surpass 1.75×10^{-7} LU/TU with a mean at 2.61×10^{-8} LU/TU. These results show good approximation capabilities by the three different structures with an advantage to DeepResNet 20×20. Errors for position and velocity are very close to integration tolerances, typically around 10^{-10}. To further

investigate the dynamical approximation capabilities of the NNs and see whether the penalty term in the Loss function is efficient to get conservation of angular momentum and total energy. Figures 4.17(a), 4.17(c) and 4.17(e) first show the norm of the approximation error for angular momentum. FF NN is at a maximum of 1.2×10^{-7} LU^2/TU, ResNet 40 at 6×10^{-8} LU^2/TU and DeepResnNet 20×20 at 4×10^{-8} LU^2/TU. Average values for the three networks are 2.45×10^{-8} LU^2/TU, 1.14×10^{-8} LU^2/TU and 6.02×10^{-9} LU^2/TU respectively. Again, DeepResnNet 20×20 achives minimum approximation error. Finally, Figures 4.17(b), 4.17(d) and 4.17(f) show the norm of the approximation error for total energy. FF NN, ResNet 40 and DeepResNet 20×20 present a maximum at 2×10^{-6} LU^2/TU^2, 1.2×10^{-6} LU^2/TU^2 and 7×10^{-7} LU^2/TU^2 with average values at 2.40×10^{-7} LU^2/TU^2, 1.57×10^{-7} LU^2/TU^2 and 9.71×10^{-8} LU^2/TU^2 respectively. Overall, DeepResNet 20×20 achieves a better accuracy for the three orbits approximation and is able to capture the constants of the motion. Table 4.3 summurizes the average error for the three different networks over the three test orbits.

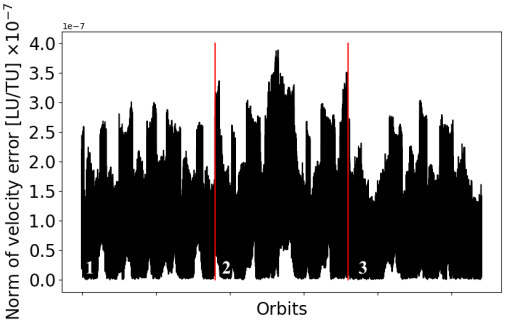
Table 4.3: Average errors over 3 orbits for the different NN structures considered

Average error over 3 test orbits	**FF [30 30 30]**	**ResNet 40**	**DeepResNet 20×20**
Norm of position error [LU]	4.24×10^{-9}	1.99×10^{-9}	1.38×10^{-9}
Norm of velocity error [LU/TU]	9.24×10^{-8}	4.29×10^{-8}	2.61×10^{-8}
Norm of angular momentum error [LU^2/TU]	2.45×10^{-8}	1.14×10^{-8}	6.02×10^{-9}
Norm of energy error [LU^2/TU^2]	2.40×10^{-7}	1.57×10^{-7}	9.71×10^{-8}
Number of parameters	2070	480	7720

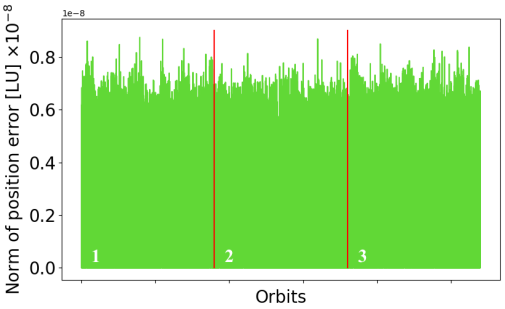
Comparing NN capabilities, it is essential to take into account the complexity of the network represented by the number of free parameters. From Table 4.3, one can infer that ResNet 40 provides the better tradeoff between accuracy and complexity of the network by providing the same order of accuracy as DeepResNet but with an order of magnitude less parameters.



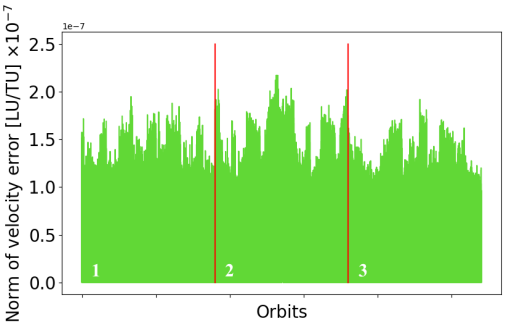
(a) Position error FF NN



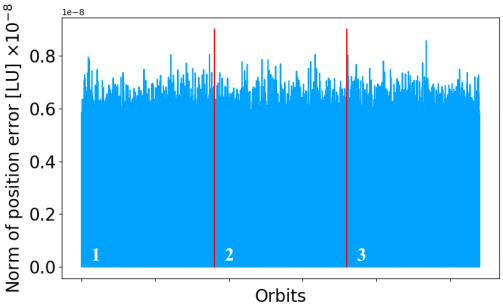
(b) Velocity error FF NN



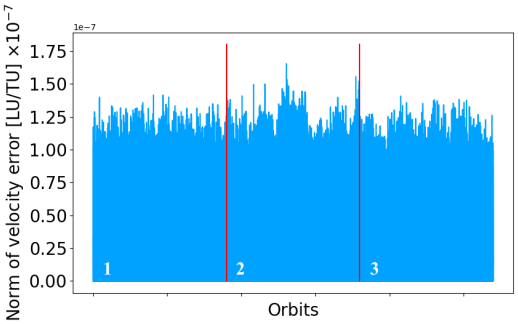
(c) Position error ResNet 40



(d) Velocity error ResNet 40

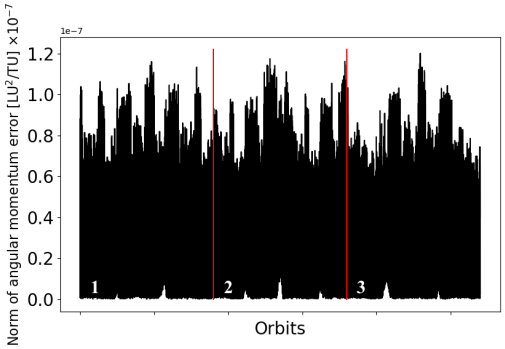


(e) Position error DeepResNet 20×20

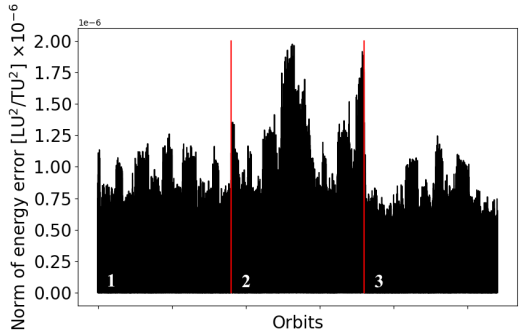


(f) Velocity error DeepResNet 20×20

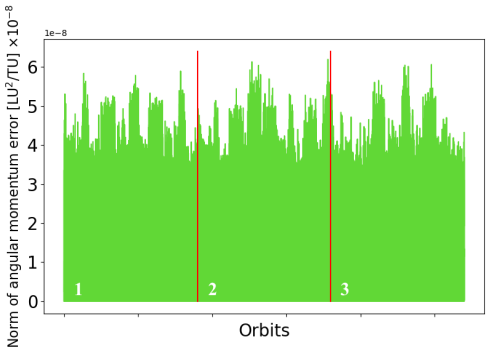
Figure 4.16: Error induced by NN over unknown data involving time prediction for a large dataset with soft constraints for different structures



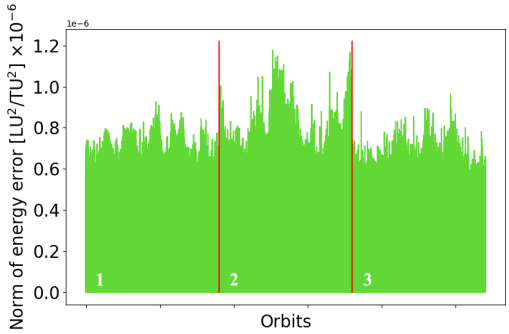
(a) Angular momentum error FF NN



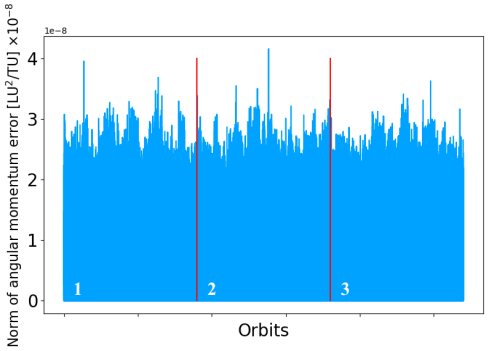
(b) Energy error FF NN



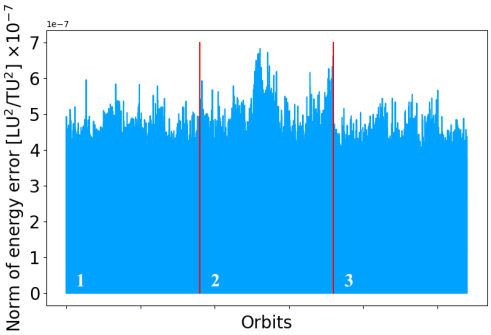
(c) Angular momentum error ResNet 40



(d) Energy error ResNet 40



(e) Angular momentum error DeepResNet 20×20



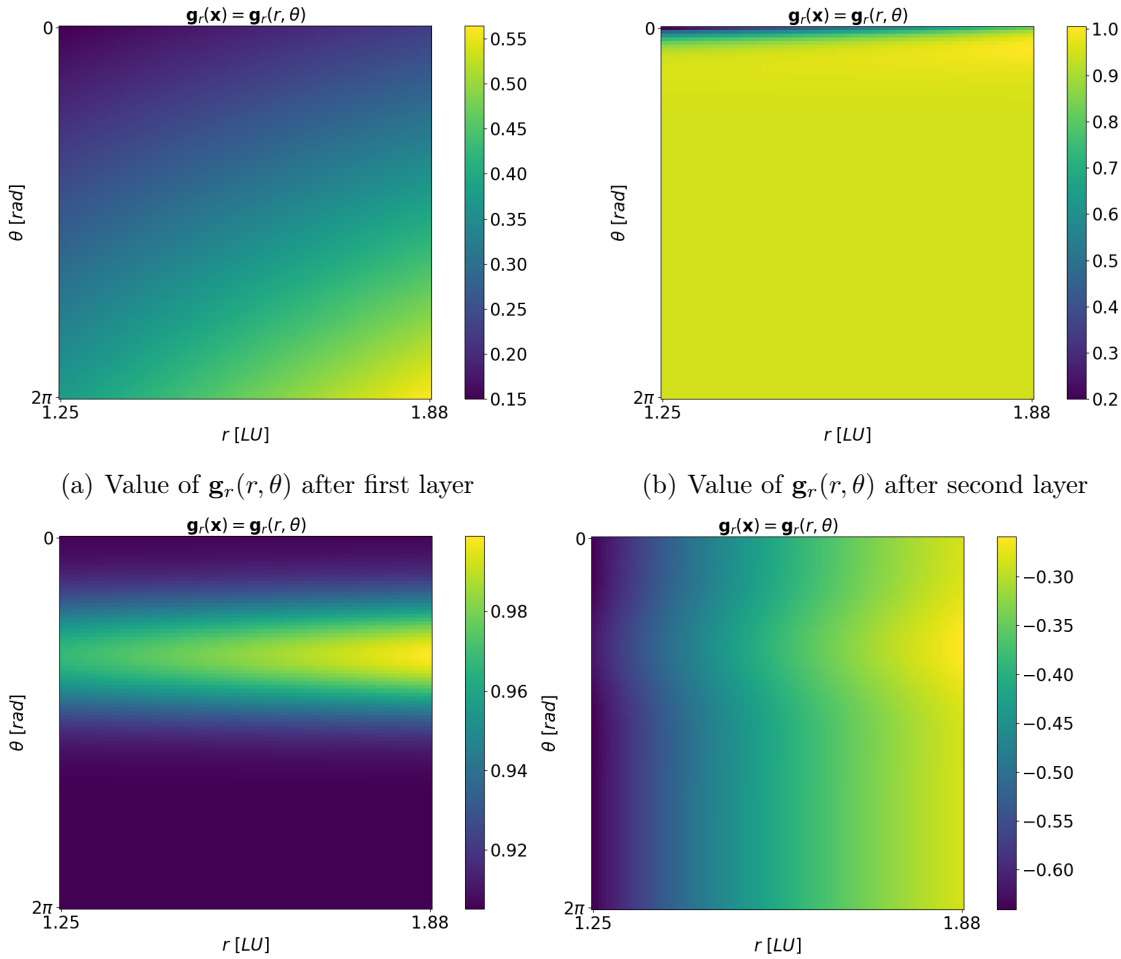
(f) Energy error DeepResNet 20×20

Figure 4.17: Error induced by NN over unknown data involving time prediction for a large dataset with soft constraints for different structures

4.5 Learning in Polar Coordinates

To investigate the precise behavior of the NN and assess its capabilities to approximate the dynamics of the central force, we perform an analysis on the value of the weights and bias of the network. More practically, we feed the network with all possible combinations of r and θ and check the value of $\mathbf{g}_r(r, \theta)$ and $\mathbf{g}_\theta(r, \theta)$ layer per layer. Four different snapshots are taken: after the first layer, after the second layer, after the third layer and at the output.

Figures 4.18(a) to 4.18(d) present the four snapshots for the value of $\mathbf{g}_r(r, \theta)$. After the first layer, Figure 4.18(a), $\mathbf{g}_r(r, \theta)$ is clearly dependent on both r and θ. The first layer of weights and bias does not set a value of $\mathbf{g}_r$ independent of θ. After the second layer, Figure 4.18(b), almost all the domain is set to 1 which implies high values for weights and bias before the activation function. The only variation that can be seen is for low r at θ near 0. After the third layer, Figure 4.18(c), the value of $\mathbf{g}_r(r, \theta)$ is almost constant with a small variation around $\theta = 2$rad. At this point, $\mathbf{g}_r$ is still dependent on θ. From Figure 4.18(d), it is evident that $\mathbf{g}_r(r, \theta)$ is independent of θ. Hence, the NN is able to learn the inherent symmetry of the dynamical system.



(a) Value of $\mathbf{g}_r(r, \theta)$ after first layer

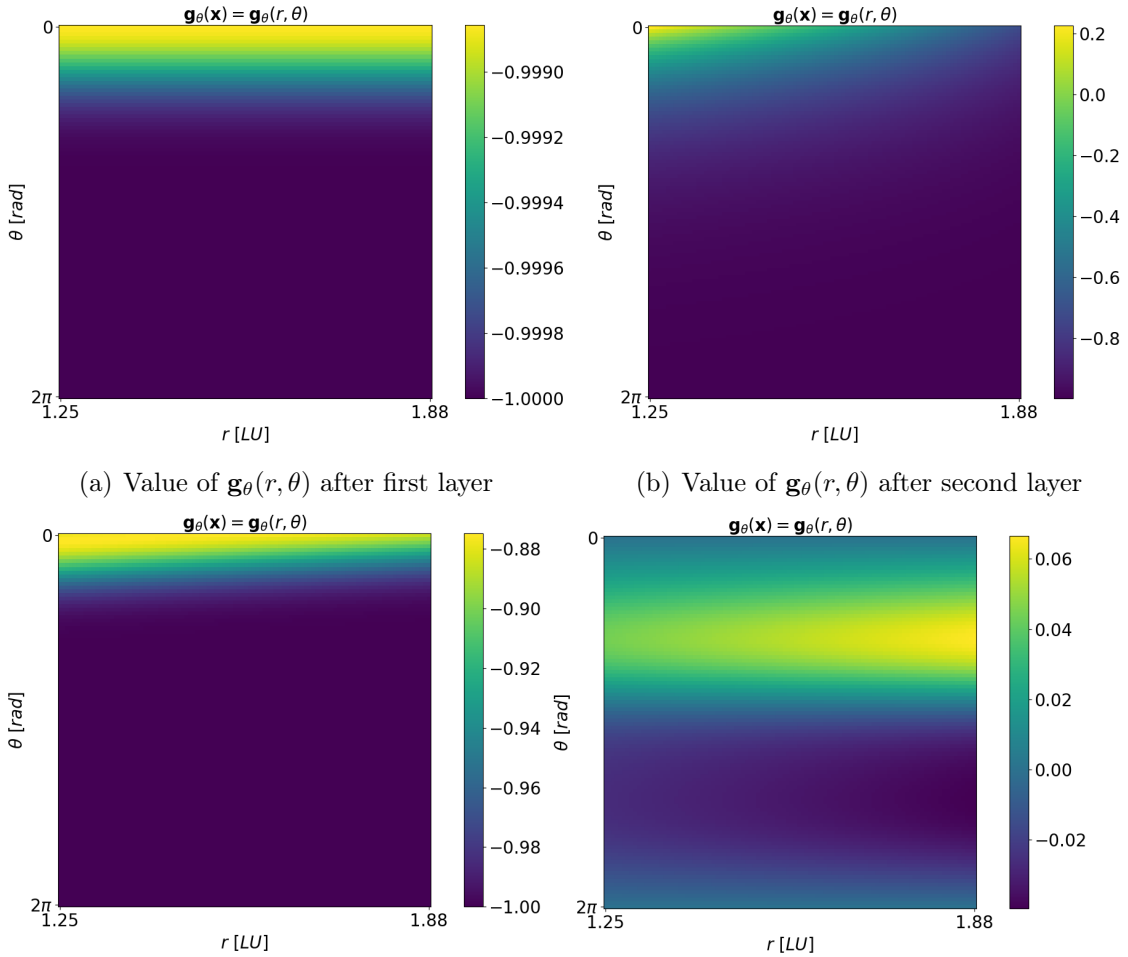
(b) Value of $\mathbf{g}_r(r, \theta)$ after second layer

(c) Value of $\mathbf{g}_r(r, \theta)$ after third layer

(d) Value of $\mathbf{g}_r(r, \theta)$ at the output

Figure 4.18: Investigation on $\mathbf{g}_r$

Figures 4.19(a) to 4.19(d) present the four snapshots for the value of $\mathbf{g}_\theta(r, \theta)$. After the first layer, Figure 4.19(a), $\mathbf{g}_\theta(r, \theta)$ is basically sent to -1. It can be seen also that the value of $\mathbf{g}_\theta$ is independant of r. After the second layer, Figure 4.19(b), low θ yields a value of $\mathbf{g}_\theta$ slightly less than -1, almost independant of r again. After the third layer, Figure 4.19(c), the value of $\mathbf{g}_\theta(r, \theta)$ has considerably changed and varies now between 0.2 and -1. Also, $\mathbf{g}_\theta$ is now dependant of r. From Figure 4.19(d), it is evident that the output of the NN approximates the true zero value of $\mathbf{g}_\theta$ with 2 decimal space accuracy. Further investigations are required to assess the effect of NN structure on learning in two dimensionnal r, θ space.



(a) Value of $\mathbf{g}_\theta(r, \theta)$ after first layer

(b) Value of $\mathbf{g}_\theta(r, \theta)$ after second layer

(c) Value of $\mathbf{g}_\theta(r, \theta)$ after third layer

(d) Value of $\mathbf{g}_\theta(r, \theta)$ at the output

Figure 4.19: Investigation on $\mathbf{g}_\theta$

Table 4.4 summurizes the average error for the three different networks over the three test orbits in polar coordinates. It is to be noted that the first column presents the results in Cartesian coordinates when there is no penaly term included in the Loss function. It is clear to see that this first attempt in polar coordinates achieves lower accuracy than a training in Cartesian coordinates with 3 orders of magnitude in difference in accuracy. Further comparisons with other structures can be made by confronting values from Table 4.4 with those from Table 4.3.

Table 4.4: Average errors over 3 orbits for the different NN structures considered in Cartesian and polar coordinates with no penalty terms in the Loss function

Average error over	**Cartesian**	**Polar**		
3 test orbits	**FF [30 30 30]**	**FF [30 30 30]**	**ResNet 40**	**DeepResNet 20×20**
Norm of position error [LU]	7.94×10^{-8}	5.02×10^{-4}	1.22×10^{-4}	9.98×10^{-5}
Norm of velocity error [LU/TU]	2.02×10^{-7}	8.50×10^{-4}	2.41×10^{-4}	1.01×10^{-4}
Norm of angular momentum error [LU^2/TU]	3.41×10^{-7}	2.53×10^{-3}	6.55×10^{-4}	2.11×10^{-4}
Norm of energy error [LU^2/TU^2]	9.33×10^{-7}	2.34×10^{-3}	8.99×10^{-4}	5.49×10^{-4}

We recall that training a NN is equivalent to solve a large optimization problem where the parameters are the weights and biases of the structure. Therefore, a random initialization at the beginning of the training phase could potentially lead to a local minimum. In order to verify whether previous case was a local minimum, the initial values of the weights from the last layer corresponding to the output $\mathbf{g}_\theta(r, \theta)$ are set to 0 (same for biases). Table 4.5 summurizes the average error for the three different networks over the three test orbits when initializing the weigths for $\mathbf{g}_\theta(r, \theta)$ to 0. In this specific scenario, the first column includes the training in Cartesian coordinates with penalty terms in the Loss function. The motivation is that setting initial weigths to 0 for $\mathbf{g}_\theta(r, \theta)$ is similar to enforce conservation of angular momentum (see second line of (3.24)). On average, ResNet 40 and DeepResNet 20×20 achieve better accuracy and are able to predict an accurate orbit up to 3.03×10^{-9} LU in position and 8.76×10^{-8} LU/TU in velocity as well as reproducing the constants of the motion.

Table 4.5: Average errors over 3 orbits for the different NN structures considered in Cartesian and polar coordinates including penalty terms in the Loss function for training in Cartesian coordinates and initializing weigths for $\mathbf{g}_\theta(r, \theta)$ at 0

Average error over	**Cartesian**	**Polar**		
3 test orbits	**FF [30 30 30]**	**FF [30 30 30]**	**ResNet 40**	**DeepResNet 20×20**
Norm of position error [LU]	4.24×10^{-9}	4.54×10^{-9}	3.12×10^{-9}	3.03×10^{-9}
Norm of velocity error [LU/TU]	9.24×10^{-8}	9.56×10^{-8}	9.05×10^{-8}	8.76×10^{-8}
Norm of angular momentum error [LU^2/TU]	2.45×10^{-8}	2.76×10^{-8}	1.09×10^{-8}	1.60×10^{-8}
Norm of energy error [LU^2/TU^2]	2.40×10^{-7}	3.71×10^{-7}	2.31×10^{-7}	2.13×10^{-7}

Chapter 5 Conclusion

5.1 Conclusion of Research Results

In this study, we have investigated the learning capabilities of neural networks based on the well known Keplerian two-body problem. The goal was to examine whether the specific structure of neural networks could learn the inherent dynamics of the two-body problem and examine whether a neural network learned model could reproduce well-known characteristics of Keplerian dynamics such as conservation of energy and angular momentum. We considered several test cases to assess the learning capability of the converged NN. Although NN approximation errors are very small for a training data set involving the prediction of a specific orbit, the approximation accuracy deteriorates rapidly for the test data set. The NN approximation completely breaks down when it is tasked to predict an orbit not included in the training data set. Furthermore, the NN-learned model performs very poorly in reproducing constants of the motion. However, the accuracy of the NN-learned model is increased by an order of magnitude when one includes the penalty term in the Loss function formulation corresponding to the violation of constants of the motion during the NN training. Furthermore, the performance of the NN-learned model increases considerably when the training data set is made richer by training the neural network over a set of orbits rather than a specific orbit. In addition to the study on the training set and the Loss function, we investigated the approximation and prediction capabilities of three types of neural networks: Feed-Forward, Residual and Deep Residual. First using Cartesian coordinates, it has been shown that the three structures are able to provide accurate results for orbit

prediction considering a large data set while incorporating violation of constants of the motion in the Loss function. While the DeepResNet structure considered provides the most accurate results, the ResNet architecture shows very similar performance with many fewer parameters. Both orbit prediction and constants of the motion are approximated within numerical integration tolerances. Same work using polar coordinates has been performed to assess the learning capabilities of a NN model in a description where the central force is explicit. Simple FF NN structure was able to find the symmetry of the two-body problem up to 2 decimal places of accuracy. Initializing the NN with weights corresponding to conservation of the angular momentum allows the training in polar coordinates to be the most accurate. Once again, ResNet and DeepResNet structures show the best results overall.

5.2 Future Recommendations

Although it seems that the NN-learned model can be trained to approximate Keplerian dynamics to good accuracy, the complexity of the learned model is still an issue to be investigated. The resulting NN model is a profligate model for Keplerian dynamics as compared to Newton's law of gravitation. Hence, future research can first be concentrated on investigating different architectures for the NN-based model such as a purely recursive model to take into account the time dependency of the input data. It is also possible to incorporate the recurrent model during the training phase directly by allowing the prediction capabilities to be several time steps instead of only one. There are also many other parameters that can be tuned to adjust the training of the NN model such as other activation functions or even the selection of other optimizer algorithms depending on the neural network architecture.

More important, the ultimate goal would be to develop a parsimonious NN model of the Keplerian dynamics. Different NN architectures mean a different number of parameters and different accuracy. Representing the dynamics of the two-body problem with a converged NN signifies that no less than the total number of parameters of the network is required to identify the system. And there always is a trade-off between accuracy and computational burden. A more complex structure might have better accuracy but larger number of parameters as well as more

demanding computational needs.

Finally, future work should concentrate on how one can use a regularization method to incorporate apriori information about the dynamical system. Not only would it allow the gain of prior information over the network parameters but it would also bring better knowledge on how to handle these parameters as well as the initial guess. Automatically incorporating radial field information as a regularization term in the Loss function would allow one to generate prior insight before the training phase. For example, one could choose to minimize the one-norm of the corresponding weights of the g_θ term to enforce central force field approximation. Depending upon the coordinate choice, one would have different regularization terms in the Loss function. Eventually, one could determine how to incorporate hard constraints in the optimization process (and deal with a modified version of the optimizers) to enforce known information during the training and obtain better approximation capabilities.

Bibliography

[1] K. S. Narendra, K. Parthasarathy, "Neural Networks and Dynamical Systems", International Journal of Approximate Reasoning, 1992.

[2] Y.-J. Wang, C.-T. Lin, "Runge-Kutta Neural Network for Identification of Dynamical Systems in High Accuracy", IEEE TRANSACTIONS ON NEURAL NETWORKS, 9(2), 1998.

[3] S. Haykin, "Neural Networks: A Comprehensive Foundation", Prentice Hall PTR, 1994.

[4] S. Haykin, "Neural Networks and Learning Machines", volume 3, Pearson Upper Saddle River, NJ, USA, 2009.

[5] C. M. Bishop, "Pattern Analysis and Machine Intelligence", Springer, New York, 2004.

[6] K. He, X. Zhang, S. Ren, J. Sun, "Deep Residual Learning for Image Recognition", arXiv:1512.03385v1, 2015.

[7] X. Bai, H. Peng, "Improving Orbit Prediction Accuracy Through Supervised Machine Learning", 2018.

[8] Z. Li, Z. Shi, "Deep Residual Learning and PDEs on Manifold", arXiv:1708.05115v3, 2018.

[9] W. S. McCulloch, W. Pittsa, "Logical Calculus of Ideas Immanent in Nervous Activity", Bulletin of Mathematical Biology, Vol. 52, no. 1/2, pp. 99-115, 1990.

[10] A. N. Kolmogorov, "On the Representation of Continuous Function of Several Variables by Superposition of Continuous Functions of One Variable and Addition", Doklady Akademiia Nauk SSSR, vol. 114, no. 5, pp. 953-956, 1957.

[11] J. Duchi, E. Hazan, Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", Journal of Machine Learning Research, 2121-2159, 2011.

[12] M. D. Zeiler, " Adadelta: an Adaptive Learning Rate Method", arXiv:1212.5701, 2012.

[13] D. P. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization", arXiv:1412.6980, 2014.

[14] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, et al, "Ad Click Prediction: A View from the Trenches", In Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 1222-1230, ACM, 2013.

[15] G. Hinton, N. Srivastava, and K. Swersky. rmsprop, `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[16] T. Dozat, "Incorporating Nesterov Momentum Into Adam", 2016.

[17] S. Ruder, "An Overview of Gradient Descent Optimization Algorithms", arXiv:1609.04747v2, 15 Jun 2017.

[18] A. Veit, M. Wilber, S. Belongie, "Residual Networks Behave Like Ensembles of Relatively Shallow Networks", arXiv:1605.06431v2, 2016.

[19] R. K. Srivastava, K. Greff, J. Schmidhuber, "Highway Networks", arXiv:1505.00387v2, 2015.

[20] S. Zagoruyko, N. Komodakis, "Wide Residual Networks", arXiv:1605.07146v4, 2017.

[21] S. Xie, R. Girshick, P. Dollar, Z.Tu, K. He, "Aggregated Residual Transformations for Deep Neural Networks", arXiv:1611.05431v2, 2017.

[22] M. Hauser, S. Gunn, S. Saab Jr, A. Ray, "State Space Representations of Deep Neural Networks", arXiv:1806.03751v3, 21 Feb 2019.

[23] Serio, Giorgia Foderà and Manara, Alessandro and Sicoli, Piero and Bottke, William Frederick, "Giuseppe Piazzi and the discovery of Ceres", University of Arizona Press, 2002.

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. "Tensorflow: a system for large-scale machine learning", in OSDI, volume 16, pages 265-283, 2016.

[25] J. L. Junkins and P. Singla, "How Nonlinear is It? A Tutorial on Nonlinearity of Orbit and Attitude Dynamics", The Journal of the Astronautical Sciences, vol. 52, no. 1-2, 2004.