POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Physics-based polynomial neural networks for learning of dynamical systems

Tesi di Laurea Magistrale in
Mathematical Engineering - Ingegneria Matematica

Author: **Daniel Sakir**

Student ID: 10488218
Advisor: Edie Miglio
Academic Year: 2020-21

Abstract

The target of this work is to analyze the connection between Taylor maps (TM) and polynomial neural networks (PNN) to numerically solve ordinary differential equations (ODEs).

The Taylor map is a polynomial approximation of the general solution of a nonlinear system of ODEs: by using the TM, if the explicit form of the equations is known, it is possible to directly evaluate the weights of the PNN.

For this reason the PNN can be used for simulation of system dynamics with different initial conditions without the training procedure. On the other hand, if the equations are unknown, the weights can be fitted in a data-driven way: from only one solution of the system it is possible to recover his dynamics (one-shot learning).

Some results will be presented in order to show the effectiveness of the proposed method, for examples the system of Van der Pol oscillator and the one that describe the dynamic of a charged particle in an electromagnetic field are considered.

In the final part of the work a comparison between the PNN and other classical neural networks (i.e. LSTM and MLP) is exposed: the PNN will be the best to recover physical information by starting from only one observation.

Keywords: dynamical system, polynomial neural network, one-shot learning

Abstract in lingua italiana

L'obiettivo del seguente lavoro è quello di mostrare la possibilità di risolvere sistemi di equazioni differenziali ordinarie mediante l'utilizzo di uno speciale neural network chiamato "polynomial neural network" (PNN).

Per la costruzione di tale modello verrà utilizzata la Mappa di Taylor(TM): essa è un'approssimazione polinomiale della soluzione generale di un sistema dinamico.

La Mappa di Taylor sarà fondamentale per l'inizializzazione delle matrici dei pesi del PNN, in particolare: se il sistema dinamico e le sue equazioni sono note, la TM permette un calcolo diretto ed il neural network non necessita di alcuna fase di allenamento; al contrario, se non sono disponibili informazioni a priori riguardanti le equazioni, la matrice dei pesi si costruisce direttamente dai dati, avendo a disposizione un'unica soluzione (in questo caso si parla di apprendimento one-shot).

Attraverso degli esempi, come quello del pendolo semplice o del modello preda-predatore di Lotka-Volterra, verranno mostrati i vantaggi di questo speciale neural network: esso raggiunge un'alta capacità di generalizzazione con cui è possibile fare delle previsioni accurate riguardo al funzionamento del sistema dinamico preso in esame.

Infine ci sarà un confronto tra il PNN e altri neural network classici (cioè LSTM e MLP): verrà mostrato come, partedo da un'unica soluzione disponibile con cui allenare il network, il PNN è quello che fornisce i risultati migliori.

Parole chiave: Sistema dinamico, apprendimento da una soluzione, polynomial neural network

Contents

Abstract i

Abstract in lingua italiana ii

Contents iii

Introduction 1

1 Taylor Maps for solving Ordinary Differential Equations 4
- 1.1 Taylor Map . . . 4
  - 1.1.1 Evaluation of the weights . . . 6
- 1.2 Test cases . . . 8
  - 1.2.1 Free fall of a body . . . 9
  - 1.2.2 Pendulum . . . 12
  - 1.2.3 Van der Pol . . . 16
  - 1.2.4 Charged particle . . . 20
- 1.3 Conclusion . . . 22

2 Taylor-Map Polynomial Neural Network 24
- 2.1 Polynomial Neural Network . . . 24
  - 2.1.1 Implementation . . . 25
- 2.2 Application of the PNN . . . 27
- 2.3 Identification problem . . . 28
  - 2.3.1 Long Short Term Memory . . . 29
  - 2.3.2 Results . . . 30
- 2.4 Fine-tuning problem . . . 33

3 Conclusions and future developments 36

Bibliography **37**

A Appendix A **39**

List of Figures **53**

List of Tables **54**

Acknowledgements **55**

Introduction

The traditional approach for representing and modeling a physical problem is the use of the dynamical systems: they are systems of differential equations (involving time derivatives) that encode a rule which describe the time dependence of a point in a geometrical space; the equations are based on conservation laws, e.g. mass, momentum, energy.

Such systems arise in different fields such as automated control, robotics, mechanical and biological systems, chemical reactions, drug development, molecular dynamics, and so on.

Examples are the mathematical models that describe the swinging of a clock pendulum or the number of fish each springtime in a lake.

At any given time, a dynamical system has a state given by a tuple of real numbers (a vector) that can be represented by a point in an appropriate state space (a geometrical manifold). The evolution rule of the dynamical system is a function that describes what future states follow from the current state.

The resolution of the differential equations which compose a dynamical system is an important field of study which involves the use of numerical methods, but they could deal with some troubles: to meet the scale-accuracy trade-off some approaches that are computationally expensive are used and leads to time consuming algorithms.

Another interesting field of study is the one of system identification: it consists in the application of statistical methods to build mathematical models of dynamical systems from measured data.

The main idea developed in this work is to make use of some Machine Learning (ML) techniques to solve two problems: the first is the resolution of the mathematical equations of the dynamical system avoiding complex numerical solvers; the second is the extraction the physical information of a real problem for which the equations are unknown, thinking to it as a sort of black box model for which only some output data are available.

The use of ML models is particularly promising in scientific problems which involve processes that are not completely understood or where it is computationally infeasible to run numerical methods at desired resolutions in space and time. However, the application of classical ML models has often met with limited success in scientific domains due to their large data requirements, inability to produce physically consistent results, and their

lack of generalizability to out-of-sample scenarios.

In this work it is explained how to build a Taylor map- polynomial neural network (TM-PNN) to solve the problem of one-shot learning of dynamical systems (one-shot means to learn from only one solution), a result that is permitted thanks to the possibility of incorporate prior physical knowledge into the neural network to improve data efficiency and the capacity of generalization of predictive models.

If the equations of a dynamical system satisfy the hypothesis of beeing approximated using polynomial differential equations, it is possible to build a polynomial neural network that is based on the Taylor map and that can approximate the general solution of the system of equations.

By using the Taylor map the weights of the PNN can be initialized directly and the solution of the system can be estimate without the training phase of the neural network. In addition any numerical solver in both simulation and data-driven system learning is used.

If the equations of the system are not available, the PNN can works without the previous stage of inizialization, it can learn the physical information of the problem and generalize in a better way than others classical ML methods (e.g. LSTM or MLP).

On the contrary, if the equations are given but there are some uncertainties related to the coefficients, the PNN can be fine-tuned by learning from the available data.

Fine-tuning consists in helping to improve the accuracy of a neural network model by integrating data from an existing neural network and using it as an initialization point to make the training process time and resource-efficient.

The organization of the thesis is as follows: in Chapter 1 there is a theoretical introduction of the Taylor map, followed by the description on how it can be used to find both the weights of the PNN and the solution of dynamical systems; in Chapter 2 the building procedure of the PNN is explained, with some details on how the layers are implemented, then we will see his application in two cases that regard the recovering problem of dynamics for a model whose equations are unknown (i.e. identification problem) and the fine-tuning issue of the PNN.

In particular the Chapter 1 begins with a theoretical explication of the Taylor map, followed by a detailed description on how to build an iterative method with which is possible to evaluate the solutions of dynamical systems: the crucial part is the computation of the coefficients of the polynomial approximation, which are called “weights”; in order to show his effectiveness some test cases, like the system of simple pendulum or the one of body in free fall, are considered.

The Chapter 2 is all about the implementation of the PNN and his possible applications: first we will see in deeply how to implement the layers and the network and how they

work, then his effectiveness on two cases is exposed.

In particular the solutions of two problems is searched: the first regards the identification capability of the PNN, the system of Lotka-Volterra will be considered as test case; the second is a fine-tuning problem, i.e. starting from a dynamical system whose equations are known, but where there are some doubts on the coefficients that make them inaccurate, a method to improve the accuracy of the PNN is shown.

1 | Taylor Maps for solving Ordinary Differential Equations

In this Chapter a theoretical description to the Taylor map will be presented: it is a polynomial approximation of the general solution of a nonlinear system of ODEs, which can be also interpreted as a polynomial regression with respect to the component of $\mathbf{X}$ (the state of the dynamical system represented by the equations).

Through the TM, if the equations of the system are available, it is possible to evaluate the coefficients of this particular regression, called "weights": within the Chapter some comparison between the weights found by this map and the ones obtained in the papers [4], [6], [7] are done. The necessity of doing that is due to the fact that the code used for this work is independent from the one used in other works.

Once the weights values are obtained, an iterative method is built and used to evaluate the solution of some systems(e.g. simple pendulum or Van der Pol), introduced as test in order to show the effectiveness of the proposed approach.

1.1. Taylor Map

The traditional way to represent a physical system is by using dynamical system: it consists in any fixed "rule" that describes the time dependence of the position of a point in its ambient space.

This "rule" is encoded by a system of differential equations (in this work only ODE are considered, but they could be also stochastic or partial derivative).

Let's consider a generic nonlinear ODEs:

$$\frac{d}{dt}\mathbf{X} = \mathbf{F}(t, \mathbf{X}), \tag{1.1}$$

where t is the independent variable and $\mathbf{X} \in R^n$ is the state vector. There is an important assumption to do: suppose that function $\mathbf{F}$ can be expanded in Taylor series with respect to the components of $\mathbf{X}$. (Note that the independent variable t can arise in the equation as an arbitrary nonlinear function).

By this assumption it is possible to represent the function $\mathbf{F}$ as a time series in its convergence region:

$$\mathbf{F} = \sum_{k=0}^{\infty} P_k \mathbf{X}^{[k]}, \tag{1.2}$$

where $\mathbf{X}^{[k]}$ means the k-th Kronecker power of vector $\mathbf{X}$ with the same terms reduction.

To better understand this concept consider the definition of Kronecker power:

Definition 1. *If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is a $p \times q$ matrix, then the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is the $pm \times qn$ block matrix:*

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}.$$

By definition, considering $\mathbf{X} = (x_1, x_2)$, his 2-Kronecker power is given by: $\mathbf{X}^{[2]} = (x_1^2, x_1x_2, x_2x_1, x_2^2)$.

Thanks to the same terms reduction procedure less terms respect to all the ones generated from the Kronecker power can be considered:

$$\mathbf{X}^{[2]} = \left(x_1^2, x_1x_2, x_2x_1, x_2^2\right) \rightarrow \left(x_1^2, x_1x_2, x_2^2\right). \tag{1.3}$$

It is due to the fact that considering same terms reduction permits to reduce the order of the vectors and matrices involved in the method that is going to be explain (the Taylor map approximation procedure). For example $\mathbf{X}^{[3]} = (x_1^3, x_1^2x_2, x_1x_2^2, x_2^3)$.

The (1.2) can be truncated to obtain a polynomial approximation of the solution of the system. In particular :

$$\mathbf{F} \approx \sum_{k=0}^{n} P_k \mathbf{X}^{[k]}. \tag{1.4}$$

Let's now introduce the definition of Taylor map:

Definition 2. *The transformation $\mathcal{M} : \mathbf{X}_0 = \mathbf{X}(t_0) \rightarrow \mathbf{X}(t_1)$ defines a Taylor map in form of*

$$\mathbf{X}(t_1) = W_0 + W_1\mathbf{X}_0 + W_2\mathbf{X}_0^{[2]} + \ldots + W_k\mathbf{X}_0^{[k]}, \tag{1.5}$$

where $\mathbf{X}, \mathbf{X}_0 \in R^n$ matrices W_i are weights, and $\mathbf{X}^{[k]}$ means the k-th Kronecker power of vector $\mathbf{X}$ with the same terms reduction.

As said before, it is possible to think to it as a polynomial regression with respect to the

vector $\mathbf{X}$, where the coefficients W_i are called "weights".

It is a powerful relation that, in the context of dynamical system, is capable of representing the state vector $\mathbf{X}$ at time t as function of the state $\mathbf{X_0}$ (i.e. the state evaluated at the time t_0) and the weights matrices.

The (1.5) can be also viewed as an iterative method: if the time interval $[t_0, T]$ is discretized with constant time step Δt, having the value of the state $\mathbf{X}$ at t_0 (i.e. $\mathbf{X_0}$), it is possible to find the value of the state at $t_1 = t_0 + \Delta t$ (i.e. $\mathbf{X_1}$). Iteratively the TM permits to generate a succession of point which are the state evaluated at each node of time t_i, as we can see in Figure 1.1.

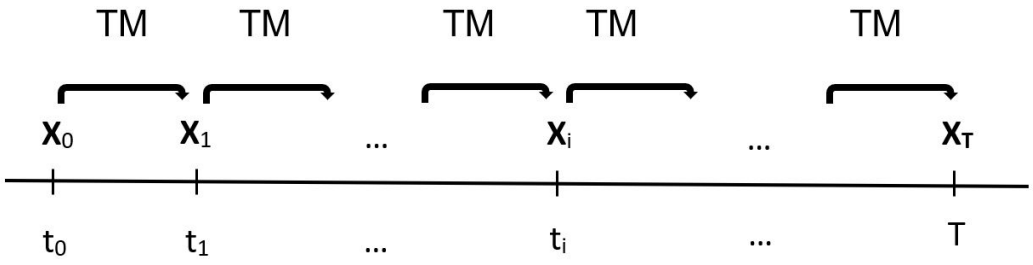

Figure 1.1: Iterative method based on TM.

1.1.1. Evaluation of the weights

Let's see how the weights matrices can be computed. In order to obtain their values the proposed steps could be followed:

1. by starting from the (1.1), it is necessary to re-write the right hand side in a truncated polynomial form (see (1.4)). To do this the assumption that $\mathbf{F}$ can be expanded in a Taylor series is a key point. The level n of accuracy of the approximation can be choose, in this work it is consider up to a third order. To understand consider these two examples: the free fall of a body with friction coefficient k and gravity constant g can be represented by the equation

$$v' = g - \frac{k}{m}v^2,$$

that is already written in his polynomial form; conversely the problem of the simple pendulum with wire length L is modelized by the equation

$$\phi'' = -g\sin(\phi)/L,$$

which does not appear in polynomial form, but considering the fact that the sinus function can be expanded by the Taylor expansion as follows:

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1},$$

it is possible to expand the right hand side to end up with this final polynomial equation:

$$\phi'' = -\frac{g}{L}\phi + \frac{g}{6L}\phi^3;$$

2. by writing each term of the sum in (1.4), it is possible to rework it as follows:

$$\frac{d}{dt}\mathbf{X} = P_0(t) + P_1(t)\mathbf{X} + P_2(t)\mathbf{X}^{[2]} + \ldots + P_n(t)\mathbf{X}^{[n]}. \tag{1.6}$$

By substituting each $\mathbf{X}$ in (1.6) in both right and left hand side with the TM (1.5), the right hand side becomes something of the form:

$$\begin{aligned} &P_0(t)+ \\ &P_1(t)\left(W_0(t) + W_1(t)\mathbf{X}_0 + \cdots + W_k(t)\mathbf{X}_0^{[k]}\right) + \\ &P_2(t)\left(W_0(t) + W_1(t)\mathbf{X}_0 + \cdots + W_k(t)\mathbf{X}_0^{[k]}\right)^{[2]} + \ , \\ &\cdots + \\ &P_n(t)\left(W_0(t) + W_1(t)\mathbf{X}_0 + \cdots + W_k(t)\mathbf{X}_0^{[k]}\right)^{[n]} \end{aligned} \tag{1.7}$$

whereas the left hand side has the following form:

$$\frac{d}{dt}W_0(t) + \ldots + \frac{d}{dt}W_k(t)\mathbf{X}_0^{[k]}; \tag{1.8}$$

3. by comparing the lhs (1.8) with the rhs (1.7) a new system of ordinary differential equations with respect to the weight matrices W_i arises:

$$\frac{d}{dt}W_i = f_i\left(W_0, \ldots, W_k, P_0, \ldots P_k\right), \tag{1.9}$$

where f_i are functions of matrices W_i and P_i. For instance, $f_0 = P_0 + P_1W_0 + \ldots P_kW_0^{[k]}$.

Since the Taylor Map transformation is assumed to be valid for any initial value $\mathbf{X}_0$, the last equation for W_i does not depend on $\mathbf{X}_0$. Moreover, it can be solved only once with the unified initial condition $W_0 = 0, W_1 = I, W_k = 0, k > 1$ with I

as identity matrix (these conditions guarantee that the equality (1.5) is verified for $t = t_0$).

4. by solving the system (1.9), the weights matrices are computed. In this work it is solved by using forward Euler algorithm in the interval of time $(0, \Delta t)$.

   The important thing to keep in mind is that the values of the weights are used in the iterative method of Figure 1.1 as constant coefficients, so it is sufficient to evaluate their values at the time $t = \Delta t$.

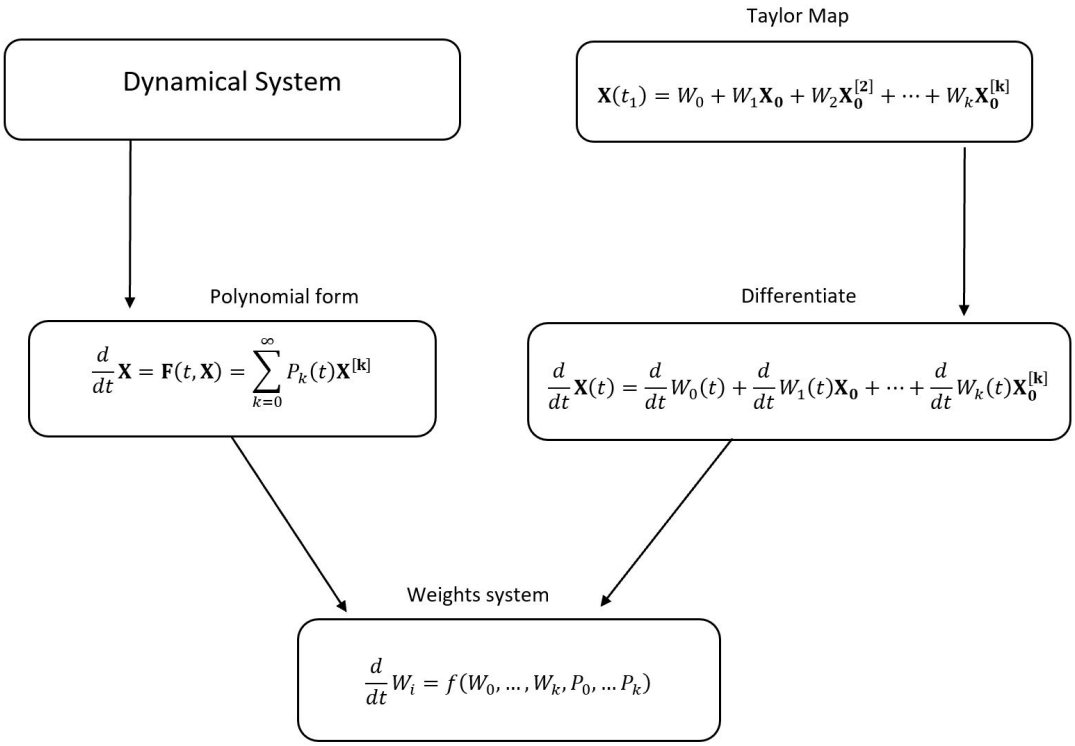

Figure 1.2: Scheme of the method.

1.2. Test cases

In this section some test cases are considered: first the matrices of weights are found by solving the system of ODEs (1.9), then the solutions are found using the iterative method of Figure 1.1 (only one or at least two dimensional problems are considered).

In particular, for the fact that the code is written differently from the one used in the papers [4], [6] and [7], a comparison between the two types of weights is done.

Finally the validity of the results are analyzed, considering the solutions of the papers and using as reference solution the one found with Runge-Kutta of fourth order.

1.2.1. Free fall of a body

Let us begin with a simple 1D example: consider the model of a free falling body with air resistance coefficients:

$$v' = g - \frac{k}{m}v^2 \tag{1.10}$$

where v is the velocity, m is the mass of the body, $g = 9.8$ kg/m^2 is the gravitational acceleration, $k = 0.392$ kg/m is the friction coefficient and $'$ represents the derivative with respect to the time variable.

The traditional approach for solving it is using numerical solvers. For instance, one of the most classical is the forward Euler method with constant time step.

By considering the following initialization of the coefficients of the system: $\Delta t = 0.1s$, $m = 100$ kg and $v(t_0) = v_0 = 0$m/s, it results:

$$v(t + \Delta t) = g\Delta t + v(t) - kv(t)^2\Delta t/m = 0.98 + v(t) - 0.000392v(t)^2. \tag{1.11}$$

The Euler method produces a Taylor map (1.11) of the second order that is calculated from a first-order discretization of (1.10).

On the other hand, by using the described TM algorithm, one can estimate the second-order Taylor map, which consists in a more accurate approximation:

$$v(t + \Delta t) = 0.979874527013 + 0.999615938364v_0 - 0.0003842685780960v_0^2. \tag{1.12}$$

Let's see in detail how it is possible to find (1.12), following the steps listed in section 1.1.

The first thing to do is to see if the ordinary differential equation 1.10 is written in a polynomial form, else it is necessary to perform a Taylor expansion. In this case it is already written as polynomial with coefficients:

$$P_0 = g, \quad P_1 = 0, \quad P_2 = -\frac{k}{m}. \tag{1.13}$$

Consider now the TM up to the second order:

$$\mathbf{X}\,(t_1) = W_0 + W_1\mathbf{X}_0 + W_2\mathbf{X}_0^{[2]}, \tag{1.14}$$

where $\mathbf{X} = v$ and $\mathbf{X_0} = v_0$ are scalars (this implies that the Kronecker powers are simple powers).

By differentiating the (1.14) and replacing it in lhs of (1.10) and re-writing the rhs of

(1.10) in his polynomial form using the coefficients (1.13), one can obtain:

$$W_0' + W_1' v_0 + W_2' v_0^2 = P_0 + P_1 v + P_2 v^2. \tag{1.15}$$

Now it is necessary to substitute v and v^2 in the rhs of (1.15) with his correspondent TM (i.e. (1.14)) to obtain a rhs as follows:

$$P_0 + P_1(W_0 + W_1 v_0 + W_2 v_0^2) + P_2(W_0 + W_1 v_0 + W_2 v_0^2)^2, \tag{1.16}$$

by omitting the terms higher than second order and by grouping together the coefficients of v_0 and v_0^2, it is possible to end up with a rhs of the following form:

$$\begin{aligned} P_0 + P_1 W_0 + P_2 W_0^2 + \\ (P_1 W_1 + 2P_2 W_0 W_1) v_0 + \\ (P_1 W_2 + P_2 W_1^2 + 2P_2 W_0 W_2) v_0^2. \end{aligned} \tag{1.17}$$

By comparing the terms of (1.17) with the terms of the lhs of (1.15), a new system of ordinary differential equations with respect to the weights matrices W_i arises.

It can be solved only once with the initial condition $W_0 = 0, W_1 = I, W_k = 0, k > 1$ with I as identity matrix.
To compare the accuracy of the two solutions one can use the analytical one:

$$v(t) = \sqrt{\frac{mg}{k}} \tanh\left(t\sqrt{\frac{kg}{m}}\right).$$

By observing Figure 1.3 and Figure 1.4 it is possible to do some interesting considerations: the solutions are quit similar, but the cumulative square error shows that the one found using the TM approach is more accurate.

In Figure 1.4 there is the plot of the Mean Square Error: for low values of Δt both Euler and TM method are going to zero, the interesting fact is that it seems like the TM algorithm generate a solution (green line) whose MSE is lower than the one found used the other method, if the value of Δt is higher (see [7]).

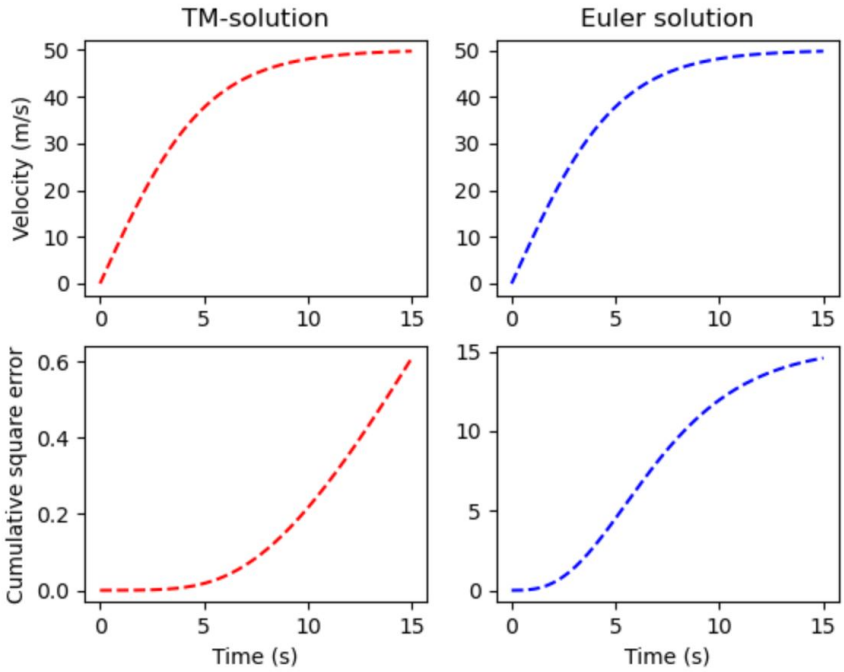

Figure 1.3: Left: TM solution; Right: Euler solution.

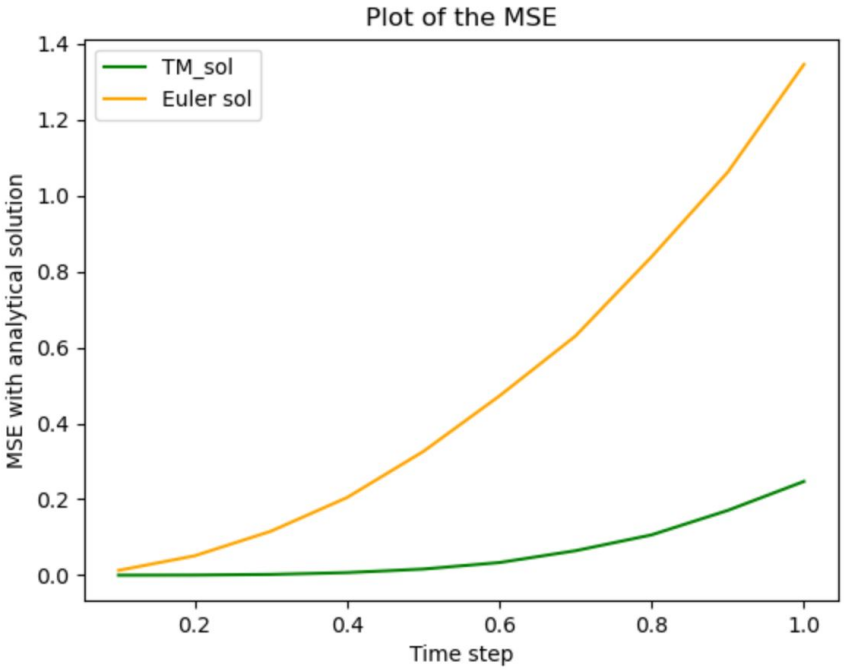

Figure 1.4: MSE between the analytical solution.

1.2.2. Pendulum

The physics-based model of the simple pendulum can be described by the differential equation:

$$\phi'' = -g \cdot \sin(\phi)/L, \tag{1.18}$$

where ϕ is the angle of the pendulum (measured with respect to the vertical), $'$ is the derivative with respect to the time variable, g is the gravitational acceleration and L is the length of the wire.

This equation can be written in a matrix form up to the third order nonlinearities, by applying the Taylor expansion of the sinus function, as made in section 1.1:

$$\frac{d}{dt}\begin{pmatrix}\phi\\ \phi'\end{pmatrix} = \begin{pmatrix}0 & 1\\ -g/L & 0\end{pmatrix}\begin{pmatrix}\phi\\ \phi'\end{pmatrix} + \begin{pmatrix}0 & 0 & 0 & 0\\ g/(6L) & 0 & 0 & 0\end{pmatrix}\begin{pmatrix}\phi^3\\ \phi^2\phi'\\ \phi\phi'^2\\ \phi'^3\end{pmatrix} =$$

$$P_1\begin{pmatrix}\phi\\ \phi'\end{pmatrix} + P_3\begin{pmatrix}\phi^3\\ \phi^2\phi'\\ \phi\phi'^2\\ \phi'^3\end{pmatrix}. \tag{1.19}$$

The mathematical description of the pendulum leads to a theoretically continuous oscillation with the same amplitude and frequency indefinitely.

Though the behaviour differs from the real damped oscillation, this simplified physics-based model can still initialize the PNN with some level of accuracy.

Let's calculate a Taylor map for the time step $\Delta t = 0.1$ s by following the algorithm presented in section 1.1:

$$\mathcal{M}(t) : \begin{pmatrix}\phi\\ \phi'\end{pmatrix} = W_1\begin{pmatrix}\phi_0\\ \phi'_0\end{pmatrix} + W_2\begin{pmatrix}\phi_0^2\\ \phi_0\phi'_0\\ \phi'^2_0\end{pmatrix} + W_3\begin{pmatrix}\phi_0^3\\ \phi_0^2\phi'_0\\ \phi_0\phi'^2_0\\ \phi'^3_0\end{pmatrix}. \tag{1.20}$$

By denoting $\mathbf{X} = (\phi, \phi')\,, \mathbf{X}_0 = (\phi_0, \phi'_0)$ and by substituting the Taylor Map (equation (1.20)) into the dynamical system written in polynomial form (equation (1.19)), one can write:

$$\mathbf{X}' = P_1W_1\mathbf{X}_0 + P_1W_2\mathbf{X}_0^{[2]} + \left(P_1W_3 + P_3W_1^{[3]}\right)\mathbf{X}_0^{[3]} + \mathcal{O}\left(\mathbf{X}_0^{[3]}\right), \tag{1.21}$$

where $W_1^{[3]}$ can be calculated from the relation $(W\mathbf{X})^{[3]} = W^{[3]}\mathbf{X}^{[3]}$. For instance, for

$W = \{w_{ij}\}$,

$$(W\mathbf{X})^{[3]} = \begin{pmatrix} w_{11}\phi + w_{12}\phi' \\ w_{21}\phi + w_{22}\phi' \end{pmatrix}^{[3]} =$$
$$\begin{pmatrix} w_{11}^3 & 3w_{11}^2 w_{12} & 3w_{11}w_{12}^2 & w_{11}^3 \\ (w_{11}^2 w_{21} & w_{11}^2 w_{22} + 2w_{11}w_{12}w_{21} & w_{12}^2 w_{21} + 2w_{11}w_{12}w_{22} & w_{12}^2 w_{22} \\ w_{21}^2 w_{11} & w_{21}^2 w_{12} + 2w_{21}w_{22}w_{11} & w_{22}^2 w_{11} + 2w_{21}w_{22}w_{12} & w_{22}^2 w_{12} \\ w_{21}^3 & 3w_{21}^2 w_{22} & 3w_{21}w_{22}^2 & w_{22}^3 \end{pmatrix} \begin{pmatrix} \phi^3 \\ \phi^2\phi' \\ \phi\phi'^2 \\ \phi'^3 \end{pmatrix} = W^{[3]}\mathbf{X}^{[3]}$$

Taking the derivative of (1.20) and comparing it with (1.21), one can obtain a system of ODEs that do not depend on $\mathbf{X}_0$:

$$W_1' = P_1 W_1, \quad W_2' = P_1 W_2, \quad W_3' = P_1 W_3 + P_3 W_1^{[3]}. \tag{1.22}$$

By solving it for the time interval $[0; 0.1]$, with the initial conditions $W_1(0) = I, W_2(0) = 0, W_3(0) = 0$ and I as the identity matrix, it results in a Taylor map that describes the dynamics of the ideal pendulum during $\Delta t = 0.1$. For instance, for $g = 9.8\ \mathrm{kg/m^2}$ and $L = 0.3\mathrm{m}$, the solution up to the two digits is:

$$W_1 = \begin{pmatrix} 0.84 & 0.09 \\ -3.1 & 0.84 \end{pmatrix}, \quad W_2 = 0,$$

$$W_3 = \begin{pmatrix} 2.2 \cdot 10^{-3} & 2.15 \cdot 10^{-4} & 1.08 \cdot 10^{-5} & 2.24 \cdot 10^{-7} \\ 3.94 \cdot 10^{-2} & 5.89 \cdot 10^{-3} & 4.08 \cdot 10^{-4} & 1.08 \cdot 10^{-5} \end{pmatrix}.$$

Comparison between the two type of weights

Now a comparison between the weights of the paper [7] and those found by the code implemented is done: it is useful to understand if the code performs well or not (remember that the code is written independently). Here there are the weights of the paper [7]:

$$W_{1p} = \begin{pmatrix} 0.84 & 0.09 \\ -3.1 & 0.84 \end{pmatrix},$$

$$W_{3p} = \begin{pmatrix} 0.02 & 0.0023 & 0.00012 & 2.3 \cdot 10^{-6} \\ 0.43 & 0.064 & 0.0044 & 0.00012 \end{pmatrix}.$$

The following are the weights found by the code:

- $\Delta t = 0.1/100$

$$W_1 = \begin{pmatrix} 0.842 & 0.095 \\ -3.097 & 0.842 \end{pmatrix},$$

$$W_3 = \begin{pmatrix} 2.19 \cdot 10^{-3} & 2.10 \cdot 10^{-4} & 1.05 \cdot 10^{-5} & 2.15 \cdot 10^{-7} \\ 3.96 \cdot 10^{-2} & 5.87 \cdot 10^{-3} & 4.04 \cdot 10^{-4} & 1.07 \cdot 10^{-5} \end{pmatrix}.$$

$\|W1p - W1\|_2 = 0.0067$ $\|W3p - W3\|_2 = 0.39512$

- $\Delta t = 0.1/1000$

$$W_1 = \begin{pmatrix} 0.841 & 0.095 \\ -3.092 & 0.841 \end{pmatrix},$$

$$W_3 = \begin{pmatrix} 2.20 \cdot 10^{-3} & 2.14 \cdot 10^{-4} & 1.08 \cdot 10^{-5} & 2.24 \cdot 10^{-7} \\ 3.94 \cdot 10^{-2} & 5.89 \cdot 10^{-3} & 4.07 \cdot 10^{-4} & 1.08 \cdot 10^{-5} \end{pmatrix}.$$

$\|W1p - W1\|_2 = 0.00924$ $\|W3p - W3\|_2 = 0.39535$

- $\Delta t = 0.1/10000$

$$W_1 = \begin{pmatrix} 0.841 & 0.095 \\ -3.092 & 0.841 \end{pmatrix},$$

$$W_3 = \begin{pmatrix} 2.2 \cdot 10^{-3} & 2.15 \cdot 10^{-4} & 1.09 \cdot 10^{-5} & 2.25 \cdot 10^{-7} \\ 3.94 \cdot 10^{-2} & 5.90 \cdot 10^{-3} & 4.08 \cdot 10^{-4} & 1.09 \cdot 10^{-5} \end{pmatrix}.$$

$\|W1p - W1\|_2 = 0.00959$ $\|W3p - W3\|_2 = 0.39537$

Many steps of the Euler method for the resolution of the system (1.22) are considered. It seems that the norm between the two matrices is not really influenced by the number of steps used and a curious fact is that the little difference could be due to a different method for the resolution of the ODEs of the weights.

As it possible to see in Figures 1.5 and 1.6 it seems that the solutions obtained with the two types of weights are the same.

Evaluation of the error using Runge-Kutta

In this section there is an evaluation of the error, considering the solution found by Runge-Kutta as reference solution. The initial point is $\mathbf{x_0} = (0.3, 0)$.

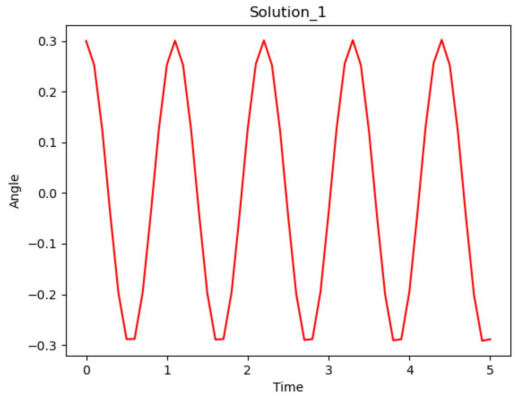

Figure 1.5: Solution$_1$.

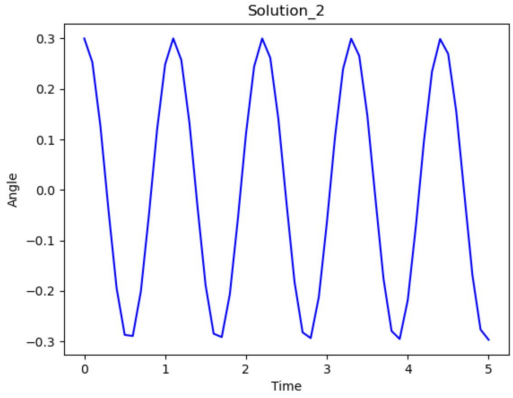

Figure 1.6: Solution$_2$

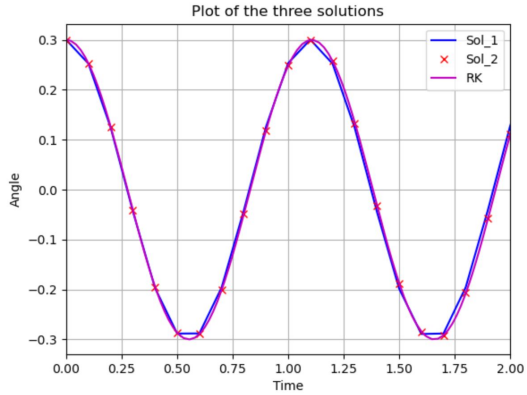

Figure 1.7: Plot of the angle.

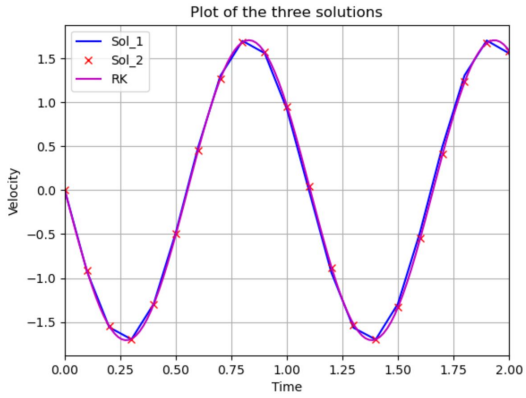

Figure 1.8: Plot of the velocity.

By the observation of Figures 1.7 and 1.8 it is clear that the method is finding the right values of the solution: although there is a little difference in the W_3 matrix, the two types of solution (solution$_1$: code weights, solution$_2$: paper weights) are similar and have the same path of the one found by Runge-Kutta.

Using a $\Delta t = 0.1$ the obtained solution is not so smooth, but it is enough because the main purpose of this work is to build a Neural Network, so the weights will be update in the training phase.

Here it is an estimate of the L_2 norm between the two solutions and the reference one :

$$\|RK_x - sol1_x\|_{L2} = 0.1382 \qquad \|RK_x - sol2_x\|_{L2} = 0.04420$$

$$\|RK_y - sol1_y\|_{L2} = 0.8699 \qquad \|RK_y - sol2_y\|_{L2} = 0.3035$$

1.2.3. Van der Pol

In dynamics, the Van der Pol oscillator is a non-conservative oscillator with non-linear damping. It evolves in time according to the second-order differential equation:

$$\frac{d^2x}{dt^2} - \mu\left(1 - x^2\right)\frac{dx}{dt} + x = 0,$$

where x is the position coordinate which is a function of the time variable t and μ is a scalar parameter indicating the strength of the damping.

For simplicity, let's set the value of the coefficient μ to be equal 1; it leads to the equation:

$$x'' = x' - x - x^2x',$$

that can be presented in the form of the system:

$$\begin{aligned} x' &= y \\ y' &= y - x - x^2y. \end{aligned}$$

This system is already written in a polynomial form, indeed it is possible to write:

$$\frac{d}{dt}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{pmatrix}\begin{pmatrix} x^3 \\ x^2y \\ xy^2 \\ y^3 \end{pmatrix} =$$

$$P_1\begin{pmatrix} x \\ y \end{pmatrix} + P_3\begin{pmatrix} x^3 \\ x^2y \\ xy^2 \\ y^3 \end{pmatrix}. \tag{1.23}$$

The corresponding TM is as follows:

$$\mathbf{X}_{i+1} = \mathcal{M} \circ \mathbf{X}_i = W_0 + W_1\begin{pmatrix} x_i \\ y_i \end{pmatrix} + W_2\begin{pmatrix} x_i^2 \\ x_iy_i \\ y_i^2 \end{pmatrix} + W_3\begin{pmatrix} x_i^3 \\ x_i^2y_i \\ x_iy_i^2 \\ y_i^3 \end{pmatrix}. \tag{1.24}$$

By denoting $\mathbf{X} = (x, y), \mathbf{X}_0 = (x_0, y_0)$ and by substituting the Taylor Map (equation (1.24)) into the dynamical system written in polynomial form (equation (1.23)), one can

write:

$$\mathbf{X}' = P_1 W_1 \mathbf{X}_0 + P_1 W_2 \mathbf{X}_0^{[2]} + \left(P_1 W_3 + P_3 W_1^{[3]}\right) \mathbf{X}_0^{[3]} + \mathcal{O}\left(\mathbf{X}_0^{[3]}\right). \tag{1.25}$$

Taking the derivative of the equation (1.24) and comparing it with the equation (1.23), one can obtain the system of ODEs in the weights variable:

$$W_1' = P_1 W_1, \quad W_2' = P_1 W_2, \quad W_3' = P_1 W_3 + P_3 W_1^{[3]}. \tag{1.26}$$

By solving it for the time interval $[0; 0.01]$, with the initial conditions $W_1(0) = I, W_2(0) = 0, W_3(0) = 0$ and I as the identity matrix, the weights of the iterative method of Figure 1.1 are computed.

Comparison between the weights

Let's see if there are relevant differences in the two types of matrices of weights. The weights of the paper([4]) :

$$W_{1p} = \begin{pmatrix} 0.999 & 0.010 \\ -0.010 & 1.001 \end{pmatrix},$$

$$W_{3p} = \begin{pmatrix} 1.59 \cdot 10^{-7} & -4.95 \cdot 10^{-5} & -3.21 \cdot 10^{-7} & -7.91 \cdot 10^{-10} \\ 4.95 \cdot 10^{-5} & -1.01 \cdot 10^{-2} & -9.96 \cdot 10^{-5} & -3.30 \cdot 10^{-7} \end{pmatrix}.$$

The following are the weights found by the code:

- $\Delta t = 0.1/100$

$$W_1 = \begin{pmatrix} 1.00 & 0.01 \\ -0.01 & 1.001 \end{pmatrix},$$

$$W_3 = \begin{pmatrix} 1.63 \cdot 10^{-7} & -4.98 \cdot 10^{-5} & -3.27 \cdot 10^{-7} & -8.11 \cdot 10^{-10} \\ 4.98 \cdot 10^{-5} & -1.01 \cdot 10^{-2} & -1.00 \cdot 10^{-4} & -3.34 \cdot 10^{-7} \end{pmatrix}.$$

$\|W1p - W1\|_2 = 5.7211e-07$ $\|W3p - W3\|_2 = 1.0637e-06$

- $\Delta t = 0.1/1000$

$$W_1 = \begin{pmatrix} 1.000 & 0.010 \\ -0.010 & 1.001 \end{pmatrix},$$

$$W_3 = \begin{pmatrix} 1.67 \cdot 10^{-7} & -5.03 \cdot 10^{-5} & -3.36 \cdot 10^{-7} & -8.42 \cdot 10^{-10} \\ 5.03 \cdot 10^{-5} & -1.01 \cdot 10^{-2} & -1.01 \cdot 10^{-4} & -3.39 \cdot 10^{-7} \end{pmatrix}.$$

$\|W1p - W1\|_2 = 1.3541e-06$ $\|W3p - W3\|_2 = 2.5008e-06$

- $\Delta t = 0.1/10000$

$$W_1 = \begin{pmatrix} 1.000 & 0.010 \\ -0.010 & 1.001 \end{pmatrix},$$

$$W_3 = \begin{pmatrix} 1.67 \cdot 10^{-7} & -5.03 \cdot 10^{-5} & -3.37 \cdot 10^{-7} & -8.45 \cdot 10^{-10} \\ 5.03 \cdot 10^{-5} & -1.01 \cdot 10^{-2} & -1.01 \cdot 10^{-4} & -3.39 \cdot 10^{-7} \end{pmatrix}.$$

$\|W1p - W1\|_2 = 1.4323e-06$ $\|W3p - W3\|_2 = 2.6446e-06$

Evaluation of the error using Runge-Kutta

Considering as initial point $\mathbf{x_0} = (1, 2)$, as we can see in Figures 1.9-1.14, the two solutions are quit identical and by observing the L_2 norm between them and the reference one, it is possible to see that the error is small, a sign that the method is generating accurate weights values.

$\|RK_x - sol1_x\|_{L2} = 1.16920$ $\|RK_x - sol2_x\|_{L2} = 1.16920$

$\|RK_y - sol1_y\|_{L2} = 1.6709$ $\|RK_y - sol2_y\|_{L2} = 1.6709$

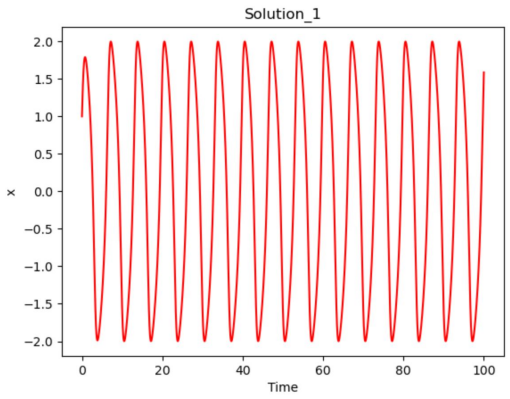

Figure 1.9: Solution$_1$

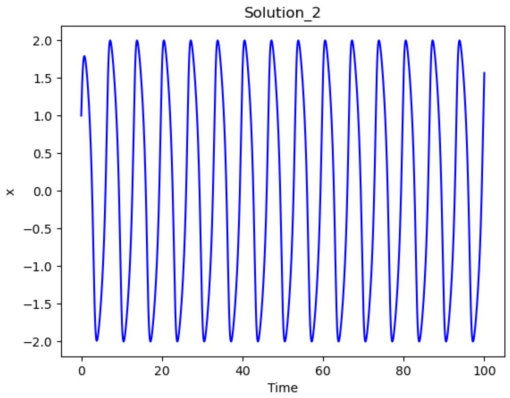

Figure 1.10: Solution$_2$

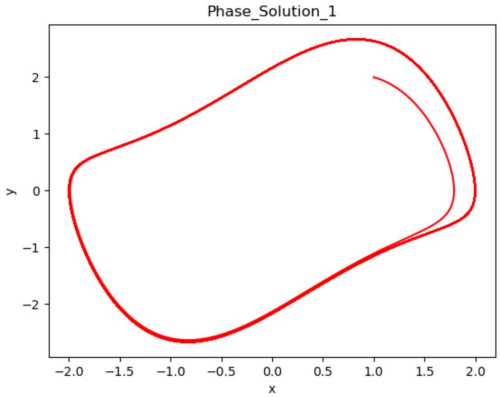

Figure 1.11: Phase of sol$_1$

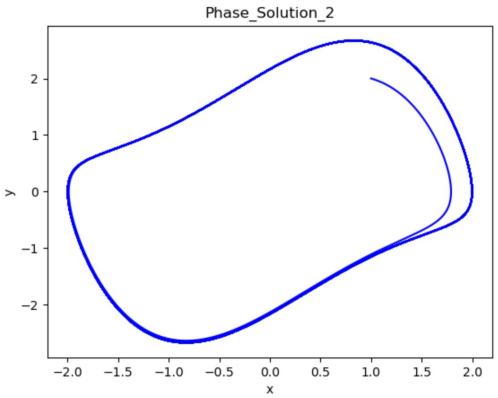

Figure 1.12: Phase of sol$_2$

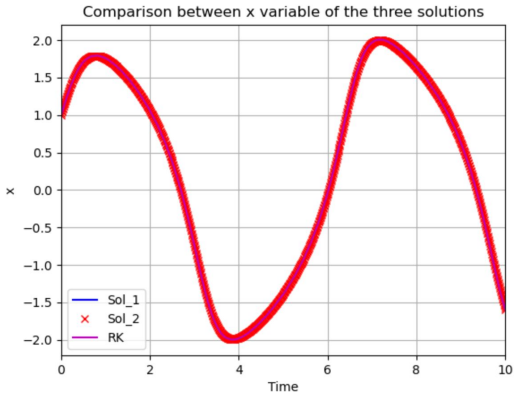

Figure 1.13: Comparison between the three solutions

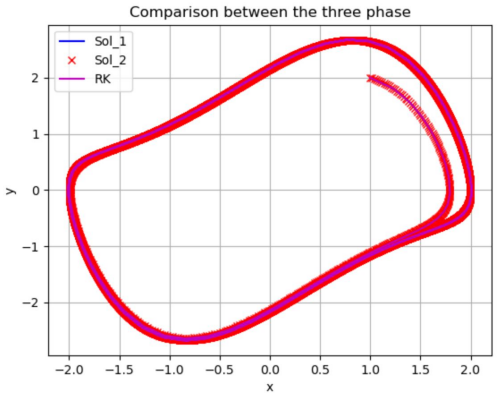

Figure 1.14: Comparison between the three phases

1.2.4. Charged particle

The following problem is important for this work because it represents the only one example having all the matrices of weights different from zero, indeed, for the previous examples, the W_2 was null.

The particle dynamics in the electromagnetic fields can be described by a system of ODEs that has a complex nonlinear form. For simplicity, let's consider an approximation of particle motion in cylindrical deflector written in form of:

$$\begin{aligned} x' &= y \\ y' &= -2x + x^2/R \end{aligned}$$

where R is equilibrium radius of particle bending, x is deviation from this radius, and $x' = y$ is derivative on bending angle with respect to the time variable t.

For example, let's consider a deflector with $R = 10$ m that rotates a reference particle with initial conditions $x = 0, x' = 0$ on angle $\pi/4$. For simulation, the case of dynamics of particle with nontrivial initial conditions is considered, because it leads to particle oscillation.

Let's find a 3rd order Taylor map that transform initial particle state $\mathbf{X}_0 = (x_0; y_0)$ at the entrance of the deflector to the resulting state $\mathbf{X}_1 = (x_1; y_1)$ at the end of it,

$$\begin{pmatrix} x \\ y \end{pmatrix} = W_1 \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + W_2 \begin{pmatrix} x_0^2 \\ x_0 y_0 \\ y_0^2 \end{pmatrix} + W_3 \begin{pmatrix} x_0^3 \\ x_0^2 y_0 \\ x_0 y_0^2 \\ y_0^3 \end{pmatrix}. \tag{1.27}$$

Writing in polynomial form the dynamical system, to underline the matrices of coefficients:

$$\begin{pmatrix} x \\ y \end{pmatrix}' = \begin{pmatrix} 0 & 1 \\ -2 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 1/R & 0 & 0 \end{pmatrix} \begin{pmatrix} x^2 \\ xy \\ y^2 \end{pmatrix}. \tag{1.28}$$

Now substituting the equation (1.27) in (1.28), then differentiate the Taylor Map (equation (1.27)). By grouping the like terms of powers of x_0 and y_0, it is possible to obtain a system of ODEs that do not depends on $\mathbf{X}_0 = (x_0; y_0)$ and represents dynamics of weight matrices

$$\begin{aligned} W_1' &= f_1(W_1, W_2, W_3), \quad W_1(0) = I \\ W_2' &= f_2(W_1, W_2, W_3), \quad W_2(0) = 0 \\ W_3' &= f_3(W_1, W_2, W_3), \quad W_3(0) = 0 \end{aligned}$$

where f_i is functions arising after like terms grouping. By integrating this system during the interval $[0; \pi/4]$, we can receive the desired map. For example, the map up to two digits is:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} 0.44 & 0.63 \\ -0.13 \cdot 10 & 0.44 \end{pmatrix} \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} 0.23 \cdot 10^{-1} & 0.12 \cdot 10^{-1} & 0.26 \cdot 10^{-2} \\ 0.40 \cdot 10^{-1} & 0.35 \cdot 10^{-1} & 0.12 \cdot 10^{-1} \end{pmatrix} \begin{pmatrix} x_0^2 \\ x_0 y_0 \\ y_0^2 \end{pmatrix} + \begin{pmatrix} 0.21 \cdot 10^{-3} 0.17 \cdot 10^{-3} 0.47 \cdot 10^{-4} 0.56 \cdot 10^{-5} \\ 0.83 \cdot 10^{-3} 0.95 \cdot 10^{-3} 0.32 \cdot 10^{-3} 0.47 \cdot 10^{-4} \end{pmatrix} \begin{pmatrix} x_0^3 \\ x_0^2 y_0 \\ x_0 y_0^2 \\ y_0^3 \end{pmatrix} . \tag{1.29}$$

Comparison between the weights

Let's compare the weights of the paper (reported in (1.29), taken from [6], with the weights found by the code, which are as follows:

$\Delta t = (\pi/4)/1000$

$$W_1 = \begin{pmatrix} 0.44 & 0.63 \\ -1.27 & 0.44 \end{pmatrix}, \quad W_2 = \begin{pmatrix} 0.023 & 0.012 & 0.003 \\ 0.04 & 0.035 & 0.012 \end{pmatrix},$$

$$W_3 = \begin{pmatrix} 2.074 \cdot 10^{-4} & 1.743 \cdot 10^{-4} & 4.64 \cdot 10^{-5} & 5.62 \cdot 10^{-6} \\ 8.26 \cdot 10^{-4} & 9.526 \cdot 10^{-4} & 3.151 \cdot 10^{-4} & 4.656 \cdot 10^{-5} \end{pmatrix}.$$

$$\|W1p - W1\|_2 = 3.287e-2 \qquad \|W2p - W2\|_2 = 5.114e-4$$

$$\|W3p - W3\|_2 = 8.447e-6$$

It is possible to see that the values are almost the same, indeed by the computation of the norm it is possible to sew low values of the errors, a fact that ensures that the algorithm is able to find correct values of weights.

Evaluation of the error using Runge-Kutta

Starting from the initial point $\mathbf{x_0} = (0.2, 0.1)$, as it is possible to see in Figure 1.15, the sol_1 (found by the code) and sol_2 are almost the same, and they are following the trend of the reference_sol that is built using Runge-Kutta.

By valuating the norm of the difference between the two type of solutions with the real one, it seems it is small, another confirmation that the code is working giving right weights values.

$$\|RK_x - sol1_x\|_{L2} = 2.3535 \qquad \|RK_x - sol2_x\|_{L2} = 2.3785$$

$$\|RK_y - sol1_y\|_{L2} = 3.2612 \qquad \|RK_y - sol2_y\|_{L2} = 3.3438$$

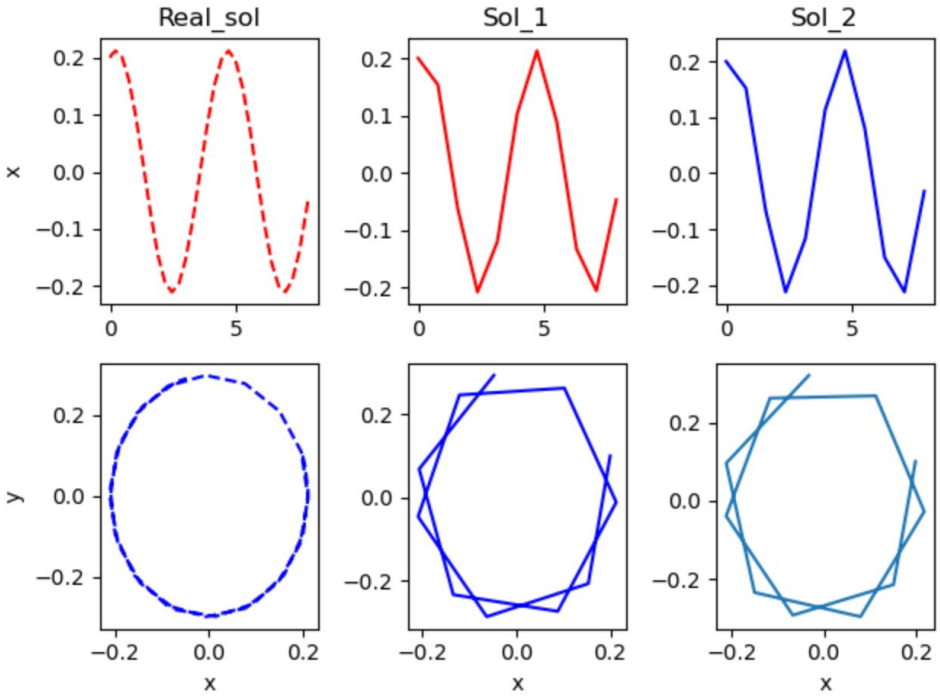

Figure 1.15: Comparison between the solutions.

1.3. Conclusion

This Chapter starts with a brief introduction to the theory of Taylor Map, followed by the explanation on how to implement the iterative method with which is possible to evaluate the solution of a given dynamical system.

Indeed by using the described procedure, that can evaluate the right matrices of weights, starting from a Physic-Based model in which the equations are known, it is possible to find his solution for every initial condition, because the weights are independent with respect to the initial condition of the system.

Subsequently there is a testing part where the results found by the code are in four cases (1-D or 2-D) are corroborate.

The interesting things that will be developed in the next Chapter are related to the following problems: what about a situation in which it is not possible to have information about the equations, but it is available a sort of output of a black-box model; What about a problem where is known how the physic works, so it is possible to directly write his equations, but there are some uncertainties related to the coefficients, due to some accidental errors or estimation problems.

2 | Taylor-Map Polynomial Neural Network

In this chapter we will see first of all a redesign of the previous iterative method as a Neural Network, which will be named "polynomial neural network (PNN)"; we will focus on how to build it.

Afterwards the predator-prey model of Lotka-Volterra is used as test in two steps: first there is a using of the new NN by imposing the right weights found by Taylor-Map approach to show that the effectiveness of the method; second, by treating the system as black-box model and by imagine that equations are unknown, there is the attempt to recover the important information about the physic dynamic by starting from only one initial solution. This is called "One-shot identification problem".

In this step there will be a comparison between the PNN and some other classical NN model, like the Long Short Term Memory(LSTM), which is intensively used in time series problem.

In conclusion we will find a fine-tuned problem: the purpose is to try to build a PNN to recover most information possible on the example of the pendulum, by initializing his weights through the TM and by making the training phase on the available data.

2.1. Polynomial Neural Network

In this section a redesign of the Taylor Map method used in an iterative way is shown; in particular the main information on how to build a special Neural Network called "polynomial neural network" are listed and explained; the PNN must evaluate the solution of a problem in the same way with respect to the previous method.

To understand the idea let's consider Figure 2.1: a 2-D vector that represents the value of the solution in a certain time i is given and imagine it and each of his Kronecker power as they were neurons of a neural network; then multiply these neurons by their corresponding weights matrices to get as many 2-D output vectors as the number of neurons; finally sum every results found in this way (every 2-D vectors) in a unique output vector.

This procedure is what characterizes the PNN, indeed it can be viewed as a function that transforms an input 2-D vector (that represent the solution at time $t = i$) in an output 2-D vector (that is the solution in the next time step $t = i + 1$).

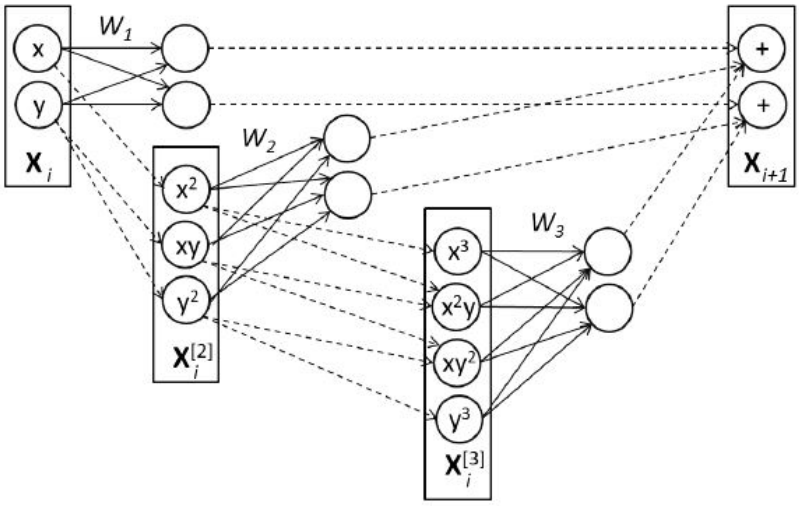

Figure 2.1: Polynomial Neural Network-neuron(from [7]).

An interesting revised of the TM is the following: in Chapter 1 the Taylor map is described as a polynomial regression with respect to the component of the state vector $\mathbf{X}$, but now it is possible to think to it as a regression with respect to the neurons.

2.1.1. Implementation

Let's have a look to the implementation of the PNN: as we can see in Figure 2.2, it is possible to imagine the PNN as a multi-output model composed by many $\mathcal{M}$-boxes whose work is to take in input the value of the solution in a certain time $t = i$ and return as output the value of the solution at the next time step $t = i + dt$.

To implement this multi-output model is sufficient to cycle many times (one for every time step) on one $\mathcal{M}$-box by changing the input at every iterate: by starting from $\mathbf{X}_0$ the box must return $\mathbf{X}(dt)$; then by taking as input the value of $\mathbf{X}(dt)$ just computed it must return $\mathbf{X}(dt + dt)$ and so on.

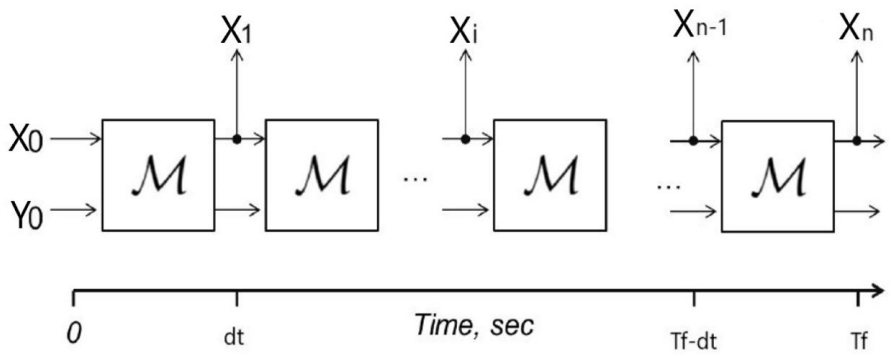

Figure 2.2: Polynomial Neural Network (adapted [7])

Let's have a look on how the $\mathcal{M}$ box works. For this purpose let's consider Figure 2.3: it consists in two helper layers that are Filter and Custom.

Filter is a layer built specializing the "layers" class of tensorflow.keras, what it does is to take as input a 2-D vector and returns, like it is possible to see, a 9-D vector which contains the components of the Kronecker powers up to the third order assembled all together.

Custom is another specialized layers whose work is to take the 9-D vector from Filter output and to multiply it with a matrices that contains all the weights evaluated using the TM method.

The output of this layer is a 2-D vector which contains the values of the solution at the next time step.

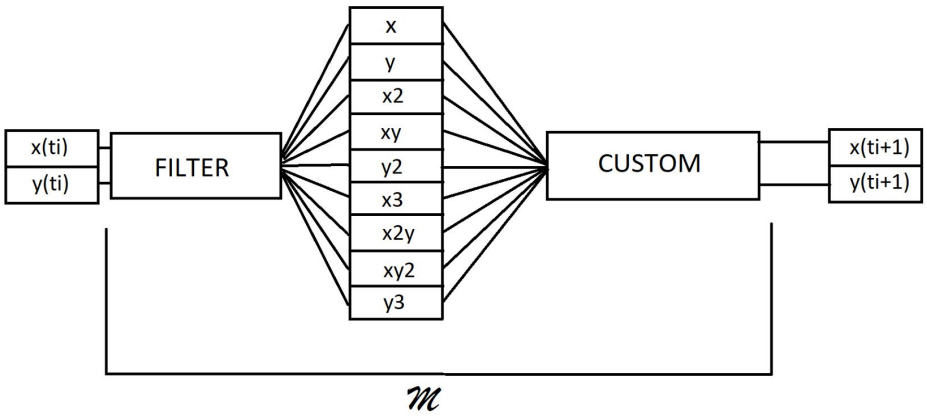

Figure 2.3: Structure of the $\mathcal{M}$ box

2.2. Application of the PNN

In this section we see an easy application of the PNN without the training phase, just to clarify that to apply this new NN corresponds to apply the iterative method seen in the previous Chapter.

Considering the Lotka-Volterra system (equation 2.1), the Taylor Map algorithm tested in Chapter 1 is used to evaluate the weights and these are used to initialize the values of the weights in the custom layer of Figure 2.3.

The situation is the following: the problem is known (Lotka-Volterra), so by using the equations it is possible to directly evaluate the values of the weights; then the PNN without any training phase (by setting the weights in the custom layer as untrainable in the code) returns the solution.

Also in this example the Runge-Kutta solution will be used as reference one, in order to perform some comparison. Let's consider the Lotka-Volterra system :

$$\begin{aligned} x' &= y + xy \\ y' &= -2x - xy, \end{aligned} \tag{2.1}$$

with two initial condition : $\mathbf{x_0} = (0.8, 0.8)$ and $\mathbf{x_1} = (0.5, 0.5)$, that correspond to the initial values of the solutions sol0 and sol1 respectively. By considering as final time $T_f = 5$ s and by taking 100 time steps, the PNN implemented evaluates, using the command 'predict' of Keras, the solution at each time step, starting with the initial condition and cycling over the total number of time steps.

As we can see in Figure 2.4 both the 'x vs time' (the first raw) and the 'phase' (second row) are quit similar with respect to the plot of the real solution; to confirm this fact notice that also the norms between the two solutions and the Runge-Kutta ones are small.

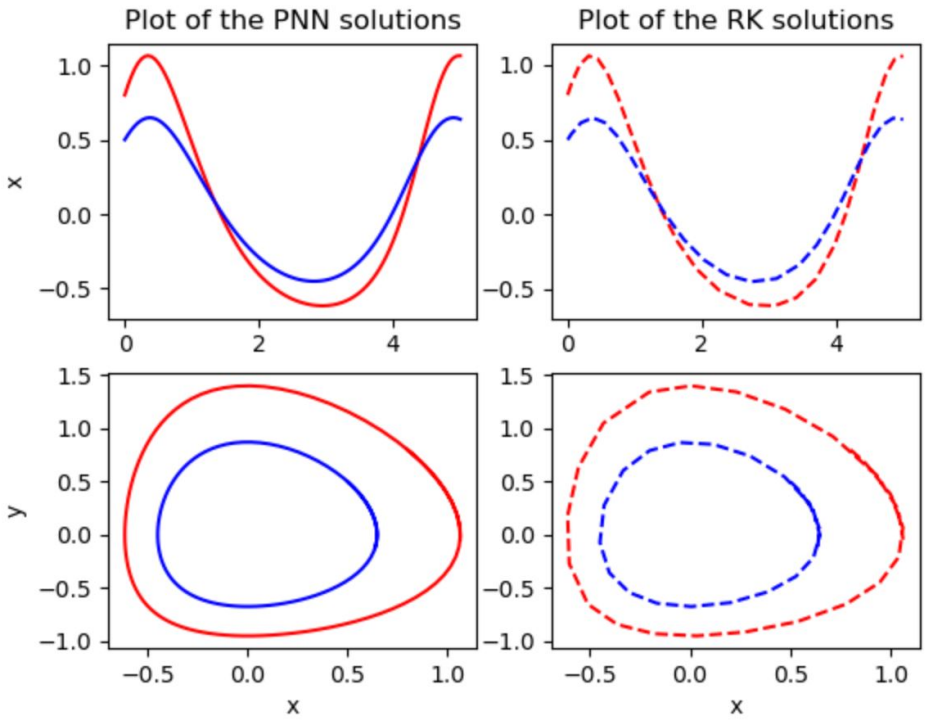

Figure 2.4: Left:solution with TM method; Right:RK solution

$$\|RK0_x - sol0_x\|_{L2} = 0.06889 \qquad \|RK0_y - sol0_y\|_{L2} = 0.11164$$

$$\|RK1_x - sol1_x\|_{L2} = 0.03851 \qquad \|RK1_y - sol1_y\|_{L2} = 0.05731$$

2.3. Identification problem

In this section we will try to learn the physic information of a dynamical system without knowing the equations, but thinking to it as a black-box model. This kind of problem is called "Identification problem".

The Taylor Map is not applicable at this point, but the main purpose is to recover the dynamics of the problem by using three types of neural network: MLP, TM-PNN and LSTM.

By starting with an initial point $\mathbf{X}_0 = (x_0, y_0)$, the discrete states $\{\mathbf{X}_i\}_{i=1,n}$ is calculated by integrating the differential equations using a Runge-Kutta (4-th order) method, with a constant time step.

After data is generated, the system of ODEs is not used in further training. Four different particular solutions are generated and they are presented as time series $\mathbf{X}(t_i) = (x_i, y_i)$.

As training set, consider only one solution which has as initial pointv$\mathbf{X}_0 = (0.5, 0.5)$, while three other solutions with initial coordinates $(0.8, 0.8), (0.1, 0.1)$, and $(0.0, 0.0)$ are used for testing.

The solutions are generated by integrating the equation from $t_0 = 0$ to $t = 4.65$ s with a constant time step $\Delta t = 0.01$ s. (all this preparation phase is summarized in Figure 2.5).

The problem is the following: is it possible to recover dynamics of the whole system for unseen inputs knowing only a particular training solution?

Consider three neural network architectures: proposed PNN with a third order map, multilayer perceptron with sigmoid activation functions (MLP) and 3 hidden states, and long shot-term memory network (LSTM) with 10 inner cells.

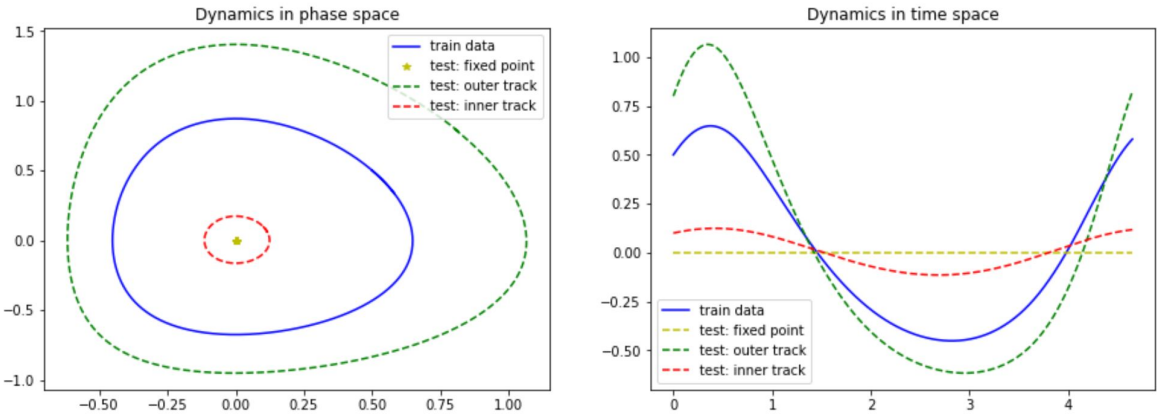

Figure 2.5: The solid blue line is the solution used as training set,the others represent fixed point,outer(green) and inner(red) track.

2.3.1. Long Short Term Memory

Let's see a brief introduction to this particular type of Neural Network (to study in deep this topic see [3]): Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of Recursive Neural Network(RNN), capable of learning long-term dependencies.

They were introduced by Hochreiter Schmidhuber (1997), and were refined and popularized by many people in following work, LSTMs are explicitly designed to avoid the long-term dependency problem.

Remembering information for long periods of time is practically their default behavior, not something they struggle to learn! All recurrent neural networks have the form of a chain of repeating modules of neural network.

In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer,as we can see in Figure 2.6 :

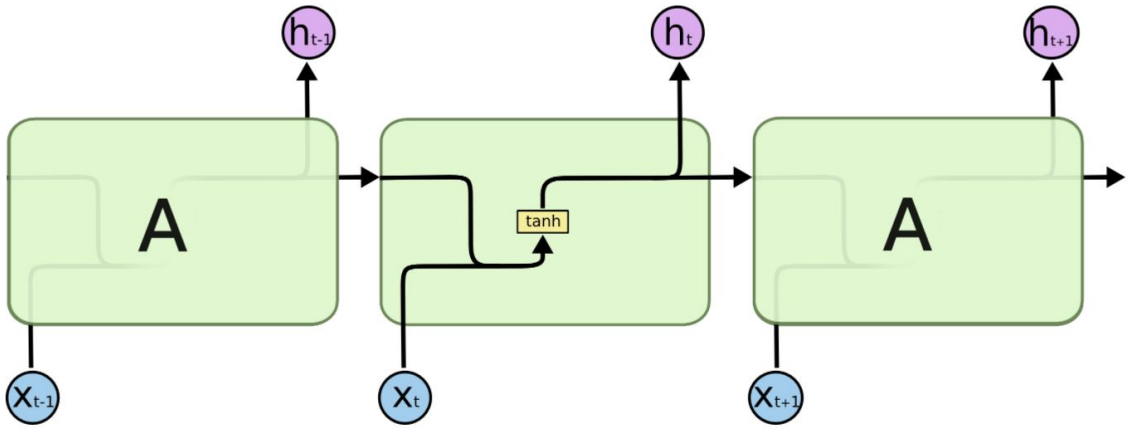

Figure 2.6: RNN structure (from [8])

LSTMs also have this chain like structure, but the repeating module has a different structure. Indeed,instead of having a single neural network layer, there are four, interacting in a very special way,as there appear in Figure 2.7:

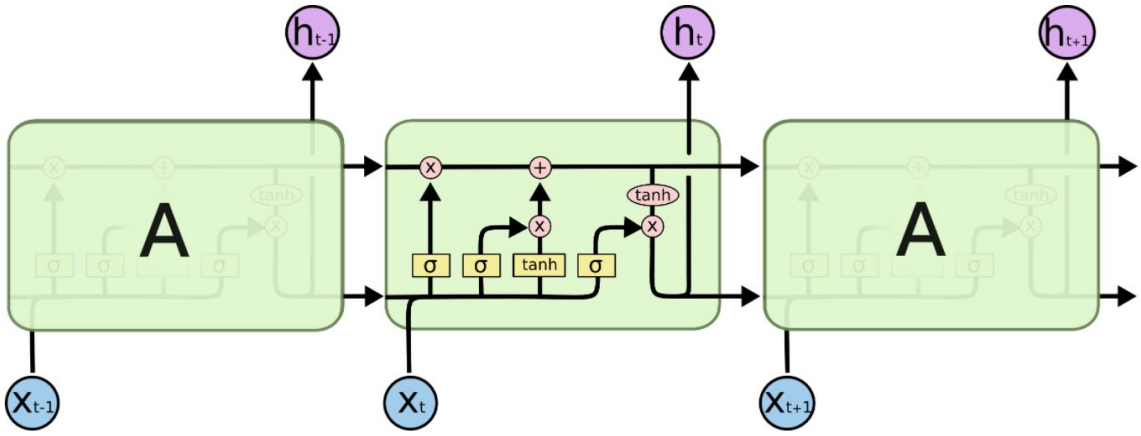

Figure 2.7: LSTM structure(from [8]).

2.3.2. Results

After the training phase with only one particular solution, both MLP and LSTM networks tend to learn the given solution without any kind of generalization (see Figure 2.8).

They are not able to predict something that has not been presented in training data.

At the same time, the proposed PNN predicts unknown dynamics in both nonlinear areas and near-linear oscillation around the stationary point.

It is clear that it is not perfect as prediction, but the main thing to notice is that for the PNN there is not a memorization phenomenon, but on the contrary it seems like it is capable of generalize the right working of the dynamical system.

Perhaps, it is possible to get the same level of generalization as for PNN by applying more intense training or slightly different settings of a state-of-the-art NN, but it is not clear how to achieve it ([7]).

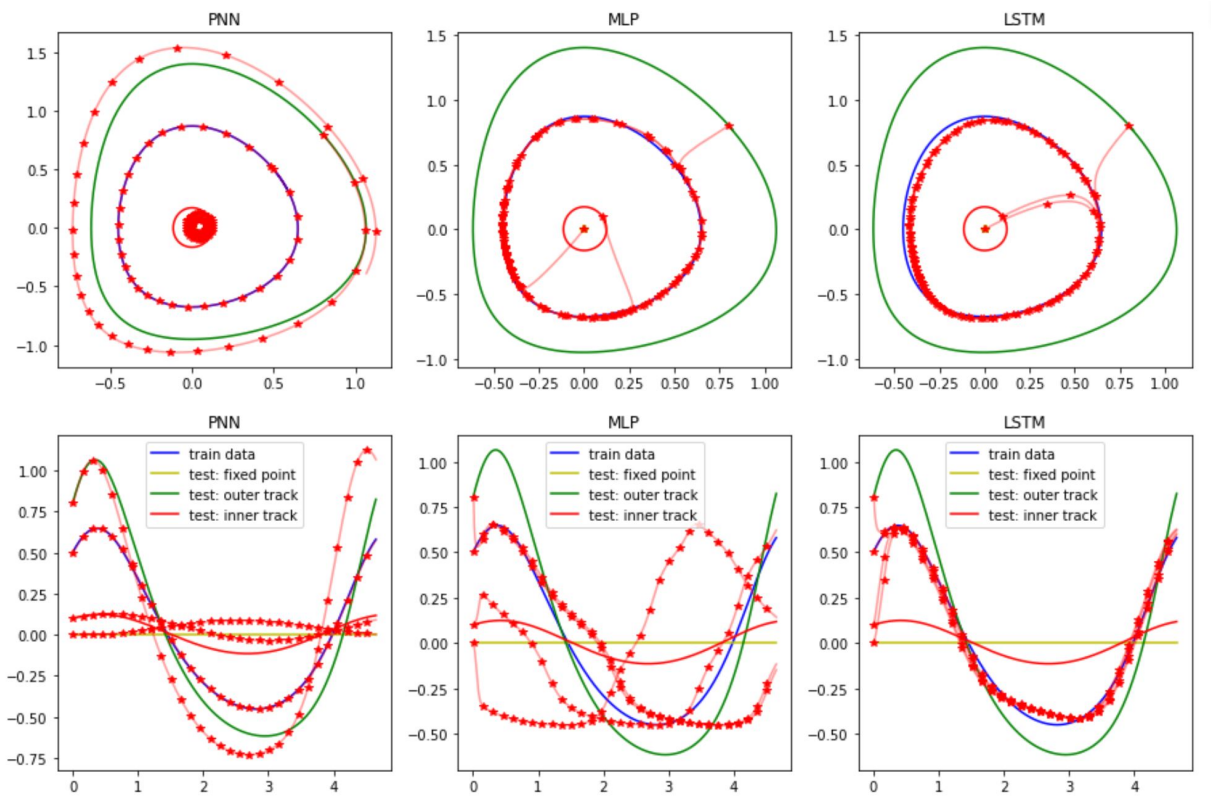

Figure 2.8: PNN - MLP - LSTM.

For example, focusing on LSTM architecture and try different parameters and training epochs, it is possible to obtain one of these two problems: a memorization phenomenon (like the one showed before, with 10 inner cells and 2000 epochs) or an underfitting phenomenon (using 10 inner cells for example and 100 or 1000 epochs), as it is possible to see in Figure 2.9.

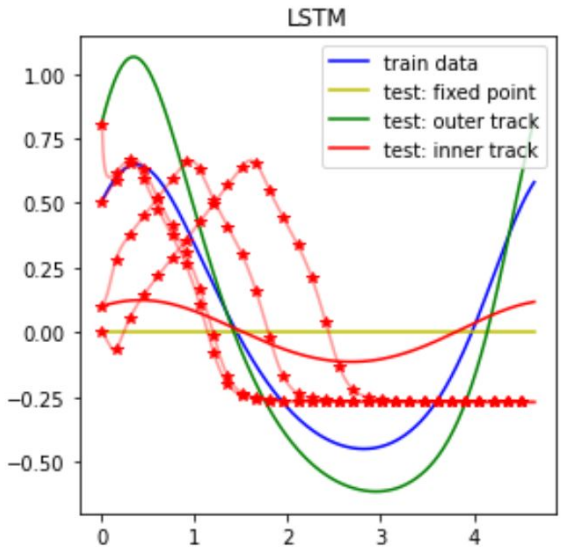

Figure 2.9: Undefitting phenomenon of LSTM.

In the next table (2.1) we can see an estimate of the Euclidean norm :

$$\|x\|_2 = \left[\sum_i abs\,(x_i)^2\right]^{1/2}$$

where the vector x is the difference between the test solution and the predicted one.

	outer track	**inner track**	**fixed point**
1000	6.501	1.186	1.143
2000	6.199	1.185	1.153
3000	5.980	1.177	1.149
4000	5.766	1.168	1.143
5000	5.550	1.156	1.134

Table 2.1: Euclidean norm of the difference between the real and the predicted values of inner track, outer track and fixed point for different values of training epochs.

Traditional neural networks have to be trained with lots of different solutions in order to perform well. It is still an open question if a state-of-the-art neural network can be trained with only one solution of the dynamical system and achieve generalization for other ones.

On the other hand, the Taylor map-based PNN is strongly associated with the theory of differential equations and is more suitable for dynamical systems learning. This section

demonstrates from both theoretical and practical points of view that the PNN is a more suitable architecture for dynamical systems learning rather than traditional ML models and neural networks.

In this section, virtual noise-free measurement are considered just to show the traditional ML and NN models limitations in the formulated problem of one-shot learning (i.e. the attempt to learn from only one solution used to train). The ODEs are used only for data generation, while weights of the PNN are estimated from the data without a priory knowledge of physical laws.

On the other hand, if the system dynamics follows approximately a system of ODEs, the PNN can be initialized from these ODEs using the Taylor mapping approach (TM-PNN) and additionally fine-tuned with the data.

2.4. Fine-tuning problem

Consider now a Fine-tuning problem: by taking weights of a trained neural network, use it as initialization for a new model being trained on data from the same domain.

Let's imagine a real physical problem related to the simple pendulum: in a virtual experiment some measures are taken, but an accidental error arises, in particular the length of the wire is measured as $L = 0.30$ cm, but the real ones is $L = 25$ cm.

The problem analyzed in this section is that the theoretical solution found by using a numerical solver is different from the real one, just because this accidental error was committed.

It is possible to recover the real information about the physical-based model using the PNN; let's generate as physical measurement 500 data (from time 0 to time $5s$, each measurement is taken every 0.01 s).

Now let's build a TM-PNN as the one in Figure 2.2, implemented as a multi-output model as we discuss previously. The Adam optimizer with a controlled gradient clipping during 1000 epochs is used for training. Consider as initial condition $\mathbf{x_0} = (0.09, 0)$.

The weights matrices are evaluated using the Taylor Map method, remembering that in the dynamical system we use $L = 0.30$ cmm, i.e. the theoretical value.

Now proceeds building a Polynomial Neural Network with the initialization of the weights using the ones just found; the next point consists in performing the training phase by using the statistical data collected (they were generated using Runge-Kutta).

In Figure 2.10 we can see the results of the predictions of the TM-PNN: the theoretical solution is the orange one, it is the one we expect as output. The cyan line is the real solution, the one that represents faithfully the physical dynamic.

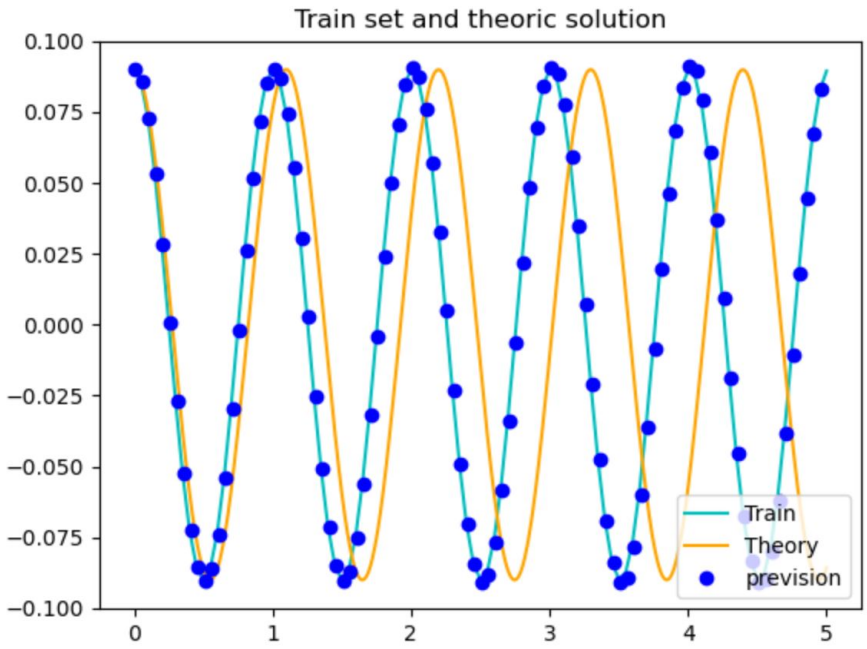

Figure 2.10: One-shot tuning of the TM-PNN for real pendulum with initial weights obtained from the theoretical ideal ODEs.

The blue dots are the values of the prevision of the TM-PNN: it is clear that initially they follow the orange line, but after less than 1 second the dots starts to follow the cyan line, it is a good representation of the real physical phenomenon.

The interesting thing is that now it is possible to use the same TM-PNN already implemented to find other solutions, referred to different initial conditions from the one used for the solution of the training phase.

As we can see in Figure 2.11, the model is capable of recover the dynamic of the system also for unseen inputs: the dotted line are the predictions and the solid ones are the real values (found using Runge-Kutta). The initial conditions considered are the following: cyan for $(0.27, 0)$, green for $(0.09, 0)$, used also for the training phase, and magenta for $(-0.17, 0)$.

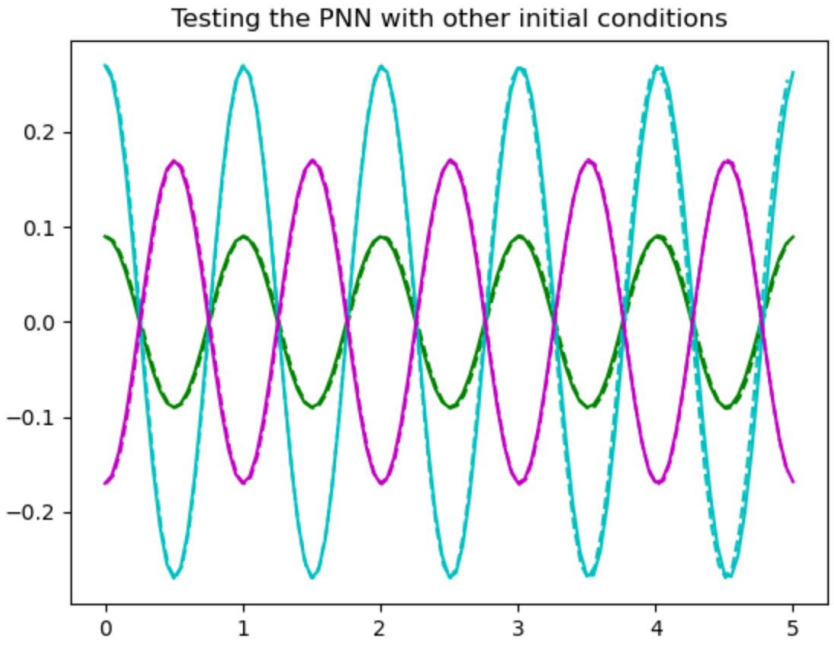

Figure 2.11: Predictions of the fine-tuned TM-PNN with unseen initial angles.

3 | Conclusions and future developments

This work proposes an approach to incorporate physics-based models into the TM-PNN architecture. This allows the preservation of a priori, physical knowledge in the NN and fine-tuning it with one sample.

In Chapter 1 the Taylor Map approach is analyzed both in theoretical and practical point of view (many test cases are considered). In Chapter 2 the structural essence of the PNN is presented: first there is the construction of such model, second the resolution of two problems; there are interesting considerations on how to recover information about a physical system having only his output (black-box model) and how to fine-tuning the PNN with only one sample.

Future developments

Since the connection between ODEs and the TM-PNN in terms of the one-shot learning of dynamical systems is recent, further research on the estimation of accuracy, performance, and limitations of the proposed method should be conducted.

Another interesting aspect to consider could be on how accurate the initial assumption of the system of ODEs and the complex TM-PNN architecture must be in terms of nonlinear orders or the number of hidden layers required for a given physical problem.

In Chapter 2, with the problem of recover information about the Lotka-Volterra system, the purpose is to show that the PNN is capable of generalize somehow, but the results could be improved working on the parameters of the NN.

A fields of studies could be the research of the unknown values of the parameters of a dynamical system: starting having only some output data and knowing the physical problem (i.e. the main equations), we could need to estimate some coefficients that are completely unknown. This could be used as a method of optimization.

Bibliography

[1] J. Brownlee. Time series forecasting with the long short-term memory network in python. https://machinelearningmastery.com/time-series-forecasting-long-short-term-memory-network-python/, August 28, 2020. [Online].

[2] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.

[4] A. Ivanov and S. Andrianov. Matrix lie maps and neural networks for solving differential equations. *arXiv preprint arXiv:1908.06088*, 2019.

[5] A. Ivanov, S. Andrianov, N. Kulabukhova, A. Sholokhova, E. Krushinevskii, and E. Sboeva. Matrix representation of lie transform in tensorflow. In *9th Int. Particle Accelerator Conf.(IPAC'18), Vancouver, BC, Canada, April 29-May 4, 2018*, pages 3438–3440. JACOW Publishing, Geneva, Switzerland, 2018.

[6] A. Ivanov, A. Golovkina, and U. Iben. Polynomial neural networks and taylor maps for dynamical systems simulation and learning. *arXiv preprint arXiv:1912.09986*, 2019.

[7] A. Ivanov, U. Iben, and A. Golovkina. Physics-based polynomial neural networks for one-shot learning of dynamical systems from one or a few samples. *CoRR*, abs/2005.11699, 2020. URL https://arxiv.org/abs/2005.11699.

[8] C. Olah. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, August 27, 2015. [Online].

[9] S. Raschka and V. Mirjalili. Python machine learning: Machine learning and deep learning with python. *Scikit-Learn, and TensorFlow. Second edition ed*, 2017.

[10] J. Willard, X. Jia, S. Xu, M. Steinbach, and V. Kumar. Integrating physics-based modeling with machine learning: A survey. *arXiv preprint arXiv:2003.04919*, 1(1): 1–34, 2020.

A | Appendix A

In appendix there are the main functions used in this thesis. The full code is available on github : `https://github.com/Danielsa95/Taylor-Map-PNN`.

```
"""
In this file there is the code that are used in all files to FIND THE
    WEIGHTS of the TaylorMap method.
In particular there are weights up to a third order ( W0,W1,W2,W3)
    that can be usefull to systems which have
a polynomial form up to a third order ( ax^3 + bx^2 + cx + d) or that
    can be approximated with a Taylor expansion.

Important: the system that are considered are 1D or 2D. (An higher
    order can be an interesting enlargement).
"""
import numpy as np
import helper_fun as tmp


# Input  ->   coef : list of P0,P1,P2,P3  which are the matrices of
    coefficients of the system ;
#        ->   Tf   : the right extrema of the interval, that is the
    point in which we want the value of the
#                    time-dependent functions of the differential
    system of the weights ;
#        ->   dim  : the dimension of the dynamical system ( 1 or 2 )
    ;
#        -> n_step : the number of step of forward Euler to solve the
    system of the weights.

# Output -> list of weights' matrices.

def weight(coef,Tf,dim,n_step=1000):
    dt=Tf/n_step

    if dim==2 :
        P0,P1,P2,P3=tmp.coeff(coef)
```

```
                                    ###### WEIGHTS INITIALIZATION ######

######## W0
        W0_0 = np.array([[0],[0]])
        W0   = np.array([[0],[0]])

######### W1
        W1_0 = np.eye(2)
        W1   = np.eye(2)

######## W2
        W2_0 = np.zeros(6).reshape(2,3)
        W2   = np.zeros(6).reshape(2,3)

######## W3
        W3_0 = np.zeros(8).reshape(2,4)
        W3   = np.zeros(8).reshape(2,4)

                                ####### EVALUATION OF WEIGHTS ######

        for i in range(n_step) :
            # W0
            W0=W0_0 + dt * ( P0 + np.matmul(P1,W0_0) + np.matmul(P2,
                tmp.kron(W0_0,2)).reshape(2,1) +
                                np.matmul(P3,tmp.kron(W0_0,3).reshape
                                    (4,1)))
            # W1
            W1 = W1_0 + dt * (np.matmul(P1, W1_0) + np.matmul(P2, 2*
                tmp.kron_0(W0,W1) )  +
                                 3*np.matmul(P3,tmp.kron_123(W0_0,W0_0,
                                    W1_0)))
            # W2
            W2 = W2_0 + dt * (np.matmul(P1, W2_0) + np.matmul(P2,  tmp
                .kron_1(W1_0, W1_0)) +
                                 np.matmul(P2,tmp.kron_0(W0_0,W2_0)) + 3*
                                    np.matmul(P3,tmp.kron_123(W0_0,W0_0,
                                    W2_0)) +
                                 3* np.matmul(P3, tmp.kron_123(W0_0,W1_0,
                                    W1_0) ))
            # W3
            W3 = W3_0 + dt * (np.matmul(P1,W3_0) + np.matmul(P2,2*tmp.
                kron_0(W0_0,W3_0)) +
```

```
                                np.matmul(P2, 2 * tmp.kron_1(W1, W2)) +
                                    np.matmul(P3,tmp.kron_1_3(W1_0)) +
                                np.matmul(P3, 3 * tmp.kron_123(W0_0,
                                    W0_0, W3_0)) + np.matmul(P3,6*tmp.
                                    kron_123(W0_0,W1_0,W2_0)))

            # Update
            W0_0=W0
            W1_0=W1
            W2_0=W2
            W3_0=W3


    if dim ==1 :

        p0,p1,p2,p3=tmp.coeff(coef)

    ######  WEIGHTS INITIALIZATION
        W0_0 = 0
        W1_0 = 1
        W2_0 = 0
        W3_0 = 0
        W0 = 0
        W1 = 0
        W2 = 0
        W3 = 0
        for i in range(n_step):
            W0=W0_0 + dt*(p0 + p1*W0_0 + p2*W0_0**2 + p3*W0_0**3)
            W1=W1_0 + dt*(p1*W1_0 + 2 * p2*W0_0*W1_0 + 3*p3*W1_0*W0_0
                **2 )
            W2=W2_0 + dt*(p1*W2_0 + p2*(W1_0**2 + 2*W0_0*W2_0) + 3*p3
                *(W2_0*W0_0**2 + W0_0 * W1_0**2) )
            W3=W3_0 + dt*(p1*W3_0 + 2*p2*(W0_0*W3_0+W1_0*W2_0) + p3*(
                W1_0**3 + 3*W3_0*W0_0**2  +6* W0_0*W1_0*W2_0) )
        # Update :
            W0_0=W0
            W1_0=W1
            W2_0=W2
            W3_0=W3
    return (W0,W1,W2,W3)
```

```
import numpy as np
import weights as mn
from math import ceil

""" In this file there are all the helper function used to evaluate
    the weights matrices and
     to perform the examples."""


# Simple function for the printing of the weights values
def show(W0,W1,W2,W3):
    print('######    W0   ######')
    print(W0)
    print('######    W1   ######')
    print(W1)
    print('######    W2   ######')
    print(W2)
    print('######    W3   ######')
    print(W3)
    return 0



# Input  -> vet : a vector of the form [a,b] ;
#        ->   n : the desidered power ;
# Output -> the n_th power of Kronecker with same term reduction.
def kron(vet,n) :
    res=np.zeros(n+1)
    for i in range(n+1):
        res[i]=vet[0]**(n-i) * vet[1]**i
    return res



# Input  ->  w0 : the 1st vector of weights ;
#        ->   W : a matrix of weight like W1, W2 or W3 ;
# Output ->  Kronecker product  with same term reduction.
def kron_0(w0,W) :
    temp = np.array([[w0[0][0], 0], [w0[1][0] / 2, w0[0][0] / 2], [0,
       w0[1][0]]])
    return np.matmul(temp,W)
```

```
# Input  ->  W1 : the W1 matrix of weights ;
#        ->   W : a matrix of weight like W1, W2 or W3 ;
# Output ->  Kronecker product  with same term reduction.
def kron_1(W1,W) :
  M3=np.kron(W1,W)
  M3[1, :] = (M3[1, :] + M3[2, :]) / 2
  M3 = np.delete(M3, (2), axis=0)
  for i in range(W.shape[1]-1) :
   M3[:,1+i]=M3[:,1+i]+M3[:,i+W.shape[1]]
  for i in range(W.shape[1]-1):
   M3=np.delete(M3,(W.shape[1]),axis=1)
  return M3



# Input  ->  W0 : the W0 vector of weights ;
#        ->  W1 : the W1 matrix of weights;
#        ->  W2 : the W2 matrix of weights;
# Output ->  Kronecker product with same term reduction.
def kron_123(w0,w1,w2):
    if w1.shape[1]==2:
        M=np.kron(w0,kron_1(w1,w2))
    if w1.shape[1]==1:
        M=np.kron(w0,kron_0(w1,w2))
    M[1,:] = (2*M[1,:] + M[3,:])/3
    M[2,:] = (M[2,:] + 2 * M[4,:])/3
    M = np.delete(M, (3,4), axis=0)
    return M



# Input  ->  W1 : the W1 matrix of weights;
# Output -> the 3rd power of Kronecker with same term reduction of W1
   .
def kron_1_3(W1):
    M=np.kron(W1,kron_1(W1,W1))
    M = np.delete(M, (3, 4), axis=0)
    M[:,1]=M[:,1] + M[:,3]
    M[:, 2] = M[:, 2] + M[:, 4]
    M = np.delete(M, (3, 4), axis=1)
    return M
```

```
#This function takes the list of coefficients' matrices and return
    them.
def coeff(lista):
    P0=lista[0]
    P1=lista[1]
    P2=lista[2]
    P3=lista[3]
    return P0,P1,P2,P3

#
    ##############################################################################

#
    ##############################################################################


##    FROM NOW THERE ARE ONLY SUPPORT FUNCTIONS USED IN FILE DIFFERENT
     FROM WEIGHTS  !!  ##

#
    ##############################################################################

#
    ##############################################################################


# Function to solve a 1D problem.
# Input  ->  pesi       : list of the weights;
#        ->  val_init   : the initial value x0 ;
#        ->  n_step     : # of step of the TM iterative method.
# Output ->  list of solutions, that are values of the function in
    every dt .
def solve(pesi,val_init,n_step):
 x=val_init
 sol=np.array([[]])
 sol=np.append(sol,x)
 for i in range(int(n_step)) :
    x=pesi[0] + pesi[1]*x + pesi[2]*x**2 + pesi[3]*x**3
    sol=np.append(sol,x)
 return sol



# Function to solve a 2D problem.
```

```
# Input  ->  pesi       : list of the weights;
#        ->  val_init   : the initial value X0=(x0,y0) ;
#        ->  n_step     : # of step of the TM iterative method.
# Output ->  list of solutions, that are values of the function in
    every dt .
def solve2(pesi,val_init,n_step):
    n_step=ceil(n_step)
    x = val_init
    sol = list()
    sol.append(x)
    for i in range(int(n_step)):
        x = pesi[0] + np.matmul(pesi[1],x) + np.matmul(pesi[2],kron(x
            ,2).reshape(3,1)) +\
             np.matmul(pesi[3],kron(x,3).reshape(4,1))
        sol.append(x)
    return sol


# Input  -> lista1 : a list of the matrices evaluated with the TM
    method implemented.
#        -> lista2 : a list of the matrices given in the paper.
# Output -> vet_norme : a vector in which in every position there is
    an estimate (the norm-2) of the distance
#                       between the two type of weights : the ones
    found with TM method and those of paper.
def norma(lista1,lista2):
    vet_norme=list()
    for i in range(int(len(lista1))):
        tmp=np.linalg.norm(lista1[i]-lista2[i])
        vet_norme.append(tmp)
    return vet_norme



# A function that given a vector "vet" and a number "x" return the
    index of the vector in which there is the
# nearest value wrt x .
def find_near(vet,x):
    k=np.abs(vet[0]-x)
    idx=0
    for i in range(len(vet)) :
        diff=np.abs(vet[i]-x)
        if(k>diff):
            k=diff
```

```
            idx=i
    return idx



# To make a comparison between the solution that i obtain using the TM
    method and the rk45 method there
# is a preliminar thing to do : the first solution is discretize in a
    constant time step dt,differently
# the second is referred to some value of time.
# Using this function taking in inputs two vector (like the two
    vectors of the times) returns a vector
# that contain indices : every elements of the shorter vector is
    searched in the bigger one, if it is found
# then the indexes in where it is is signed, if not then the function
    find_near is called.
# Remember that both the two vectors contain value of time between 0
    and Tf .
def find_index(x,y):
    if len(x) < len(y):
        tmp = x
        x = y
        y = tmp
    count = 0
    idx = list()
    find = False
    for i in range(len(y)):
        while count + i < len(x) - 1 and find == False:
            if (y[i] == x[i + count]):
                idx.append(i + count)
                find = True
            count += 1
        if count + i >= len(x) - 1:
            count = idx[-1]
            k = find_near(x, y[i])
            idx.append(k)
            count = k
        find = False

    return idx

## This function is used for Gravity_fall, it is used to evaluate MSE
    for dt = 0.1, 0.2 , ... , 1.0 .
# Input  -> coef : the list of the coefficients of the equation ;
```

```
#         -> m    : the mass ;
#         -> Tf   : the final time ;
#         -> init : the initial point.
# Output -> mse_1,mse_2  : they are two vectors that contains the
    evaluation of the MSE.

def MSE(coef,m,Tf,init):
    dt_init=0.1
    mse_1=np.zeros(10)
    mse_2=np.zeros(10)
    p0,p1,p2,p3=coef
    ### real sol
    sol = lambda t: np.sqrt(m * 9.81 / 0.392) * np.tanh(t * np.sqrt
        (0.392 * 9.81 / m))
    lis = (p0, p1, p2, p3)

    for i in np.arange(1, 11, 1):
        dt = dt_init * i
        W0_mse, W1_mse, W2_mse, W3_mse = mn.weight(lis, dt, 1)
        sol_real=sol(np.arange(dt,Tf+dt,dt))

        ### solution 1)
        sol_1 = solve((W0_mse, W1_mse, W2_mse, W3_mse), init, ceil(Tf
            / dt))
        sol_1 = sol_1[1:]
        mse_1[i-1]=np.mean((sol_1-sol_real)**2)


        ### solution 2)
        sol_2 = np.zeros(ceil(Tf / dt))
        x = init
        for j in np.arange(dt, Tf + dt, dt):
            x +=  9.81 * dt - (dt * 0.392 * x ** 2) / m
            sol_2[round(j/dt)-1]=x
        mse_2[i-1] = np.mean((sol_2 - sol_real) ** 2)

    return mse_1 , mse_2
```

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.layers import Layer
from tensorflow.keras.layers import Activation,Dense,LSTM
from tensorflow.keras.models import Sequential
import weights as mn

""" In this file there is the code of the layers and the
    implementation of the Neural Networks used."""


"""
This layer (a subclasses of the layers.Layer main class of Tensorflow.
    keras) has the easy task to take
input values (x,y) at time t_i and return a vector which contains all
    the component of the Kronecker power with same term
reduction,

(x,y) ---------- > (x, y, x^2, xy, y^2, x^3, x^2 * y, x*y^2, y^3).

The other layer (the one named custom) takes as input the output of
    the 'filter' layer and returns
two values : (x,y) at time t_i+1.

"""
class filter(Layer) :
    def __init__(self,num_units,activation="relu"):
        super(filter,self).__init__()
        self.num_units=num_units
        self.activation=Activation(activation)

    def build(self,input_shape):
        self.weight=self.add_weight(shape=[input_shape[-1],self.
            num_units],trainable=False)
        b_init = tf.zeros_initializer()
        self.bias = tf.Variable(
             initial_value=b_init(shape=(self.num_units,), dtype='
                float32'),
            trainable=False)

    def call(self,input):
        t0 = np.array
            ([[[1.,0.,1.,1.,0.,1.,1.,1.,0.],[0.,1.,0.,0.,1.,0.,0.,0.,1.]]],
            dtype='float32')
```

```
        t1 = np.array
            ([[[0.,0.,1.,0.,0.,1.,1.,0.,0.],[0.,0.,0.,1.,1.,0.,0.,1.,1.]]],
            dtype='float32')
        t2 = np.array
            ([[[0.,0.,0.,0.,0.,1.,0.,0.,0.],[0.,0.,0.,0.,0.,0.,1.,1.,1.]]],
            dtype='float32')
        t0 = tf.constant(t0)
        t1 = tf.constant(t1)
        t2 = tf.constant(t2)
        tmp0 = tf.matmul(input,t0)
        tmp1 = tf.matmul(input,t1) + tf.constant
            ([[1.,1.,0.,0.,0.,0.,0.,0.,0.]])
        tmp2 = tf.matmul(input,t2) + tf.constant
            ([[1.,1.,1.,1.,1.,0.,0.,0.,0.]])

        y=tmp0 * tmp1 * tmp2
        return y
#
    ##########################################################################################



#
    ##########################################################################################

class custom(Layer) :
    def __init__(self,num_units):
        super(custom,self).__init__()
        self.num_units=num_units

    def build(self,input_shape):
        self.weight = self.add_weight(shape=[input_shape[-1], self.
            num_units], trainable=True)
        b_init = tf.zeros_initializer()
        self.bias = tf.Variable(
            initial_value=b_init(shape=(self.num_units,), dtype='
                float32'),
            trainable=True)


    def call(self,input):
        y=tf.matmul(input,self.weight) + self.bias
        return y
#
```

```
    ##############################################################################


##### Using this function is possible to create the PNN
def createPNN(coeff=0,dt=0,n_dim=0,initialization=False):
    if(initialization == True):
      w0,w1,w2,w3=mn.weight(coeff,dt,n_dim)        ###  evaluation of
          the weight
      W_init = np.array([[w1[0][0], w1[1][0]],
                         [w1[0][1], w1[1][1]],
                         [w2[0][0], w2[1][0]],
                         [w2[0][1], w2[1][1]],
                         [w2[0][2], w2[1][2]],
                         [w3[0][0], w3[1][0]],
                         [w3[0][1], w3[1][1]],
                         [w3[0][2], w3[1][2]],
                         [w3[0][3], w3[1][3]]
                        ], dtype='float32'), \
               np.array([0,0], dtype='float32')
    else :
       W_init = np.array([[1, 0],
                          [0, 1],
                          [0, 0],
                          [0, 0],
                          [0, 0],
                          [0, 0],
                          [0, 0],
                          [0, 0],
                          [0, 0]
                         ], dtype='float32'), \
               np.array([0,0], dtype='float32')


    l1=filter(9)
    l2=custom(2)
    model=Sequential()
    model.add(l1)
    model.add(l2)
    model.predict([[0., 0.]])
    opt = tf.keras.optimizers.Adam(lr=0.02, beta_1=0.99,
                                   beta_2=0.99999, epsilon=1e-1, decay
                                      =0.0)
    model.compile(loss='mean_squared_error', optimizer=opt)
    model.layers[1].set_weights(W_init)       #### setting the right
```

```
        weight, the ones found using the TM method
    return model
#
    ##############################################################################


def pred(nstep,model,x0):
    xx=model
    x=np.zeros(nstep+1)
    y=np.zeros(nstep+1)
    x[0]=x0[0][0]
    y[0]=x0[0][1]
    tmp=x0
    for i in range(nstep):
        vet=xx.predict(tmp)
        x[i+1]=vet[0][0][0]
        y[i+1]=vet[0][0][1]
        tmp=vet

    return x,y

#
    ##############################################################################

def createMLP(inputDim, outputDim):
    model = Sequential()
    model.add(Dense(4, input_dim=inputDim, activation='sigmoid'))
    model.add(Dense(4, activation='sigmoid'))
    model.add(Dense(outputDim, activation='linear'))
    model.compile(loss='mean_squared_error', optimizer='adamax')
    return model


def createLSTM(inputDim, outputDim):
    model = Sequential()
    model.add(LSTM(10, input_dim=inputDim, input_length=1))
    model.add(Dense(outputDim, activation='linear'))
    model.compile(loss='mean_squared_error', optimizer='rmsprop')
    return model

def iterative_predict(model, X0, N, reshape = False):
    ans = np.empty((N, 2))
    X = X0.reshape(-1,2)
    for i in range(N):
```

```
        if reshape:
            X = model.predict(X.reshape(1,1,2))
        else:
            X = model.predict(X)
        ans[i] = X
    return np.vstack((X0, ans[:-1]))
```

List of Figures

1.1 Iterative method based on TM. 6
1.2 Scheme of the method. 8
1.3 Left: TM solution; Right: Euler solution. 11
1.4 MSE between the analytical solution. 11
1.5 Solution_1. 15
1.6 Solution_2 15
1.7 Plot of the angle. 15
1.8 Plot of the velocity. 15
1.9 Solution_1 19
1.10 Solution_2 19
1.11 Phase of sol_1 19
1.12 Phase of sol_2 19
1.13 Comparison between the three solutions 19
1.14 Comparison between the three phases 19
1.15 Comparison between the solutions. 22

2.1 Polynomial Neural Network-neuron(from [7]). 25
2.2 Polynomial Neural Network (adapted [7]) 26
2.3 Structure of the $\mathcal{M}$ box 26
2.4 Left:solution with TM method; Right:RK solution 28
2.5 The solid blue line is the solution used as training set,the others represent fixed point,outer(green) and inner(red) track. 29
2.6 RNN structure (from [8]) 30
2.7 LSTM structure(from [8]). 30
2.8 PNN - MLP - LSTM. 31
2.9 Undefitting phenomenon of LSTM. 32
2.10 One-shot tuning of the TM-PNN for real pendulum with initial weights obtained from the theoretical ideal ODEs. 34
2.11 Predictions of the fine-tuned TM-PNN with unseen initial angles. 35

List of Tables

2.1 Euclidean norm of the difference between the real and the predicted values of inner track, outer track and fixed point for different values of training epochs. 32

Acknowledgements

Here you might want to acknowledge someone.