Recurrent Neural Networks for dynamical systems identification

M.Facci, D.Zurlo

Dipartimento di Ingegneria Informatica Automatica e Gestionale
Antonio Ruberti
Control Engineering
Sapienza Università di Roma

January 15, 2022

Contents

1 Introduction

System identification is a field of system theory in which the aim is to reconstruct the evolution of the state, in case of state estimation, or reconstruct some constant parameters of the system, in case of parameters identification, of a dynamical system. A dynamical system is characterized by its state variables that give a complete description of the system at any instant of time. Starting from a physical model it is possible to describe it, with some approximation, as a set of linear or nonlinear differential equations depending on how well we want to describe the real system. An important aspect to take into account is that, even if we are able to obtain an accurate model of the system, we cannot take into account the uncertainties acting on it due to the statistical nature of these uncertainties. The reason why this field is so studied is that having an accurate behavior of the system under analysis allows us to tackle some important problems, such as prediction of some phenomena, control the system through state feedback, find critical parameters, etc. An important step forward in this field was done at the beginning of 60', thanks to the contribution of R. E. Kalman, that provided a recursive method, called Kalman filter, of estimating the state of a dynamical system in presence of noise, and in case of Gaussian noise, it is also the estimator that provides the best solution. Instead, for a nonlinear system, an extension is necessary to estimate the state or some constant parameters. After that, many other variants have been developed such as Particles Filter, Unscendent Kalman Filter, etc. but what they have in common is that all of them require a mathematical model, which in some cases could be difficult to obtain. For this reason, a new kind of system identification has been introduced by using Artificial Intelligence that does not need to have a dynamic model, but the price to pay is that it is necessary a huge amount of data collected directly from the system and also a huge computational power is needed in order to train the system. Nowadays, the latter is not a problem thanks to the powerful GPUs and parallel computing, while for what concerns the data, it depends on the specific problems.
To summarize, system identification is aimed at constructing or selecting mathematical models of dynamic systems to serve certain purposes (prediction, diagnosis, control, etc.), using measurements of the input and output signals of the system to estimate the values of adjustable parameters in a given model structure.
Learning dynamical systems from data is a complex task that can be described as a parameter estimation problem. This requires the construction of a prior model and then a fitting phase of its parameters to the available data. Therefore a bad initial model means that the learned model does not perfectly capture the dynamics of the system. So, specific knowledge and insights into the available data are necessary to produce an appropriate model.
In real life, continuous processes are observed as discrete processes and most of the observations at the time steps t_i can be missed. Modeling this kind of system with a neural network, or a classical sequence modeling procedure is possible for example using architectures like Recurrent Neural Network somehow, which is not designed for this purpose.
Anyway, to reduce the need for domain knowledge and to automate modeling of dynamical systems from data, model-free methods that only rely on the available data have been developed. One approach that has started to be explored in recent years is the utilization of artificial neural networks (ANNs) and deep learning for data-driven modeling of dynamical systems due to the higher computational power and the large amounts of data that can be processed and to be trained on.

2 Neural Networks for dynamical systems identification - Zurlo Dario

2.1 State of the art

This report is focused on how it is possible to implement a state estimation for a nonlinear system using neural networks NNs. The first important step toward this new technique is done by Narendra and Parthasarathy in [1], where they prove the possibility to use NNs to solve system identification problems. Nechyba and Yangsheng in [2] proposed an efficient and flexible NN architecture which is capable of modelling nonlinear dynamical system using a cascade of two learning architecture which adjust the structure of the network. Wang and Chen in [3] proposed a fully automated NN that differ form a classical recurrent neural network RNN for the training phase that consists in two phases; the first one used to identify a minimal network size and the second one identifies the optimal trainable parameters. Yadaiah and Sowmya in [4] proposed an architecture composed by a cascade of two different NNs. The first one is an RNN fed by the same inputs of the plant and the output of this NN is the estimate state that feed the second NN that is a simple feed forward neural networks FFNN, which give in output the estimated output. Ogunmoul, Olalekan et al. in [5] investigate the effectiveness of deep neural networks DNN in modeling dynamical systems with complex behaviors, in particular they investigate the use of multilayer networks, simple RNN and long-short time memory LSTM.
All of these technique mentioned so far have in common the use of RNN because this kind of networks is capable to deal with time series. Nowadays the most used RNN architecture is the LSTM, used successfully in many tasks such as speech recognition, translation, sentiment analysis etc because it is able to capture both long and short time dependencies. To understand how they are capable to perform better then simple RNN I have dedicated a section to explain the internal structure of one node of an LSTM and one section to give an idea on how the learning algorithm works. Then in section 2.4 I have formalized the system identification problem with NN presented in [4].
Another interesting architecture of NN used to learn dynamical systems are the Residual Network ResNet. The basic structure is like a FFNN with one input and one output but it has also a bypass on each neuron. An evolution o ResNet are the NeuralODE introduced in [6] to learn complex dynamics. The main advantage with respect the ResNet is that the NeuralODE are more memory efficient, for this reason they are becoming very popular, not only to learn complex dynamics but also for other purpose. In section 2.5 I give an idea of how the NeuralODE works.

2.2 RNN and LSTM

In a FFNN the input size is fixed and then they are not able to work with input with variable length as in case of time series. Actually the problem of working with time series is not the length of the input rather it is the fact that there is a correlation between input, and a classical NN is not able to capture it. Instead a RNN, in its basic formulation, is able to capture the dependencies between consecutive input. In figure 1 is shown the structure of a generic RNN where it is possible to see that differently from a FFNN it has a connection between the output and the input, this introduce the key concept of "parameters sharing". In figure 2 it is shown the same RNN but unrolled, where it is easy to see that they are composed of a chain of units, this representation is useful to understand how the weights update works.
The units have an internal structure able to modify the input, according with some weights and functions, that allow the network to learn. Many units has been developed to deal with different problem. The simplest one is shown in figure 3 that works well if the inputs have only

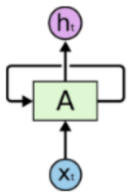


Figure 1: Compact RNN

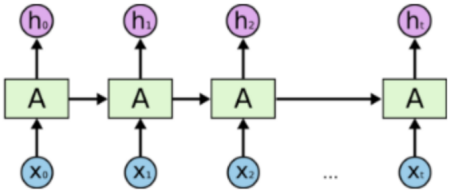


Figure 2: Unrolled RNN

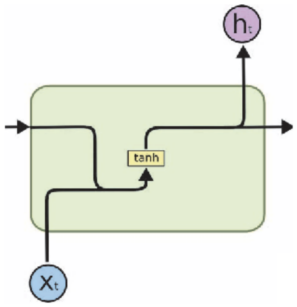


Figure 3: Simple RNN internal structure of one node

very short self dependencies, this was also the first kind of unit of a RNN developed, no longer used today due to its limitation. In place of this kind of unit researchers have been developed another kind of unit able to capture both long and short dependencies among input, the unit is called Long-Short Time Memory LSTM due to its capacity just mentioned. in figure 4 it is shown the internal structure of a LSTM units.
The key concept of an LSTM unit is that it has an internal state, carefully regulated by the gates, that let the information opportunely manipulated to pass through it. There are three different kind of gates.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{1}$$

The first one is the forget gate (1) that decide what information is going to be thrown away from the cell state C_{t-1} to forget irrelevant history. It is composed by a sigmoid function that gives in output a value between 0 and 1 and the input of the function are the weights W_f, the input of the current unit x_t, the output of the previous unit h_{t-1} and the bias b_f.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2}$$

Then the input gate (2) decide what part of the new information x_t is relevant, and use this to store this information into the cell state C_t. This gate is composed by a sigmoid function

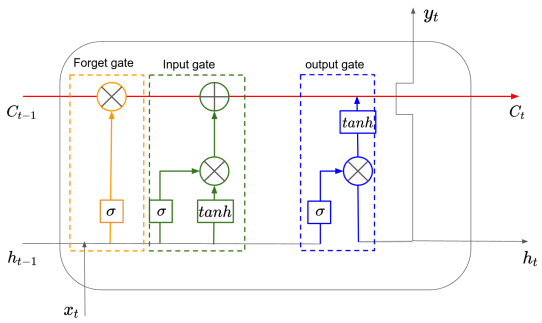


Figure 4: LSTM internal structure of one node

that decides which values it will update. The structure is similar to the previous one, but with different weight and baias, respectively W_i and b_i

$$\tilde{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{3}$$

The other component is the hyperbolic tangent (3) that generate a vector of the new candidate state, it generate an output between -1 and 1. The input to this function has the same structure as the previous one with the weights W_c and the bias b_c.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{4}$$

Finally the output gate (4) control what information encoded in the cell state is sent to the network as input in the next step h_t. This gate is composed by a sigmoid function which decide what part of the cell state send to the output. The input of the function are the output weights W_o multiplied by the previous output and the current input, summed to the output bias b_o. The other parts is the tanh function that weight the output with the internal state. At the end the output of the unit is give by the following equation

$$h_t = o_t * tanh(C_t) \tag{5}$$

Like any NN, it needs to a training phase before use it to adjust the internal weights, but differently from a FFNN it is is not possible to use the back propagation to propagate the error from the output to the input, but it is necessary a modified version, that is the Back Propagation Through Time BPTT. First of all it is necessary to compute the loss function, give by

$$L(x1, ..., x_T, y1, ..., y_T, w_h, w_o) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t) \tag{6}$$

that as it is possible to see is the sum of all the losses of each output.

2.3 Back Propagation Through Time

The training phase of a NN allow it to learn the approximate relation between the input and the output by adjusting the parameter with the following law

$$W_{new} = W_{old} - \alpha \frac{\partial L}{\partial W} \tag{7}$$

where W is the parameter to adjust and α is the step size, through this value we can impose how much we want to update the parameter in the direction in which the gradient decreases. Back propagation, in general, is used to compute $\frac{\partial L}{\partial W}$ after computing the loss function at the end of the forward phase. Since in the RNN the loss depends on the previous state and output it is necessary to use the Back Propagation Through Time BPTT. To compute the BPTT we need to calculate the gradient of the error function with respect to the parameters, that are W_f, W_i, W_c and W_o. In order to calculate the gradients with respect to these parameters, it is necessary to compute the errors of the neurons. In figure 5 it is possible to see the flow of the error propagated backward, from the loss function to the input.
An important problem of RNN is the vanishing gradient because it does not allow to the NN to learn anymore. The vanishing gradient occurs when we compute the gradient of the loss, in the back propagation phase, and at a certain point this gradient become either to large, in this case tends to explode, or to small and in this case tends to vanish. The reason why this occurs is because when we compute the gradient of the loss function with respect to the parameters

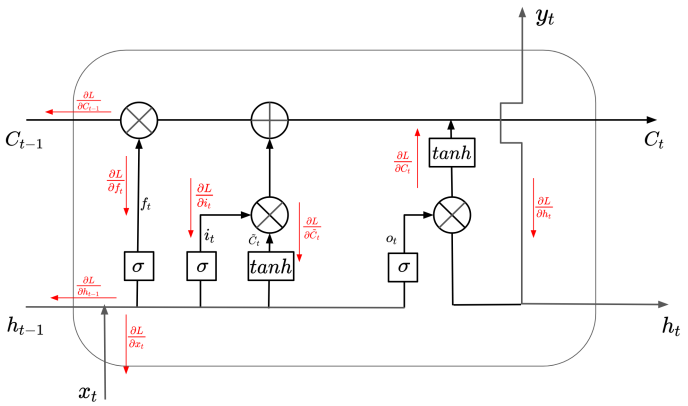


Figure 5: Flow of the derivative in one node

we use the chain rule, this causes that the gradient calculated deep in the network is "diluted", due to the products of the gradients at each step in time. In fact at each time time we compute

$$\begin{aligned}\frac{\partial E_t}{\partial W} &= \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}\cdots\frac{\partial c_2}{\partial c_1}\frac{\partial c_1}{\partial W} \\ &= \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}(\prod_{i=2}^{t}\frac{\partial c_i}{\partial c_{i-1}})\frac{\partial c_1}{\partial W}\end{aligned} \tag{8}$$

in a general RNN we can write that

$$c_t = \sigma(Wc_{t-1} + Ux_t) \tag{9}$$

where W is the weight associated to the previous cell and U is the weight associated to the current input. The derivative of c_t with respect to the previous state is given by

$$\frac{\partial c_t}{\partial c_{t-1}} = \sigma'(Wc_{t-1} + Ux_t)W \tag{10}$$

that substitute in (8) we get

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}(\prod_{i=2}^{t}\sigma'(Wc_{i-1} + Ux_i)W)\frac{\partial c_1}{\partial W} \tag{11}$$

this last expression tends to vanish when t is large, because the derivative of the sigmoid or the *tanh* is smaller then one. Moreover it can also explode if the weight W is larger enough to overpower the smaller derivative. The LSTM solve this problem thanks to its internal structure, composed by different gate. In fact the state vector is given by

$$c_t = c_{t-1} \otimes \sigma(W_f[h_{t-1}, x_t]) + tanh(W_c[h_{t-1}, x_t]) \otimes \sigma(W_i[h_{t-1}, x_t]) \tag{12}$$

where $\otimes$ is the Hadamar product, that consist in the element-wise product between elements of the two vectors. Compactly (12) can be written as

$$c_t = c_{t-1} \otimes f_t + \tilde{c}_t \otimes i_t \tag{13}$$

The derivative of c_t with respect to the previous state is composed by four contribution

$$\begin{aligned}\frac{\partial c_t}{\partial c_{t-1}} = \frac{\partial f_t}{\partial c_{t-1}}c_{t-1} + \frac{\partial c_{t-1}}{\partial c_{t-1}}f_t + \frac{\partial i_t}{\partial c_{t-1}}\tilde{c}_t + \frac{\partial \tilde{c}_t}{\partial c_{t-1}}i_t \\ = A_t + f_t + B_t + C_t\end{aligned} \tag{14}$$

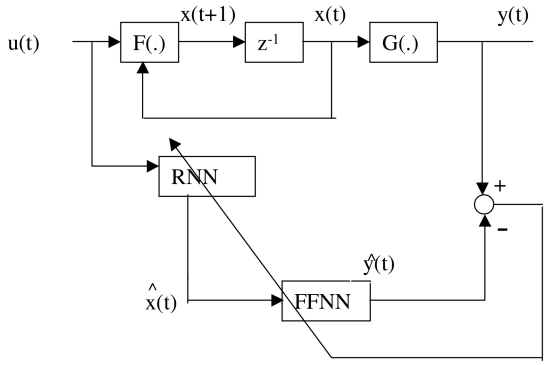


Figure 6: Block scheme of the system and the estimator

and substituting in the expression (8) we get

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}(\prod_{i=2}^{t}(A_i + f_i + B_i + C_i))\frac{\partial c_1}{\partial W} \tag{15}$$

Inside the product it is present the output of the forget gate that allow to control the value of the gradient at each time step by adjusting the parameters associated to this gate. Other solutions can be adopted to deal with the vanish gradient, for example using the Truncated BPTT, initialize properly the weights etc.

2.4 Problem Formulation

Among all the techniques mentioned at the beginning the most intuitive is the one presented in [4] where the authors used two different NNs placed in series. The aim of this section is to give the basic idea behind this paper. The authors considered a general class of nonlinear systems as

$$\dot{x}(k+1) = f(x(k), u(k), w(k)) \tag{16}$$
$$y(k) = g(x(k), v(k)) \tag{17}$$

where $x(k) \in \mathbb{R}^{n\times n}$ is the system state, $u(k) \in \mathbb{R}^{n\times p}$ is the input, $y(k) \in \mathbb{R}^{n\times q}$ is the output, $w(k)$ is the process noise and $v(k)$ is the measurements noise. The goal is to estimate the state of the system using the input $u(k)$ and the measured output $y(k)$. In particular the input $u(k)$ is used to feed the RNN that tries to mimic the system state dynamics and the output of this NN is the estimated state $\hat{x}(k)$, used to feed the FFNN that tries to mimic the measurements equations. The output of the FFNN is the estimated output $\hat{y}(k)$, that together to the measured output generates the error necessary to train the NNs. In figure 6 it is shown the block scheme of the system and the estimator.
The first step is to build the structure of the NN in which it is important to select a suitable number of node in each layers and the number of hidden layers, because they affect the final performance. In fact a NN with many neurons and many layers can approximate very well nonlinear functions but in the other hand it is possible to incur in the over fitting problem and then it is better to reduce the numbers of neurons and layers at the design level or using the dropout technique, that at each iteration eliminate a certain amount of connection. The estimator needs to be trained before use it, then once the structure of the network is defined the next step is to collect the data necessary to perform the training. In particular these data form the dataset that contains the pair $(u(k), y(k))$ at every instant of time. The values of the

output in the dataset can be retrieved by performing some tests on the system and measuring the output with the available sensors.
In the paper on which this section is based on the authors test the estimator on two dynamical system. The first one is a MIMO nonlinear system with three state, two inputs and two outputs. They tested different configurations of NN and the one that they chose, since it perform better than the others, is composed by 7 neurons in the input layer then 7 neurons in the first hidden layer and 3 neurons in the output layer of the RNN then 7 neurons for the input layer of the FFNN and 2 neurons in the output layer, they call this configuration 7-7-3-7-2. In order to compare the result with standard technique they also implemented an Extended Kalman Filter EKF. In figures 7, 8 and 9 it is shown the evolution of the real state and the estimated one where it is possible to see that the estimator base on NN perform better then the EKF. The second application is a linearized Continuous Stirred Tank Reactor, that has three state, three inputs and three outputs, in this case the NN configuration is 9-5-3-3-3 using the same reasoning as before for the notation. Also in this case, as it is possible to see form the figures 10, 11 and 12 the estimator base on NN perform better than the Kalman Filter.

2.5 ResNet and NeuralODE

The ResNet is composed by layers stacked one on top of the other and in addition it has a skipping connection, as it is possible to see from figure 13.
The output after the first layer is given by the sum of the output of the layer $f(z_t, \theta_{t+1})$ and the skip connection z_{t-1}. The generic state transformation in a ResNet ca be formalized as

$$z_{t+1} = z_t + f(z_t, \theta_{t+1}) \tag{18}$$

The importance of the skip connection is that prevent to discard important information by f due to the back propagation, in this way the information bypass the layer. The depth of this network is given by t that are the number of transformation. There is an evident correlation between ResNet and ODE, in fact if we look at all the transformation between the input of the network and the output, given by

$$\begin{aligned} z_1 &= z_0 + f(z_0, \theta_1) \\ z_2 &= z_1 + f(z_1, \theta_2) \\ &\cdot \\ &\cdot \\ &\cdot \\ z_{t+1} &= z_t + f(z_t, \theta_{t+1}) \end{aligned} \tag{19}$$

it relies upon the same recursive relationship as the method of Euler to solve differential equation numerically. One of the biggest downside of ResNet is that they are computationally expensive to train due to its depth.
The evolution of ResNet are the NeuralODE that have a similar structure but instead to having many layers they have only one shared layer as it is possible to see from figure 14.
This makes the NeuralODE more memory efficient. In fact the idea proposed by the authors is that they consider the continuous limit of each discrete layer in the ResNet, so that the transformation of the hidden state become continuous

$$\frac{\partial z}{\partial t} = f(z(t), t, \theta_t) \tag{20}$$

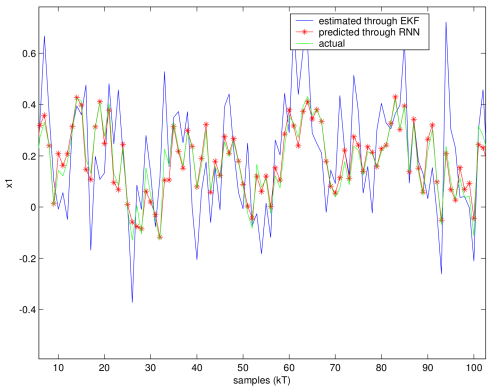


Figure 7: Comparison between RNN and EKF for estimated state x_1

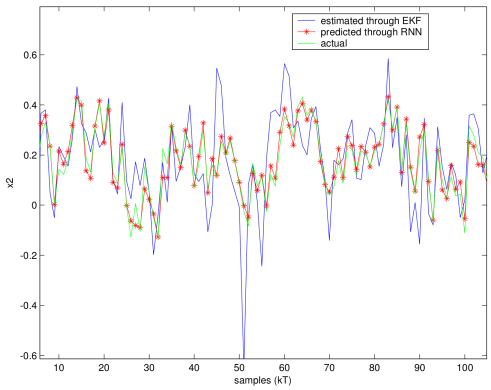


Figure 8: Comparison between RNN and EKF for estimated state x_2

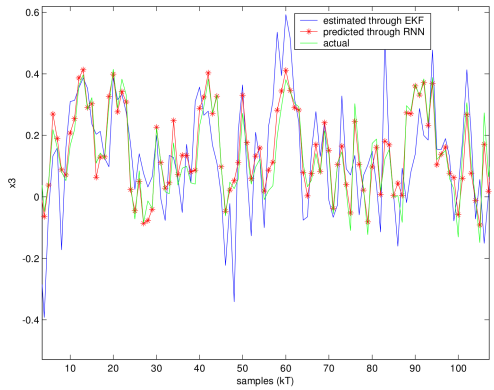


Figure 9: Comparison between RNN and EKF for estimated state x_3

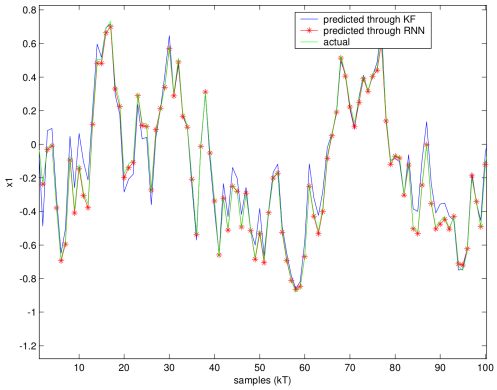


Figure 10: Comparison between RNN and KF for estimated state x_1

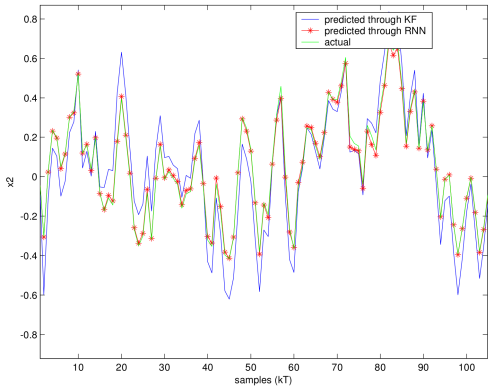


Figure 11: Comparison between RNN and KF for estimated state x_2

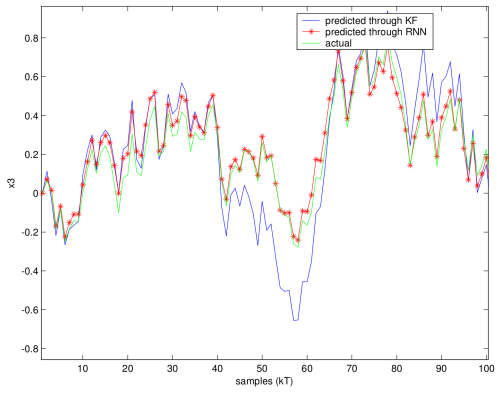


Figure 12: Comparison between RNN and KF for estimated state x_3

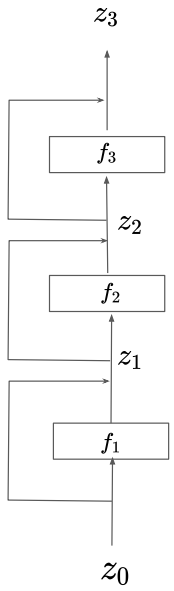


Figure 13: ResNet architecture

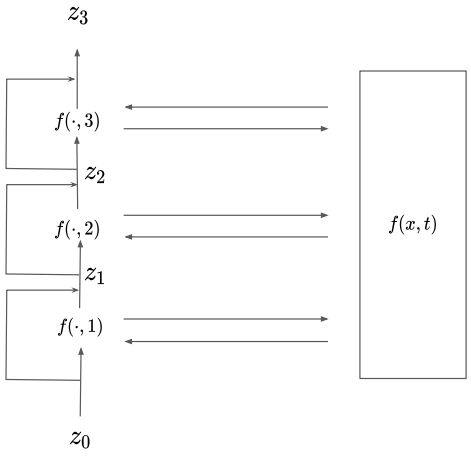


Figure 14: NeuralODE architecture

Instead the innovation behind the NeuralODE is the back propagation through depth. To understand how it woks it is necessary to compute the loss function given by

$$\mathcal{L}(t_0, t_1, \theta_t) = \mathcal{L}(ODESolve(z(t_0), t_0, t_1, \theta, f)) \tag{21}$$

where $ODESolve$ is any solver and f is the NN. Then it is necessary to compute the gradient of the loss with respect to the hidden state

$$\frac{\partial \mathcal{L}}{\partial z(t)} \tag{22}$$

that depend on time (depth). Since to compute this time derivative all the hidden layers must be traversed in a reverse way keeping track of the result obtained, the authors presented in the paper an efficient technique to compute the gradient using the adjoint method. The adjoint method define the adjoint state

$$a(t) = -\frac{\partial \mathcal{L}}{\partial z(t)} \tag{23}$$

that represent how the loss depends on the hidden state at any time t. The time derivative of the adjoint state is given by

$$\frac{\partial a(t)}{\partial t} = -a(t)^T \frac{\partial f(t, z(t), \theta_t)}{\partial z(t)} \tag{24}$$

The solution of the adjoint state can be written as an integral

$$a(t) = \int -a(t)^T \frac{\partial f(t, z(t), \theta_t)}{\partial z(t)} dt \tag{25}$$

and can be solved numerically using the ODE solver. In other words the gradient is computed using the ODE solve backwards from the final time to the initial time.
The problem that we can deal with NeualODE is to find an approximation of the dynamical function of a system. In particular if we have a dynamical system described

$$\frac{dz}{dt} = f(z(t), t)$$

and we are able to observe a certain number of values along the trajectory in the form of (z_i, t_i) with $i = 1, ..., M$ where M is the number of observation, then using the NeuralODE we can approximate the dynamics $f(z, t)$ with $\hat{f}(z, t, \theta)$ that is a NN.

2.6 Conclusion

In this report, I briefly reviewed two of the most important architecture used in system identification, that are also widely used for many other applications. The first architecture is the LSTM that is an evolution of the RNN, used to overcome the vanishing gradient problem and to capture long and short time dependencies among the inputs. The second architectue is the NeualODE capable to learn the dynamics of a system. This architecture is an evolution of the ResNet more efficient in tems of memory usage. Both the architecture are valid for system identification but the second one is mote promising because it is able to deal with complex system and is more computationally lighter.

3 Neural Ordinary Differential Equations - Facci Matteo

3.1 State of the art

3.1.1 Background

As mentioned above in section 1, the main network structures suitable for the purpose of systems identification are the Recurrent Neural Networks (RNNs), typically used to face time-series prediction's problems, or the relatively new Residual Networks (ResNet), developed to reduce problems inherent with very deep networks, with both structures containing repeating blocks of layers that can retain sequential information to be modified at each step through a learned function.
In particular for the latter, in order to combat the lack of accuracy due to the high number of layers, shortcut connections were included in the network. This means that the input of one layer is directly added to the transformed input and the sum of these components comprises the layer's output (*skip connection*). This is called a residual block and it is the basic unit of a ResNet.

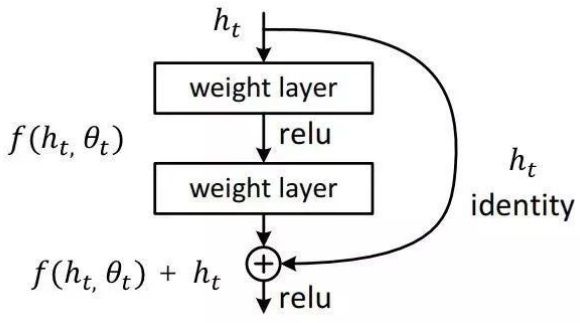


Figure 15: The *skip connection* in a residual block.

Skip connections help information flow through the network by sending the hidden state along with the transformation by layer t to layer $t+1$, preventing important information from being discarded by f.

3.1.2 From sequences of transformations to Neural ODE

The aforementioned neural network architectures may in general be described by the equation

$$\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t, \theta_t) \tag{26}$$

where h_t is the "hidden" information at time t and $f(h_t, \theta_t)$ is the learned function of the current hidden information with parameters θ_t. One of the main problems consists in how to improve the results obtained by using these networks performing a gradual reduction of the stepsize $[t, t+1]$. This can be imagined as an increase in the number of evaluations in an RNN or the number of residual layers in a ResNet. This leads to a differential version of the (26)

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), t, \theta) \tag{27}$$

that is an Ordinary Differential Equation (ODE), since the solution is the function $\mathbf{h}(t)$.
By solving the latter, the desired sequence of hidden states is obtained. The equation needs to be solved during each evaluation, starting from an initial state $\mathbf{h}_0$. This is the Initial Value Problem (IVP).

Chen et al. introduced a new family of ANNs called Neural Ordinary Differential Equations (ODENet)[6], basing on the fact that skip connections in ResNet are similar to a realization of Euler's method for the numerical solution of the ODEs, in fact, Euler's method is a discretization of the continuous relationship between the input and output domains of the data. Neural networks are also discretizations of this continuous relationship through hidden states in a latent space (for instance, RNNs create a pathway through this latent space by allowing states to depend directly on each other, similar behavior of Euler's method).
The skip connections are discretized time steps of the Euler's method, more precisely a chain of residual blocks in a neural network is basically a solution of the ODE with the Euler method, this means that it is possible to regulate the depth of the neural network by only choosing the discretizing scheme, that is making the solution more or less accurate and increasing the number of layers to infinity.

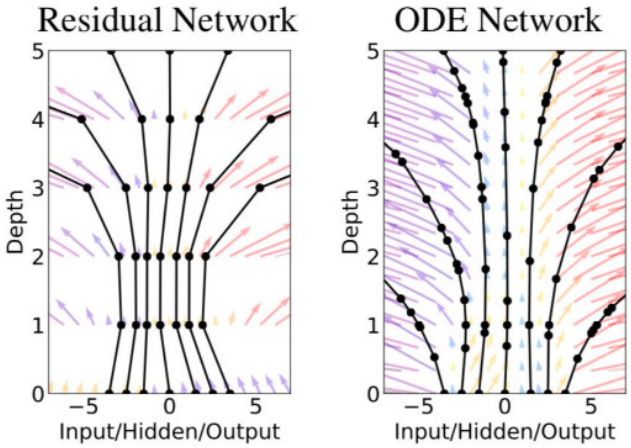


Figure 16: Difference between ResNet with a fixed number of layers and ODENet with a flexible number of layers.

ODENet, instead of learning the residuals between fixed points, parametrizes the local derivative of the input data with an ANN block.
An ODE solver is responsible for the steps to be taken from an input $\mathbf{h}(t_i)$ at time t_i to an output $\mathbf{h}(t_{i+1})$ considering the derivative at each point.
One of the strengths of ODENet compared to ResNet is precisely the choice of the ODE solver. Using an ODE solver with an adaptive step length, in order to step between any two-time points, permits this architecture to handle time-series with irregular measurements and perform the right trade-off between accuracy and computational speed, by considering larger stepsizes. In their paper, Chen et al. proposed that an ODENet block is useful for modeling and making predictions of time-series, and this leads to a capability to model and learn dynamical systems from data without prior knowledge of the underlying dynamics.
This is due to the fact that ODENet learns the local derivative $\frac{d\mathbf{h}(t_i)}{dt}$ at input point $\mathbf{h}(t_i)$ necessary for the ODE solver to step to the output $\mathbf{h}(t_{i+1})$. In this way, the ODE is directly encoded in the ANN during the training phase.
After that phase, the network can take as input any point $\mathbf{h}(t)$ and predict the next point $\mathbf{h}(t + \Delta t)$, basing on the derivative computed by the ODE solver at each step and encoded in the network.
This aspect permits the network to predict the dynamics of the system described by a corresponding ODE.

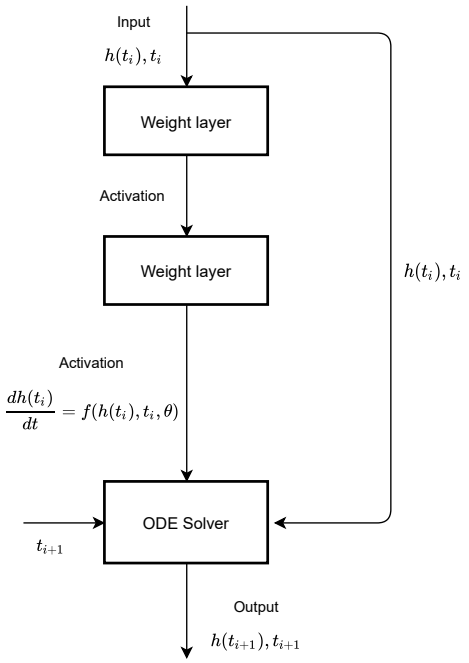


Figure 17: The ODENet architecture composed by two layers connected to an ODE solver.

Despite the relevant result obtained, which is analyzed more in detail in the following sections, previous researches have been carried out on the subject and the usage of particular neural networks for the solution of differential equations. Lu et al.[7], and Haber and Ruthotto [8] built architectures with stepping methods from ODE solvers and ResNet layers. In particular, the latter worked on an interpretation of deep residual CNNs as nonlinear systems of partial differential equations (PDEs) [9]. These papers aimed to improve the ResNet with the usage of ODEs and PDEs, without mentioning anything about learning dynamical systems.
Raissi et al. [10, 11] worked on predicting dynamics governed by PDEs by learning using Gaussian processes and taking steps using numerical ODE solvers. Later, they used a neural network-based approach, with a prior knowledge of the dynamical system to be learned and the dynamics discretized by a multistep ODE solver scheme in order to fit the unknown scalar parameters of their previously developed model [12, 13].
Long et al. [14] developed PDE-Net to learn PDE dynamics taking inspiration from the Euler ODE solver stepping scheme and similarly, also Raissi et al. [15] developed a network using a stepping scheme inspired by ODE solvers from the linear multistep family. Both networks, differently from ODENet are based on data sampled at regular short time intervals.
After the ODENet's release, Ayed et al. [16] focused on forecasting dynamical systems from data following an unknown ODE. They followed the same approach used in ODENet by approximating the unknown ODE with a ResNet module.
Euler's forward method has been adopted as ODE solver while ODENet allows any ODE solver to be used. This aspect permitted to know all the steps in the solver, leading to a simpler training phase with less flexibility in the ODE solver. Furthermore, the usage of a fixed time-step solver causes the lack of capability of training with data sampled at different time intervals.

3.2 ODENet

3.2.1 Ordinary differential equations

ODEs describe a dynamical system and, in general, the rate of change of a variable $y(t)$ with respect to an independent variable, in this case, the time t.

An explicit ODE of the n-th order has the form

$$y^{(n)} = f(t, y, y', y'', ..., y^{(n-1)}, \theta) \tag{28}$$

where f is the differential, a function of the independent variable t, the dependent variable $y(t)$ and its derivatives with respect to time, that is $y^{(i)} = \frac{d^i y}{dt^i}$ with $i = 1...n$, and an optional parameter θ. A higher-order ODE, with derivatives of order greater than one, can always be re-expressed as a system of first-order ODEs by redefining the higher-order differentials as new variables.
A system of N linear first-order ODEs can be written in matricial form as

$$y' = Ay + b \tag{29}$$

with the $N \times N$ coefficient matrix A and the $N \times 1$ constant vector b.

3.2.2 Initial value problems (IVPs)

A solution $y(t) = G(t)$ to a first-order ODE is only uniquely defined up to an unknown integration constant. Similarly, to an n-th order ODE correspond n integration constants. This aspect implies that, for a full solution, knowledge of an initial condition (t_0, y_0) is required to define these constants. An ODE together with a given initial condition is known as an initial value problem (IVP). And the solution $y(t)$ to the IVP satisfies both the ODE and the initial condition $y(t_0) = y_0$.
More precisely an IVP for a first-order ODE is defined as:

$$\frac{dy(t)}{dt} = f(y(t), t, \theta) \tag{30}$$

$$y(t_0) = y_0 \tag{31}$$

supposing $y(t_1)$ is needed, the solution is stated as:

$$y(t_1) = y(t_0) + \int_{t_0}^{t_1} f(y(t), t, \theta)dt \tag{32}$$

If the (32) cannot be analytically integrated one has to use approximation methods, i.e. numerical integration.
One of the most simple and well-known is the aforementioned Euler's method which, starting from the initial condition $y(t_0) = y_0$, takes linear steps along the ODE $y'(t) = f(t, y(t))$ with a fixed time-step h. This means that the updating law is:

$$y(t_{i+1}) = y(t_i) + hf(y(t_i), t_i) \tag{33}$$

with $t_{i+1} = t_i + h$ and f is the derivative.

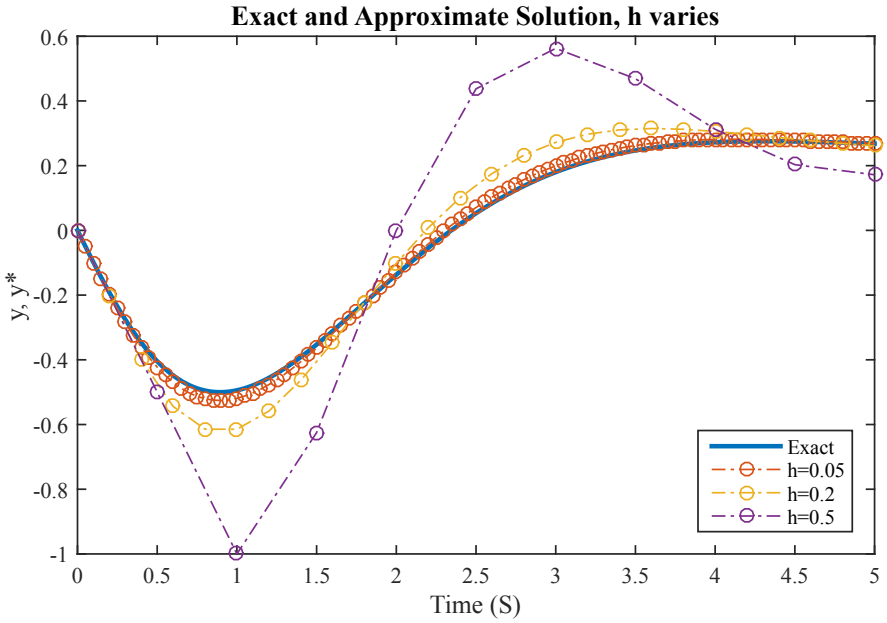


Figure 18: Euler's method with several values of h, the solution is better with smaller values of h.

Most methods for numerically solving IVPs are made for systems of first-order ODEs since all higher-order ODEs can be rewritten as such. Other first-order solvers are the ones belonging to the Runge-Kutta family, and they are based on the idea that instead of going from y_i to y_{i+1} directly, a set of intermediate steps are taken at fractions of the step size h and added in a weighted average.
Therefore, summarizing Euler's method takes a single step in the direction of the local derivative. Runge-Kutta takes several intermediate shorter steps in the time interval, and from these constructs a final long step.
There are many other ODE solvers to choose from, the main criteria to care about for the choice are stiffness, number of calculations per iteration, implicit or explicit solver, single-step size or multi-step size (adaptive).
However, all of these methods are computationally intensive, especially during the training phase, which requires differentiation of the integration steps to be able to add up all gradients of the network parameters θ, incurring a high memory cost.
In the paper, Chen et al. presented an alternative approach to calculating the gradients of the ODE by using the so-called adjoint method by Pontryagin. This method works by solving a second augmented ODE backward in time, and can be used with all ODE solvers, and has a low memory cost.

3.2.3 Backpropagation using the adjoint sensitivity method

The function approximation problem now takes place over a continuous hidden state dynamics and t_i and t_{i+1} can be considered as two additional free parameters to be optimised with a numerical ODE solver called ODESolve: $\mathbf{z}(t_{i+1}) = ODESolve(\mathbf{z}(t_i), f, t_i, t_{i+1}, \theta)$, where f is a neural network.
The central innovation of the paper is an algorithm for backpropagating (reverse-mode differentiation) through the continuous hidden state dynamics.
Even though backpropagating through the ODE solver steps is straightforward, the memory cost is high and additional numerical errors occur. In order to perform backpropagation, the gradient of the loss function with respect to all parameters must be computed.

As different ODE solvers may take a varying number of steps when numerically integrating between two time points, a general method for computing the gradient of the loss at each intermediate step is required. The adjoint sensitivity method can be used regardless of the choice of the ODE solver and with constant memory consumption.
Therefore, a differentiable loss function is defined as:

$$L(\mathbf{z}(t_1)) = L\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) = L(ODESolve(\mathbf{z}(t_0), f, t_0, t_1, \theta) \tag{34}$$

Alongside the original dynamical system that describes the process, the adjoint system describes the derivative states at each point of the process backward, via the chain rule. In this way it is possible to obtain the derivative by the initial state, and, similarly, by the parameters of a function that is modeling the dynamics (one "residual block", or the discretization step in the "old" Euler's method).

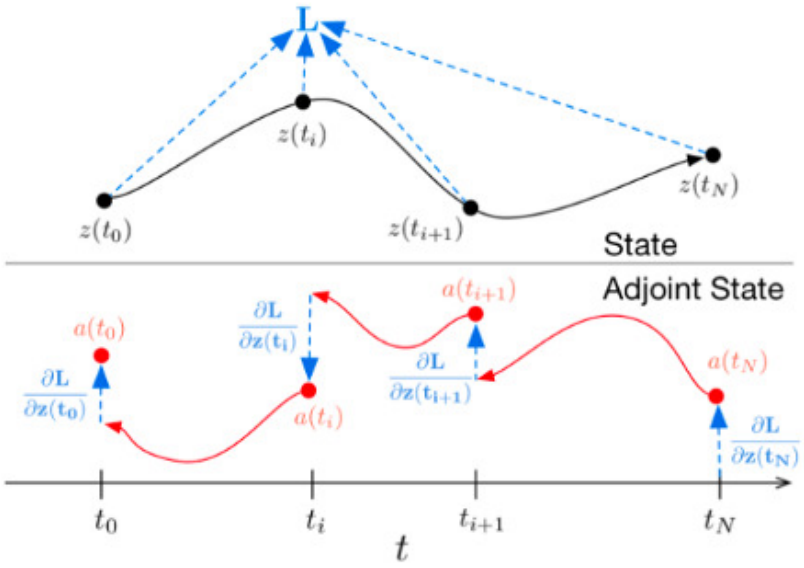


Figure 19: Backpropagation gradients for the ODESolve() method.

The adjoint method works by constructing the so-called adjoint state $a(t) = \frac{dL}{d\mathbf{z}(t)}$ where L is the loss function and $\mathbf{z}(t)$ is the output after each step taken by the ODE solver, which follows the well-known differential equation

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), t, \theta) \tag{35}$$

The adjoint state follows the differential equation

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} \tag{36}$$

$$\frac{d\mathbf{a}(t)}{dt} = \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t)}{\varepsilon} \tag{39}$$

$$= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\frac{\partial}{\partial \mathbf{z}(t)} T_\varepsilon(\mathbf{z}(t))}{\varepsilon} \quad \text{(by Eq 38)} \tag{40}$$

$$= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\frac{\partial}{\partial \mathbf{z}(t)} \left(\mathbf{z}(t) + \varepsilon f(\mathbf{z}(t), t, \theta) + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \quad \text{(Taylor series around } \mathbf{z}(t)\text{)} \tag{41}$$

$$= \lim_{\varepsilon \to 0^+} \frac{\mathbf{a}(t+\varepsilon) - \mathbf{a}(t+\varepsilon)\left(I + \varepsilon \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon^2)\right)}{\varepsilon} \tag{42}$$

$$= \lim_{\varepsilon \to 0^+} \frac{-\varepsilon \mathbf{a}(t+\varepsilon)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon^2)}{\varepsilon} \tag{43}$$

$$= \lim_{\varepsilon \to 0^+} -\mathbf{a}(t+\varepsilon)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} + \mathcal{O}(\varepsilon) \tag{44}$$

$$= -\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} \tag{45}$$

Figure 20: Original proof from paper (Appendix B).

Solving the differential equation backwards in time, from time t_N to t_0, similarly to a regular backpropagation, one obtains the gradients with respect to the hidden state at any time as:

$$\frac{dL}{d\mathbf{z}(t_N)} = \mathbf{a}(t_N) \tag{37}$$

$$\frac{dL}{d\mathbf{z}(t_0)} = \mathbf{a}(t_0) = \mathbf{a}(t_N) + \int_{t_N}^{t_0} \frac{d\mathbf{a}(t)}{dt} dt = \mathbf{a}(t_N) - \int_{t_N}^{t_0} \mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}(t)} dt \tag{38}$$

One can simply integrate backward in order to get the gradient at all different timesteps.
To find the initial value for the adjoint differential equation, the chain rule starting from the loss function is used, therefore also the gradients with respect to t_0, t_N and θ are needed. At the start , t and θ are considered as states with constant time derivatives such that:

$$\frac{dt(t)}{dt} = 1 \tag{39}$$

$$\frac{d\theta(t)}{dt} = 0 \tag{40}$$

and they can be combined with $z(t)$ to form the augmented state

$$f_{aug}(\begin{bmatrix}\mathbf{z} & \theta & t\end{bmatrix}) = \frac{d}{dt}\begin{bmatrix}\mathbf{z} \\ \theta \\ t\end{bmatrix} = \begin{bmatrix}f(\mathbf{z}(t), t, \theta) \\ 0 \\ 1\end{bmatrix} \tag{41}$$

and the augmented adjoint state

$$\mathbf{a}_{aug}(t) = \begin{bmatrix}\mathbf{a} \\ \mathbf{a}_\theta \\ \mathbf{a}_t\end{bmatrix}(t) \tag{42}$$

$$\mathbf{a}_\theta(t) = \frac{dL}{d\theta(t)} \tag{43}$$

$$\mathbf{a}_t(t) = \frac{dL}{dt(t)} \tag{44}$$

obtaining the ODE

$$\frac{\mathbf{a}_{aug}(t)}{dt} = -\begin{bmatrix}\mathbf{a}(t) & \mathbf{a}_\theta(t) & \mathbf{a}_t(t)\end{bmatrix}\frac{\partial f_{aug}(t)}{\partial\begin{bmatrix}\mathbf{z} & \theta & t\end{bmatrix}} \tag{45}$$

where $\frac{\partial f_{aug}(t)}{\partial\begin{bmatrix}\mathbf{z} & \theta & t\end{bmatrix}}$ is the Jacobian of the augmented state

$$\frac{\partial f_{aug}(t)}{\partial\begin{bmatrix}\mathbf{z} & \theta & t\end{bmatrix}} = \begin{bmatrix}\frac{\partial f}{\partial \mathbf{z}} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0}\end{bmatrix}(t) \tag{46}$$

The gradient with respect to the model parameters $\frac{dL}{d\theta}$ can be obtained setting $\mathbf{a}(t_N) = \mathbf{0}$ and integrating the second element of (45) backward in time. One obtains:

$$\frac{dL}{d\theta} = \mathbf{a}_\theta(t_0) = -\int_{t_N}^{t_0}\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial\theta}dt \tag{47}$$

At the end, the gradients at time instant t_0 and t_n are given by:

$$\frac{dL}{dt_N} = \mathbf{a_t}(t_N) = \frac{dL}{d\mathbf{z}(t_N)}\frac{d\mathbf{z}(t_N)}{dt_N} = \mathbf{a}(t_N)f(\mathbf{z}(t_N), t_N, \theta) \tag{48}$$

$$\frac{dL}{dt_0} = \mathbf{a_t}(t_N) - \int_{t_N}^{t_0}\mathbf{a}(t)\frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial t}dt \tag{49}$$

Therefore all the gradients needed to backpropagate through the ODE solver are computed by solving the ODE for the augmented state backward in time.

3.2.4 Learning dynamical systems

Since ODENet approximates the derivative of a system, it can be used as a tool for modeling dynamical systems. In fact, it is possible to learn the underlying dynamics of the system by integrating the model between any two data points in the data set and then backpropagating to update the model parameters.
As the model is continuously defined, it can be integrated between any two time points, avoiding the problem of modeling data with irregular time steps.
One of the applications proposed by the authors regards the time-series modeling through ODEs having irregularly sampled data, in particular ill-defined data, problems with missing data in some time intervals, or simply inaccurate latent variables.
Some approaches concatenate time information to the input of an RNN, but these aren't able to solve the problem at all.
A solution to this, based on the ODENet module, is a continuous-time generative model, which, given an initial state $\mathbf{z}(t_0)$ and observation times $t_0...t_N$, computes the latent states $\mathbf{z}(t_1)...\mathbf{z}(t_N)$ and the outputs $\mathbf{x}(t_1)...\mathbf{x}(t_N)$:

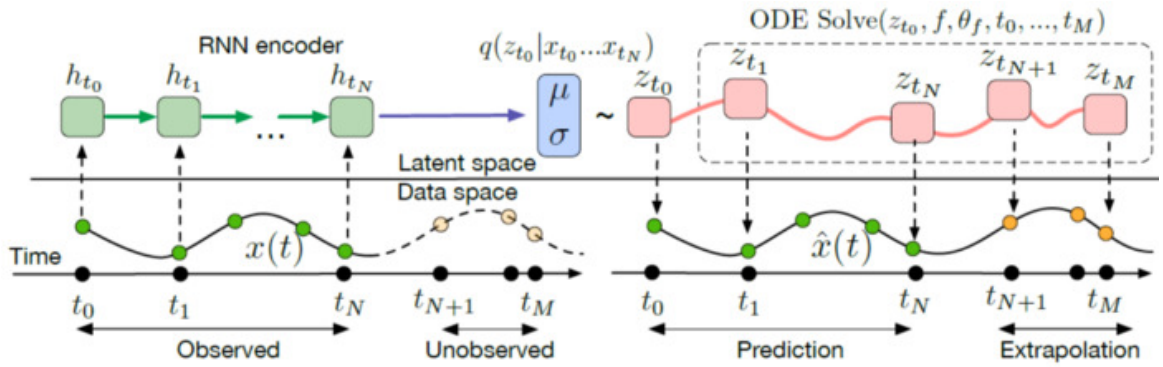
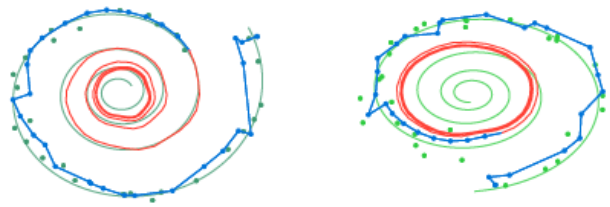
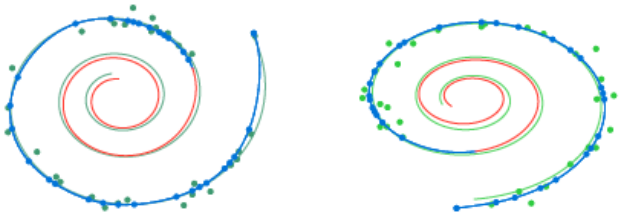


Figure 21: Computation graph of the latent ODE model.

The function f (a neural network function) is responsible for calculating the latent state $\mathbf{z}$ at any time t beginning at the current timestep. The model is a variational autoencoder that encodes the past trajectory (green in the figure above) in the initial latent state $\mathbf{z}(t_0)$ using an RNN.
The latent state distribution is hereby captured through the parameters of the distribution (a Gaussian with mean μ and standard deviation σ). From this distribution, a sample is taken and passed through ODENet.
The architecture has been tested on a dataset of bi-directional two-dimensional spirals, sampled at irregular time points with the addition of Gaussian noise. The following figure shows qualitatively the better performance of the Neural ODE:



(a) Recurrent Neural Network



(b) Latent Neural Ordinary Differential Equation

Figure 22: Dots are sampled noisy trajectories, the blue line is the true trajectory, the orange line stands for recovered and interpolated trajectory.

3.2.5 Training Neural ODEs

For sake of completeness, a simple neural network with neural ordinary differential equations (ODEs) to learn the dynamics of a physical system has been trained.
The full section has been developed with the help of Matlab and its Deep Learning Toolbox, which provides a framework for designing and implementing deep neural networks with algorithms, pretrained models, and useful apps, on a laptop with a single CPU (Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz - RAM 8GB).

the learned function $f(\mathbf{y}(t), t, \theta)$ has been modeled with a `dlnetwork` object embedded into a custom layer.
This example aims to learn the dynamics x of a given physical system $x' = Ax$ (a simple spiral function), where A is a 2×2 matrix.
`ode45` solver has been used for computing the solution of the same model from additional initial conditions.

$$x_0 = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \tag{50}$$

$$A = \begin{bmatrix} -0.1 & -1 \\ 1 & -0.1 \end{bmatrix} \tag{51}$$

After a training phase of 1500 iterations with a mini-batch-size of 200, and considering that every 50 iterations a solution is given by the ODE solver about the learnt dynamics, the final good results are the followings:

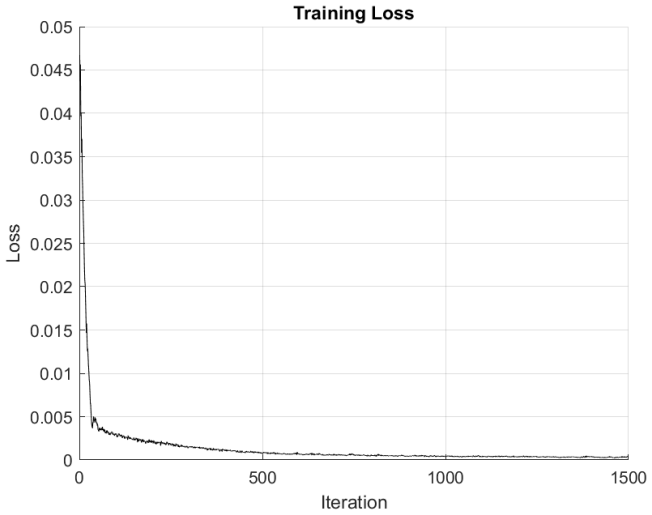


Figure 23: Loss after 1500 iterations.

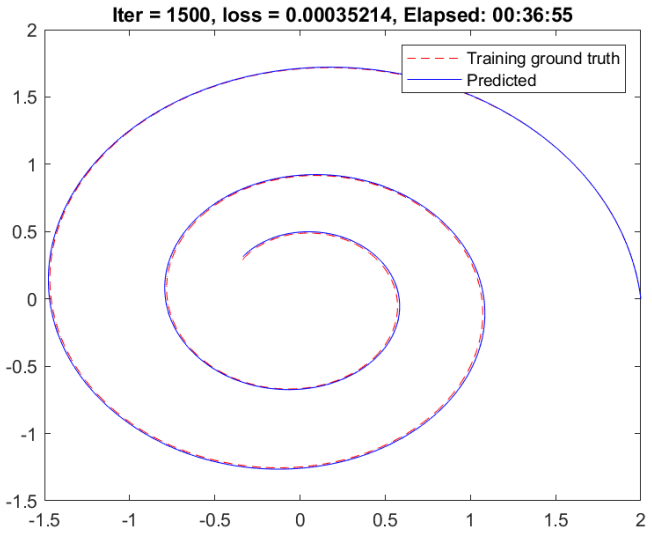


Figure 24: Phase diagram.

And for what concerns the evaluation of the trained model, it has been used as a right-hand side for the same problem with different initial conditions, solved numerically the ODE learned dynamics with `ode45` and compared the results with the true model.

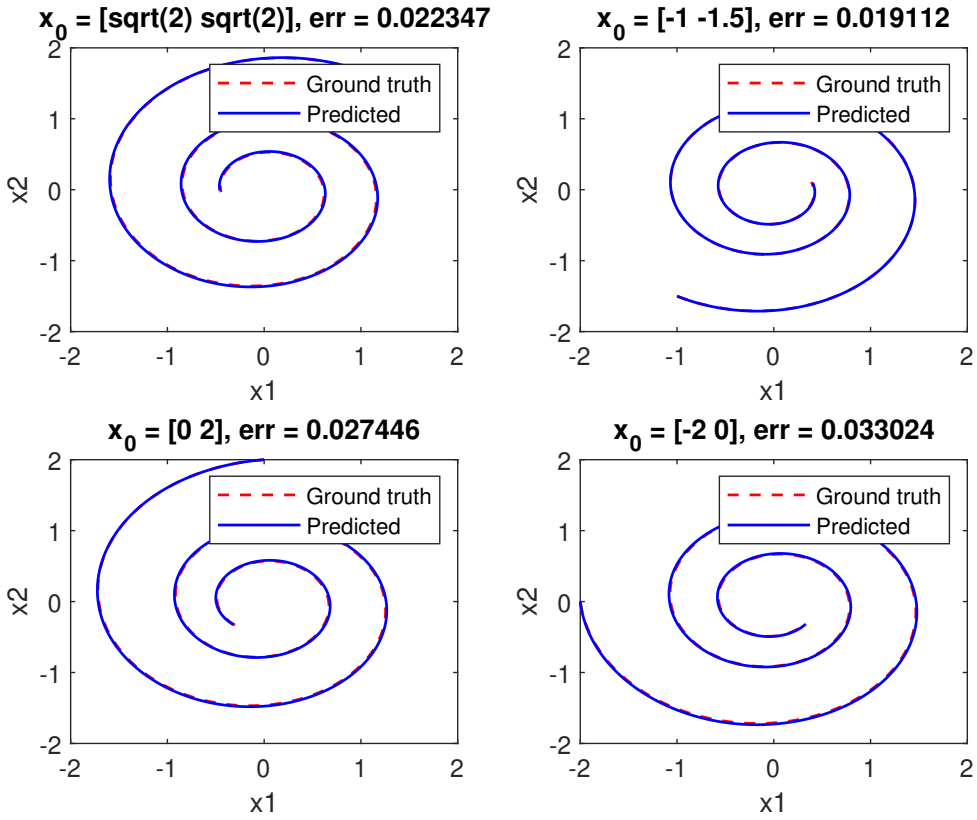


Figure 25: Phase diagram.

3.2.6 Limitations of Neural ODE

If the map which is going to be modeled cannot be described by a vector field, i.e. the data does not represent a continuous transformation, and since a Neural ODE is a continuous transformation that cannot lift data into a higher dimension, it would try to smush around the input data to a point where it is mostly separated. However, this wrong approach often leads to the network learning overly complicated transformations. Dupont et al. [17] introduced the concept of Augmented Neural ODEs, providing a few examples of intractable data for a Neural ODE, and tried to point out a solution to the problem, that is the increase of the dimensionality of the data.
The way to encode this into the Neural ODE architecture is to increase the dimensionality of the space the ODE is solved in, e.g. if the hidden state is a vector in $\mathbb{R}^n$, it is possible to add m more dimensions and solve the ODE in $\mathbb{R}^{(n+m)}$, in such a way that the augmented ODE becomes:

$$\frac{d}{dt}\begin{bmatrix} h(t) \\ a(t) \end{bmatrix} = f\left(\begin{bmatrix} h(t) \\ a(t) \end{bmatrix}, t, \theta\right) \tag{52}$$

$$\begin{bmatrix} h(0) \\ a(0) \end{bmatrix} = \begin{bmatrix} x \\ 0 \end{bmatrix} \tag{53}$$

Therefore there is a simple concatenation of a vector of zeros to the end of each datapoint x, allowing the network to learn some values for the extra dimensions. The data can be arranged

in a linearly separable form with extra freedom, and the extra dimensions can be ignored when using the network. In this way, instead of learning a complicated map in a given dimension, the augmented Neural ODE learns a simpler map with a higher dimension. This permits the validation error to be almost null, against the results of the standard Neural ODE. A drawback of this solution is the introduction of more parameters, compensated by the degree of freedom that can lead to more feasible solutions.
Another criticism is that adding dimensions reduces the interpretability and elegance of the Neural ODE architecture. Extra dimensions may be unnecessary and may influence a model away from physical interpretability. Therefore, augmenting the hidden state is not always the best idea, and augmented Neural ODEs are useful only in isolated situations. Practically, Neural ODEs should be used for areas in which a smooth transformation increases interpretability and results, potentially areas like physics and irregular time series data.

3.3 Conclusion

Chen et al. presented an innovative and interesting approach to thinking about neural networks but it is not already proven that it would be effective for each kind of dynamical system.
Neural ODE can help to build continuous-time time series models, which can easily handle data coming at irregular intervals. However, ODEs can only model deterministic dynamics.
In the case of data sampled at regular time intervals (like video or audio), there would not be any advantage with respect to other standard approaches which can be simpler and faster in that situation.
Anyway, there are some other considerations to be done about the architecture analyzed.
The first one regards mini-batching, in fact, it might be an issue with the approach, since batching together multiple samples requires solving a system of ODEs at once. This can multiply the number of required evaluations significantly.
The other is that the uniqueness of the ODE solution is only guaranteed if the network has finite weights and uses Lipschitz nonlinearities such as $tanh$ or $relu$. In addition, conventional nets can be evaluated exactly with a fixed amount of computation, and are typically faster to train, without an error tolerance for the solver to be considered.
However, an important advantage of the method presented in the paper is that one may choose the balance between a fast and a precise solution by affecting the precision of the numerical integration during training and evaluation phases, this leads to a constant memory cost at training time and an adaptive time cost. The last aspect is due to the approximation of the solution of the ODE, in fact, sometimes only a few iterations of the solver are needed to get an acceptably good answer, and this could save time.
In conclusion, despite the numerous defects of various kinds, the most important aspect is that this method, this architecture, is generally applicable (requiring only that the nonlinearities of the neural network be Lipschitz-continuous) and may be applied to time-series modeling, supervised learning, density estimation or other sequential processes, and above all system identification tasks.

4 Conclusions

Various methods and architectures have been analyzed in order to perform systems identification. First, a focus on LSTM has been done, in particular on its capability to deal with a sequence of input data that have self-dependencies and how this can be used for system identification. Then, an in-depth study on neural ODEs follows immediately after. Neural Ordinary Differential Equations are able to parametrize the ODE which describes the dynamical system and produce a very accurate approximation of the resulting derivative. This means that this architecture can be one of the most valid between the solutions analyzed for modeling the dynamics of a system.
It is possible to conclude that NNs for system identification is a valid technique because they are capable to handle complex systems without the specific knowledge of physics behind the model. The price to pay is the huge computational power needed to train the network and also the necessity to collect the data provided by the systems. Despite the big drawback, NNs for system identification will play an important role in the system identification field thanks to their great power.

References

[1] K. S. Narendra and K. Parthasarathy, "Identification and control of dynamical systems using neural networks," in IEEE Transactions on Neural Networks, vol. 1, no. 1, pp. 4-27, March 1990, doi: 10.1109/72.80202.

[2] M. C. Nechyba and Yangsheng Xu, "Neural network approach to control system identification with variable activation functions," Proceedings of 1994 9th IEEE International Symposium on Intelligent Control, 1994, pp. 358-363, doi: 10.1109/ISIC.1994.367791

[3] Jeen-Shing Wang and Yen-Ping Chen, "A fully automated recurrent neural network for unknown dynamic system identification and control," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 53, no. 6, pp. 1363-1372, June 2006, doi: 10.1109/TCSI.2006.875186.

[4] N. Yadaiah and G. Sowmya, "Neural Network Based State Estimation of Dynamical Systems," The 2006 IEEE International Joint Conference on Neural Network Proceedings, 2006, pp. 1042-1049, doi: 10.1109/IJCNN.2006.246803.

[5] Ogunmolu, Olalekan P. et al. "Nonlinear Systems Identification Using Deep Dynamic Neural Networks." ArXiv abs/1610.01439 (2016)

[6] Chen, Ricky T. Q., Yulia Rubanova, Jesse Bettencourt, David Duvenaud
Neural Ordinary Differential Equations
19 June 2018

[7] Yiping Lu, Aoxiao Zhong, Quanzheng Li, Bin Dong
Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations
27 October 2017

[8] Lars Ruthotto, Eldad Haber
Stable Architectures for Deep Neural Networks
9 May 2017

[9] Lars Ruthotto, Eldad Haber
Deep Neural Networks Motivated by Partial Differential Equations
12 April 2018

[10] Maziar Raissi, Paris Perdikaris, George Em Karniadakis
Numerical Gaussian Processes for Time-dependent and Non-linear Partial Differential Equations
29 March 2017

[11] Maziar Raissi, George Em Karniadakis
Hidden Physics Models: Machine Learning of Nonlinear Partial Differential Equations
2 August 2017

[12] Maziar Raissi, Paris Perdikaris, George Em Karniadakis
Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations
28 November 2017

[13] Maziar Raissi, Paris Perdikaris, George Em Karniadakis
Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations
28 November 2017

[14] Zichao Long, Yiping Lu, Xianzhong Ma, Bin Dong
PDE-Net: Learning PDEs from Data
26 October 2017

[15] Maziar Raissi, Paris Perdikaris, George Em Karniadakis
Multistep Neural Networks for Data-driven Discovery of Nonlinear Dynamical Systems
4 January 2018

[16] Ibrahim Ayed, Emmanuel de Bézenac, Arthur Pajot, Julien Brajard, Patrick Gallinari
Learning Dynamical Systems from Partial Observations
26 February 2019

[17] Emilien Dupont, Arnaud Doucet, Yee Whye Teh
Augmented Neural ODEs
2 April 2019