UNIVERSITAT DE BARCELONA

FUNDAMENTAL PRINCIPLES OF DATA SCIENCE MASTER'S THESIS

Reservoir computing for learning the underlying dynamics of sequential data points

Author:
Núria Sánchez Font

Supervisor:
Dr. Oriol Pujol Vila

A thesis submitted in partial fulfillment of the requirements for the degree of MSc in Fundamental Principles of Data Science

in the

Facultat de Matemàtiques i Informàtica

June 29, 2020

UNIVERSITAT DE BARCELONA

Abstract

Facultat de Matemàtiques i Informàtica

MSc

Reservoir computing for learning the underlying dynamics of sequential data points

by Núria Sánchez Font

Reservoir computing (RC) is a learning technique used to infer the underlying dynamics given a set of sequential data points. For instance, it may learn the dynamics of an input sequence in order to produce a related output sequence or it may learn the dynamics of a certain data in order to be capable of predicting the following time steps. The neural network employed is composed by a single hidden layer along with an input and output layers. As we will see, reservoir computing is a recurrent neural network approach but with the main difference that it deterministically sets all the connections within the different components of the network with the exception of the output connections, since these will be the connections to be learnt. This is possible because of the so called *echo states*, which is the key concept behind the reservoir computing approach. Therefore, reservoir computing needs to learn a much lower number of parameters, which makes it computationally cheaper than other RNN approaches. However, this is not the only difference. As will be exposed later on, the learning procedure consists on performing a linear regression, which is less costly than the usual backpropagation.

The reservoir computing technique has recently gained a lot of popularity thanks to the work of chaos theorist Edward Ott and four collaborators at the University of Maryland in the area of chaotic dynamical systems (Pathak et al., 2018b and Pathak et al., 2017). In that work, they were able to predict the dynamics of some chaotic systems up to 8 Lyapunov times, which is an impressive distant horizon.

Acknowledgements

First of all, I would like to thank my supervisor, Oriol Pujol, for all the time he has been willing to spend discussing the different issues that came along during the project. His constant support allowed me to get the most out of this experience.

Moreover, I would like to thank the partners of the master, who have made this adventure more enjoyable.

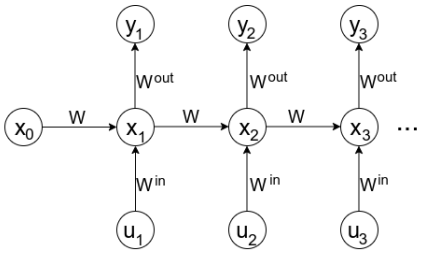
Finally, I would also like to thank my family and friends for their continuous encouragement throughout my years of study.

Contents

Chapter 1

Introduction

The most common approach when dealing with sequential series is to use recurrent neural networks (RNN) due to their capacity of storing information about the past. That is, at each time step, they input a point of the sequential series along with a memory component, which stores information regarding the previous time-steps. In addition, it is also possible to input information regarding the previous outputs. The below picture depicts an example of a vanilla RNN



where $x_t = \tanh(Wx_{t-1} + W^{\text{in}}u_t)$ are the hidden states and $y_t = \text{softmax}(W^{\text{out}}x_t)$ the predictions at time t. The main downside of such approach is the *exploding* or *vanishing gradient* encountered when performing the backpropagation. In addition, all the matrices giving the connections between the different elements have to be learnt. In order to overcome such difficulties, the so called *echo states network* is introduced.

Under certain conditions, the hidden state of a RNN can be univocally determined by the previous input history and, if it has feedback connections, by the previous outputs as well. That is, there exists a function E such that

$$x(n) = E(..., u(n-1), u(n); ..., y(n-2), y(n-1)). \tag{1.1}$$

Therefore, the hidden state can be thought of as an *echo* of the past history. As we are going to further develop along the project, the fact that the hidden states are an echo of the past allows to deterministically define the matrices of connections W^{in} and W, in such a way that the only weights to be learnt are the output ones (W^{out}). In addition, such weights are going to be learnt by performing a linear regression. Therefore, the existence of this E function drastically decreases the computational cost of training. This learning technique is the so called *reservoir computing*.

The truth is that the reservoir computing technique has recently gained a lot of popularity thanks to the work of chaos theorist Edward Ott and four collaborators at the University of Maryland in the area of chaotic dynamical systems (Pathak et al., 2018b and Pathak et al., 2017). In it, they trained a neural network, by means of the reservoir computing approach, with a data sequence sampled from the Kuramoto-Sivashinsky equation. Then, the network was able to predict the future evolution

of the system up to 8 Lyapunov times, which is an impressive distant horizon. It is worth noting the fact that the network was solely trained with past data, it did not have any knowledge regarding the equation itself. What makes this result so outstanding is the fact that the system generating the data points do not need to be modelled by people. Therefore, it simplifies a lot the predicting task. The prediction of chaotic dynamics based solely on past data may have a wide range of applications: monitor the heartbeat in order to prevent a heart attack, keep track of a sea's swell in order to predict rough waves that may endanger a ship and its crew, track solar flares in order to predict solar storms that would severely damage Earth's electronic infrastructure and whether forecasting among others.

Therefore, the aim of this project is to study the theory behind reservoir computing as well as analyzing its application on the task of learning a dynamics. On the one hand, the former, could include studying the conditions that guarantee the existence of the function E given in equation (1.1) or understanding the reason why the existence of such function allows to solely train the output connections by means of a linear regression. On the other hand, the latter will entail the construction of an appropriate network in order to infer the underlying dynamics of a given set of sequential data points. Therefore, during testing, the network will be capable of predicting the following time steps.

The project is structured as follows. In Chapter 2, we establish the setting of the reservoir computing learning technique. The details regarding the structure of the neural network as well as its induced dynamics, the intuition behind the so called *echo states* and the details regarding the training and testing of the network are provided in Section 2.1, 2.2 and 2.3 respectively. In addition, Section 2.4 illustrates the main features regarding the dynamics and the topology of an echo states network by means of a simple example. In addition, this chapter determines all the terminology that is going to be used along the project.

Once all the theoretical concepts behind the reservoir computing have been established, we are going to focus on some of its applications. In Chapter 3, the network is going to be trained in order to learn two fundamental dynamics, such are the discrete periodic dynamics (Section 3.1) and the multiple attractor dynamics (Section 3.2). The former can be thought of as training the network in order to make it capable of playing a melody whereas, in the latter, we will see how the network learns a constantly changing dynamics without memorizing any specific task.

Afterwards, in Chapter 4, we aim to train a neural network in order to learn and predict the dynamics given by the so call *Mackey-Glass system*. As Section 4.1 exposes, this system is determined by a certain *delay differencial equation* whose delay parameter (denoted as σ) is going to determine the degree of chaos present on the orbits. Thus we are going to be dealing with $\sigma = 17$, which renders a mildly chaotic dynamics, and $\sigma = 30$, which renders a wilder chaotic dynamics.

The main goal so far was to understand the theoretical basis of the reservoir computing as well as its learning capabilities when dealing with different types of dynamics and, in particular, with chaotic dynamics. For this purpose, the main reference that we followed was Jaeger, 2010, which is the paper where echo states networks were first introduced. At this point, we decided to eventually apply the already acquired knowledge in order to train a neural network so as to make it capable of learning a dynamics of our choice, which was the *standard map*. We opted for such system because, on the one hand, it displays a very challenging behavior and, on the other hand, we did not find any literature regarding the application of the reservoir computing on it. Thus, Chapter 5 is devoted to this task. Section 5.1

gives some intuition on the dynamics defined by the standard map. As already exposed there, it may display chaotic or non-chaotic behaviors according to a certain parameter denoted as K. Afterwards, Sections 5.2, 5.3 and 5.4 present all the choices regarding the parameters and the procedures that gave the best testing result. However, Section 5.5 describes all the approaches that were tried before reaching the best result. Such approaches are worthy to mention and study since, although they may had not worked in this particular case of study, they belong to the set of typical tunning strategies. Therefore, they provide a systematic technique to carry out the fine tunning of the parameters and procedures.

Finally, in Chapter 6, we discuss the ideas that have appeared to be key along the project. In addition, we outline the current main branches of research related with the reservoir computing technique.

Along the project, many experiments have been performed. Although, in this memory, we only provide the main results, all the code can be found on the GitHub repository `https://github.com/NuriaSF/ReservoirComputing_TFM`.

Chapter 2

Basics of reservoir computing

Reservoir computing (RC) is a learning technique used to infer the underlying dynamics given a set of sequential data points. For instance, it may learn the dynamics of an input sequence in order to produce a related output sequence or it may learn the dynamics of a certain data in order to be capable of predicting the following time steps. The neural network employed is composed by a single hidden layer along with an input and output layers. Section 2.1 details the topology of the neural network as well as its induced dynamics. As we will see, reservoir computing is a recurrent neural network approach but with the main difference that it deterministically sets all the connections within the different components of the network with the exception of the output connections, since these will be the connections to be learnt. This is possible because of the so called *echo states*, which is the key concept behind the reservoir computing approach. Section 2.2 introduces the notion of echo states and discusses why they are so powerful. Therefore, reservoir computing needs to learn a much lower number of parameters, which makes it computationally cheaper than other RNN approaches. However, this is not the only difference. As Section 2.3 exposes, the learning procedure consists on performing a linear regression, which is less costly than the usual backpropagation. Finally, Section 2.4 illustrates the performance and main features of reservoir computing by means of a simple example.

Therefore, in this chapter, we are going to establish the setting of the reservoir computing approach as well as the main terminology that will be used along the project.

2.1 Topology and dynamics of the neural network

Let us begin by detailing the structure of the network. The neural network we are going to be dealing with consists on a K-dimensional input denoted by u, on an N-dimensional hidden layer denoted by x and on an L-dimensional output, denoted by y. The nodes of the hidden layer are usually referred to as *internal units*, *internal states* or just *states*. As we will show later, the values of the different elements of the network are going to be updated at each time step, which will define a dynamics over the different elements of the network. Therefore, each component can be thought as a multidimensional time series. That is:

- $u(n)^T = (u_1(n), u_2(n), ..., u_K(n)))$ is a K-dimensional vector giving the input units at time n.
- $x(n)^T = (x_1(n), x_2(n), ..., x_N(n)))$ is an N-dimensional vector giving the internal units at time n.
- $y(n)^T = (y_1(n), y_2(n), ..., y_L(n)))$ is an L-dimensional vector giving the output units at time n.

Since we are dealing with a discrete dynamics, the time step n is going to take values on the set of natural numbers. It is important to keep in mind that y denotes the output produced by the trained network. That is, it stands for the predictions.

Let us now determine the connection matrices between the different layers:

- **Input connections**, $W^{\text{in}} \in \mathcal{M}(\mathbb{R})_{N\times K}$, matrix giving the connection weights from the input units to the internal units. That is, $W^{\text{in}}_{i,j}$ gives the weight of the connection from the j-th input unit to the i-th internal unit.
- **Internal connections**, $\boldsymbol{W} \in \mathcal{M}(\mathbb{R})_{N\times N}$, matrix giving the connection weights between the different internal units. That is, $W_{i,j}$ gives the weight of the connection from the j-h internal unit to the i-th internal unit.
- **Feedback connections**, $W^{\text{back}} \in \mathcal{M}(\mathbb{R})_{N\times L}$, matrix giving the connection weights from the output units to the internal units. That is, $W^{\text{back}}_{i,j}$ gives the weight of the connection from the j-th output unit to the i-th internal unit.
- **Output connections**, $W^{\text{out}} \in \mathcal{M}(\mathbb{R})_{L\times(K+N+L)}$, matrix giving the connection weights from the input, internal and output units to the output units. That is:
  - $W^{\text{out}}_{i,j}$ for $1 \leq j \leq K$ gives the connection weight from the j-th input unit to the i-th output unit.
  - $W^{\text{out}}_{i,j}$ for $K < j \leq K+N$ gives the connection weight from the j-th internal unit to the i-th output unit.
  - $W^{\text{out}}_{i,j}$ for $K+N < j \leq K+N+L$ gives the connection weight from the j-th output unit to the i-th output unit.

Notice that all type of connections within the different objects are allowed. That is, there may be directed connections between the input and the hidden layer, the input and the output, the hidden and the output, the output and the hidden, recurrent connections within the hidden layer and recurrent connections within the output connections. As will be seen in Section 2.2, *from the set of weight matrices, W^{out} is the only one that is going to be learnt whereas the other ones are going to be deterministically defined according to the problem.* Figure 2.1 illustrates the general structure of this type of neural networks.

Despite the fact that any kind of connections among the different parts of the network are allowed, it is usually enough to consider that the output is solely connected to the internal units. Since this is the most common approach in the literature regarding reservoir computing, this is precisely the approach we are going to follow along the project. Notice that, in this case, the output matrix becomes $W^{\text{out}} \in \mathcal{M}(\mathbb{R})_{L\times N}$ and the neural network structure is slightly modified, as illustrated in Figure 2.2.

So far, we have determined the structure of the neural network, that is, we have defined the layers along with their connections. Let us now introduce how these two elements induce a dynamics on the network. On the one hand, the internal state x is going to be updated at each time step as follows

$$x(n+1) = f(W^{\text{in}}u(n+1) + Wx(n) + W^{\text{back}}y(n)), \tag{2.1}$$

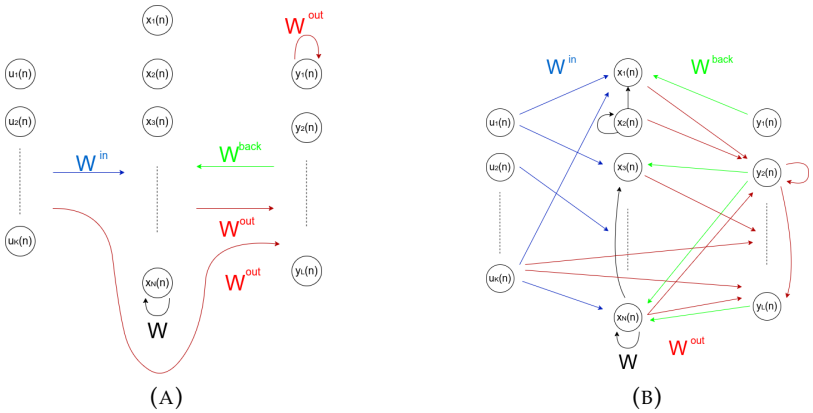


FIGURE 2.1: **(A)** Schema of the general structure of a reservoir's network. **(B)** Example of a neural network following the structure depicted in (A) with the connections explicitly illustrated.

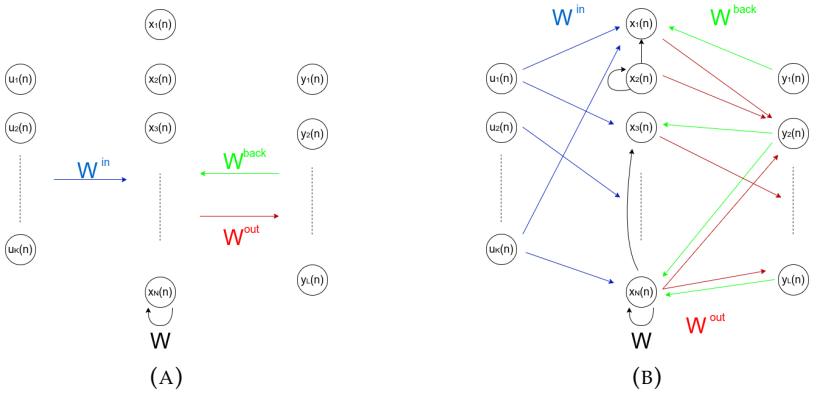


FIGURE 2.2: **(A)** Schema of a reservoir's network where the output only depends on the internal states. This is the structure we are going to consider from now on. **(B)** Example of a neural network following the structure depicted in (A) with the connections explicitly illustrated.

where f is the activation function. On the other hand, the value of the output y is going to be computed as

$$y(n+1) = f^{\text{out}}\left(W^{\text{out}} x(n+1)\right), \tag{2.2}$$

where f^{out} is the activation function. Again, since *tanh* is the most used activation function on the literature, this is the activation function we are also going to implement in our experiments. Notice that it is applied component-wise.

Since the nodes of the hidden layer are time series, we usually talk about the dynamics of the nodes.

2.2 Echo states

According to what we have exposed in Section 2.1, the neural network employed in the reservoir computing approach can be thought of as a *recurrent neural network (RNN)*. However, there is a key concept behind reservoir computing that makes it

different from the other RNN approaches: the existence of *echo states*. Along this section, we are going to develop what echo states are and why they are so powerful.

Echo states were first introduced in Jaeger, 2010, thus all the definitions and results presented in this section are taken from that paper.

Let us begin by introducing some notation. We are going to denote by U^{in} to the space containing the input. That is, $u(n) \in U^{\text{in}}$ for all time step n. In the same way, we are denoting by U^{out} the space where the output is contained. On the other hand, $u^{-\infty}$ stands for *left-infinite* sequences. It is defined as

$$\bar{u}^{-\infty} = \{u(n+i)\}_{i \in -\mathbb{N}} = \{..., u(n-2), u(n-1), u(n)\}.$$

In a similar way, we define $\bar{y}^{-\infty}$. Finally, we will define the operator T as the network state update operator. That is, $x(n+h) = T(x(n), y(n); u(n+1), ..., u(n+h))$. We will say that an state $x \in A$ is end-compatible with some left-infinite sequence $\bar{u}^{-\infty}$ and $\bar{y}^{-\infty}$ if there exists a state sequence $\{..., x(n-1), x(n)\}$ such that $T(x(i), u(i+1), y(i)) = x(i+1)$, and $x = x(n)$.

From now on, we are going to assume the so called *standard compactness conditions*, which are:

- Input and output are drawn from compact spaces U^{in} and U^{out}.
- Internal states lie in a certain compact set A.

The reason for such requirements is due to the fact that, on the one hand, the values of all elements of the network have to be bounded, since, otherwise, the information is going to be lost when applying the *tanh* activation function. Recall that *tanh* collapses values that are large in module to ± 1. On the other hand, we need the spaces to be closed, since, otherwise, the sequences may converge to values that are outside the subspace and, thus, unattainable.

Following the notation introduced so far, we are going to provide the formal definition of echo states.

Definition 2.2.1. Assume standard compactness conditions.

(*i*) Assume the network does not have feedback connections. Then, the network has *echo states*, if the network state $x(n)$ is uniquely determined by any left-infinite input sequence $\bar{u}^{-\infty}$.

(*ii*) Assume the network has feedback connections. Then, the network has *echo states*, if the network state $x(n)$ is uniquely determined by any pair of left-infinite input sequence $\bar{u}^{-\infty}$ and left-infinite output sequence $\bar{y}^{-\infty}$.

Remark 2.2.2. This definition can be equivalently stated as follows.

(*i*) Assume the network does not have feedback connections. Then, there exist *echo functions* $E = (e_1, e_2, ..., e_N)$, where $e_i : (U^{\text{in}})^{-\mathbb{N}} \to \mathbb{R}$, such that for all left-infinite input histories $\{..., u(n-1), u(n)\} \in (U^{\text{in}})^{-\mathbb{N}}$ the current network state is

$$x(n) = E(..., u(n-1), u(n)).$$

That is,

$$x_i(n) = e_i(..., u(n-1), u(n)), \quad \text{for all } 1 \leq i \leq N.$$

(*ii*) Assume the network has feedback connections. Then, there exist *echo functions* $E = (e_1, e_2, ..., e_N)$, where $e_i : (U^{\text{in}})^{-\mathbb{N}} \times (U^{\text{out}})^{-\mathbb{N}} \to \mathbb{R}$, such that for all left-infinite input and output histories $\{..., u(n-1), u(n)\} \times \{..., y(n-1), y(n)\} \in (U^{\text{in}})^{-\mathbb{N}} \times (U^{\text{out}})^{-\mathbb{N}}$ the current network state is

$$x(n) = E(..., u(n-1), u(n); .., y(n-1), y(n)).$$

That is,

$$x_i(n) = e_i(..., u(n-1), u(n); ..., y(n-1), y(n)), \quad \text{for all } 1 \leq i \leq N.$$

Therefore, the echo state property assumes the internal state to be an "echo" of the past. That is the reason why this concept is named *echo states*.

We will say that a network whose topology is as exposed in Section 2.1 is an *echo states network (ESN)* if it satisfies the echo state property stated in Definition 2.2.1.

Let us now give two important properties of echo states networks.

Proposition 2.2.3. *Let us assume an echo states network without feedback connections. In particular, we are supposing standard compactness conditions. Let d denote the Euclidean distance. Then,*

(*i*) *The network satisfies the state forgetting property. That is, for all left-infinite input sequence* $\{..., u(n-1), u(n)\} \in U^{-\mathbb{N}}$ *there exists a null sequence* $(\delta_h)_{h \geq 0}$ *such that for all states* $x, x' \in A$*, for all* $h \geq 0$*, for all input sequence suffixes* $\bar{u}_h := \{u(n-h), ..., u(n)\}$ *it holds that* $d(T(x, \bar{u}_h), T(x', \bar{u}_h)) < \delta_h$.

(*ii*) *The network satisfies the input forgetting property. That is, for all left-infinite input sequences* $\bar{u}^{-\infty}$ *there exists a null sequence* $(\delta_h)_{h \geq 0}$ *such that for all* $h \geq 0$*, for all input sequence suffixes* $\bar{u}_h := \{u(n-h), ..., u(n)\}$*, for all left-infinite input sequences of the form* $\bar{w}^{-\infty}\bar{u}_h$, $\bar{v}^{-\infty}\bar{u}_h$*, for all states x end-compatible with* $\bar{w}^{-\infty}\bar{u}_h$ *and states* x' *end-compatible with* $\bar{v}^{-\infty}\bar{u}_h$*, it holds that* $d(x, x') < \delta_h$.
Observation: *the sequences of the form* $\bar{w}^{-\infty}\bar{u}_h$ *are the concatenation of both sequences. That is,* $\{..., w(n-h-2), w(n-h-1), u(n-h), .., u(n)\}$.

Remarks 2.2.4. In rough outlines, Proposition 2.2.3 says that recent values of the input will have more influence when updating the internal state than older inputs, since the input influence gradually fades out. Therefore, two different inputs that after some time steps are close enough will produce the same sequence of internal states from that point on. This occurs regardless of the initialization of the internal state. That is, if we initialize the hidden layer with two different values but the sequence of inputs is the same, both sequence of internal states will end up converging. Thus the whole dynamics of the reservoir is not determined by its initialization due to the fact that its influence fades out as time goes by.

On the other hand, notice that we have assumed a network with no feedback. It is worth noting that one could concatenate the input and output vectors and regard it as a new input as well as concatenate the input and feedback matrices and regard it as a new input matrix. By doing this change of notation, we get rid of the feedback connections, since they are now included in the input component. Therefore, we could then apply Proposition 2.2.3.

Proposition 2.2.3, says that an echo states network satisfies the input and state forgetting properties. However, the reciprocal is also true. That is, if a network satisfies the input forgetting property or the state forgetting property, then the networks

has echo states. Therefore, these three notions of echo states, state forgetting and input forgetting are equivalent. The proves of this proposition along with the reciprocal ones can be found in Jaeger, 2010.

Up to this point, the natural question would be how one could assert whether a network has echo states. Notice that proving its existence by means of the definition or by means of the input or state forgetting property does not appear to be feasible. The truth is that there is not a complete answer to this question. However, let us now state a result that provides a sufficient condition for echo states and a sufficient condition for the non-existence of the echo states.

Proposition 2.2.5. *Assume a sigmoid network with a tanh activation function. Let d denote the Euclidean distance.*

(*i*) *Let the weight matrix W satisfy* $\sigma_{max} = \Lambda < 1$, *where* σ_{max} *is its largest singular value. Then* $d(T(x,u,y),T(x',u,y)) < \Lambda \cdot d(x,x')$ *for all inputs u and outputs y, for all states* $x, x' \in [-1,1]^N$. *This implies echo states for all inputs u, all outputs y and all states* $x, x' \in [-1,1]^N$.

(*ii*) *Let the weight matrix have a spectral radius* $|\lambda_{max}| > 1$, *where* λ_{max} *is the largest eigenvalue of W in module. Then, the network has an asymptotically unstable null state. This implies that it has no echo states for any input and output sets* $U^{in} \times U^{out}$ *containing* 0 *and internal state set* $A = [-1,1]^N$.

Again, the proof of Proposition 2.2.5 can be found in Jaeger, 2010.

The main downside of this result is how restrictive it actually is. As we are going to see in the experiments performed in the following chapters, it is quite common to define a connection matrix W whose largest eigenvalue is greater than 1. In these cases, we are not regarding a null input for any time step, thus we are not contradicting Proposition 2.2.5.

So far, we have stated the definition of echo state networks along with its properties. Let us now discuss why they are so powerful. By definition, an ESN is a recurrent neural network displaying the structure and the dynamics explained in Section 2.1 such that its internal states at a certain time step are univocally determined by its past along with the input, internal and feedback connections (see Remark 2.2.2). In other words, let us denote by y_{teach} the desired output we want our network to produce. That is, we want our network to predict a value y that is as close as possible to the desired value y_{teach}. Assume we have available the input values $\{u(1),...,u(T+1)\}$ and the desired output values $\{y_{\text{teach}}(1),...,y_{\text{teach}}(T)\}$. In addition, suppose that the connection matrices W^{in}, W and W^{back} are deterministically defined. Then, the network ends up with a univocally determined values of the internal state $\{x(n_{\min}+1),...,x(T+1)\}$ regardless of the initialization of the internal state and the first values of the input and the output, as long as the value of T and $n_{\min}$ are large enough. Notice that the input and the internal states may fluctuate at the beginning, but it will not make any difference at the values of $\{x(n_{\min}+1),...,x(T+1)\}$. We should keep in mind the necessity of discarding the first $n_{\min}$ time steps, since they are very influenced by the initial values of u, x and y. Since the values of the internal states are completely determined by the values of the input and the output along with the input, internal and feedback connections, then the input, internal and feedback connections DO NOT NEED TO BE LEARNT. Since $y(n) = \tanh(W^{\text{out}}x(n))$, then it is enough to perform a linear regression in order to solely learn W^{out}. Therefore, notice that the main goal is to end up with a

wider variety of hidden states trajectories in order to obtain a subspace containing y_{teach}. Thus, the values of W^{in}, W and W^{back} need to be tuned in order to get the appropriate subspace. Therefore, W^{in}, W and W^{back} are not parameters to be learnt but parameters to be tuned. In Section 2.4, we are going to analyze different choices of all hyperparameters, including such matrix connections.

As we have just mentioned, obtaining a wider variety of states' trajectories is key in order to appropriately learn the desired dynamics. In other words, the trajectories of the internal states have to be a *reservoir of dynamics*. That is the reason why the graph given by W is usually called *reservoir*. Thus this learning technique is called *reservoir computing*. It can be regarded as a particular case of RNN with the difference that the only set of connections that need to be learnt are the output ones, which are learnt by means of a linear regression. Notice that this approach is much cheaper computationally than usual RNN: the number of connections to be learnt is much lower and the learning algorithm is a linear regression instead of the usual backpropagation, which avoids problems as the vanishing/exploding gradient.

2.3 Training the neural network

In this section, we are going to provide the formulas for training and testing the network. In addition, we are going to develop the metrics used to evaluate the performance of the network.

In equations (2.1) and (2.2), we introduced the formulas determining the dynamics of the network. However, such formulas are going to be slightly modified at training time. The training is going to be carried out in two different steps, which are developed next.

(i) Let T be the number of training data points. Then, the internal state is going to be updated for the T training time steps following a teacher forcing approach:

$$x(n+1) = \tanh\left(W^{\text{in}}u(n+1) + Wx(n) + W^{\text{back}}y_{\text{teach}}(n)\right), \tag{2.3}$$

for all $0 \leq n \leq T-1$, where $y_{teach}(n)$ is the desired output at time step n and $x(0)$ is going to be the initial state of the reservoir. Recall that the latter is not relevant due to the state forgetting property (see Proposition 2.2.3). Notice that, once this step is finished, we end up with N time series of size T, one for each internal node. Therefore, x may be thought of as a matrix of size $T \times N$, where the j-th column stands for the trajectory of the j-th node.

Let us remark that, at this step, we only deal with the known data. In other words, we do not make predictions nor use the matrix W^{out}. That is precisely what we mean by *teacher forcing*: instead of making predictions and using them (y), we directly work with the ground truth (y_{teach}) in order to force the desired behavior.

(ii) As shown in equation (2.2), the output is computed as a linear combination of the states of the reservoir. Therefore, the output matrix W^{out} is going to be the one that minimizes

$$||W^{\text{out}}x - \tanh^{-1}(y_{\text{teach}})||_2, \tag{2.4}$$

where x are the internal states computed in the previous step after the first n_{min} time steps were dismissed. *Notice that a transient of n_{min} time steps should*

be discarded in order to guarantee the converge of the reservoir's dynamics, since the first values are quite influenced by the initial values of the input, output and hidden states (see Section 2.2). Equivalently, the loss function to minimize is going to be the *mean square error*:

$$\begin{aligned} \mathrm{MSE}(y, y_{\mathrm{teach}}) &= \frac{1}{T - n_{\min}} \sum_{n=n_{\min}}^{T} \left[\tanh^{-1}(y_{\mathrm{teach}}(n)) - \tanh^{-1}(y(n)) \right]^2 \\ &= \frac{1}{T - n_{\min}} \sum_{n=n_{\min}}^{T} \left[\tanh^{-1}\left(y_{\mathrm{teach}}(n)\right) - W^{\mathrm{out}} x(n) \right]^2 . \end{aligned}$$

The second step of the training implies that the learning procedure consists on performing a linear regression. Therefore, the main goal of the first step is to get a wide enough variety of nodes' trajectories in order to construct a subspace containing the desired output.

Depending on the specific task we are dealing with, we may slightly change equations (2.3) and (2.4). For instance, we may add a noise term in the former in order to prevent the predictions from degenerating. This would actually be the case when dealing with periodic or wildly chaotic dynamics. On the other hand, it may happen that the subspace generated by the states of the reservoir do not generate a subspace containing the desired output. If that is the case, it may be useful to regard a quadratic relationship between y and the internal states x. Therefore, equation (2.4) would become

$$||W^{\mathrm{out}}[x, x^2] - \tanh^{-1}(y_{\mathrm{teach}})||_2,$$

where $[x, x^2]$ would stand for the horizontal concatenation of x and x^2 obtained in the first step (with the first $n_{\min}$ steps discarded) and $W^{\mathrm{out}} \in \mathcal{M}(\mathbb{R})_{L\times(N+N)}$. If that was the case, then equation (2.2) should become

$$y(n+1) = \tanh(W^{\mathrm{out}}[x(n+1), x(n+1)^2]).$$

Actually, when dealing with the standard map, we will implement a slightly different version of this quadratic relationship. Finally, it is also possible to add some regularization term to the loss function. It is quite common to implement a ridge regression.

Once the network has been trained, that is, once the output weights have been learnt, then the network can freely run. That is, it can autonomously predict the output. In order to obtain the prediction at time step $n+1$, with $n+1 > T$, two steps have to be carried out:

(i) Update the internal state of the reservoir by means of equation (2.1). Notice that, in order to compute $x(n+1)$, $x(n)$ is required. Therefore, if we regard $n+1 = T+1$, then $x(n) = x(T)$ would be the the last hidden state obtained during training.

(ii) Compute the prediction $y(n+1)$ by means of equation (2.2).

In order to assess the accuracy of the predictions, we will not only implement the *mean square error* but also the so called *normalized root mean square error (NRMSE) for a certain horizon*, which provides information about the long-term predictions. A very common choice is an horizon of 84 time steps, denoted as NRMSE_{84}. This metrics is computed as follows:

- The reservoir is updated by means of teacher forcing for 1000 time steps (equation (2.3)).
- The network freely runs for another 84 time steps and the value $y(1000+84)$ is kept (equations (2.1) and (2.2)).
- The two previous steps are performed 50 times. Therefore, we end up with 50 values: $\{y(n\cdot 1000+n\cdot 84)|n=1,2,...,50\}$.
- Compute the NRMSE_{84} as follows:

$$\text{NRMSE}_{84}=\left(\frac{\sum_{n=1}^{50}\left[y(n\cdot 1000+n\cdot 84)-y_{\text{teach}}(n\cdot 1000+n\cdot 84)\right]^2}{50\,\sigma^2}\right)^{\frac{1}{2}},$$

  where σ^2 is the variance of the original signal. Recall that σ^2 can be computed from the original values as follows

$$\sigma^2=\text{E}\left[(y_{\text{teach}})^2\right]-\text{E}\left[y_{\text{teach}}\right]^2,$$

  where the expectation is empirically approximated by the mean.

The NRMSE_{84} is very useful in order to measure the accuracy of the predictions of the trained network when dealing with chaotic dynamics. The reason is the fact that this metrics provides information regarding how close are the learnt an the target dynamics for different levels of chaos, since the degree of chaos may vary along time.

2.4 Toy problem

In this section, we are going to develop an example in order to see all the features and procedures discussed above. All the experiments carried out can be found in the notebook *Intuition_EchoStates.ipynb*.

Let us assume that we have a neural network displaying the structure exposed in Section 2.1, but without feedback connections. That is, there are input, internal and output connections, but not feedback. Let us also suppose that such network inputs, at each time step $n\geq 1$, the value

$$u(n)=\sin\left(\frac{n}{5}\right).$$

For the moment, just forget about the output, since it does not influence the dynamics of the states of the reservoir. In other words, we are not actually defining any particular task. In this case, the update is performed as follows

$$x(n)=\tanh(W^{\text{in}}u(n)+Wx(n-1)).$$

Let us now establish the size of the reservoir (i.e, of the hidden layer) and the connection values given by W^{in} and W. As we have mentioned in Section 2.2, it is important to end up with a rich reservoir of dynamics. Therefore, it is a common practice to take a quite large reservoir. In our case, we will take $N=100$. However, this value has to be chosen according to the particular task we are trying to solve. Recall that the reservoir's goal is to end up with a set of N trajectories capable of generating a

subspace containing the desired output. In addition, the values of W^{in} and W are chosen with the same purpose. W is usually a very sparse matrix whose non-null values are randomly chosen to be $-a$ or a with the same probability, where a is usually a small value. The particular value of a is not actually important since, once the reservoir is constructed, its adjacency matrix W is rescaled in order to get the desired value of $\lambda_{\max}$ or $\sigma_{\max}$. What is actually important is to define the non-null values, which should represent around 2% of the total connections, to be opposite in sign. These features of W encourages a wider range of states trajectories. In our particular case, we will randomly construct W by taking values 0, -0.4 and 0.4 with probabilities 0.95, 0.025 and 0.025 respectively. Figure 2.3 shows the trajectories of three randomly selected nodes for two different choices of W. We can see that sparsity actually encourages different behaviors on the states' trajectories. In this case, we say that the reservoir is *inhomogeneous*, since there is a wide variety of trajectories. See Section 2.2 of the notebook called *Intuition_EchoStates.ipynb* in order to see more behaviors regarding different choices of W. Finally, W^{in} is constructed in a similar way but without the sparse feature. In our case, we randomly construct it by taking values 1 or -1 each with probability 0.5.

Let us now check that the input and state forgetting properties actually hold. On the one hand, we are going to initialize the hidden state in three different ways: all nodes equal to 0, all nodes equal to 1 and randomly following a uniform distribution on the interval [-1,1]. Figure 2.4 (A) shows the three different trajectories of three randomly selected nodes. There, we can see that the three trajectories converge after a few time steps. Therefore, the state forgetting property is satisfied. On the other hand, let us initialize the reservoir with the null state and introduce two types of input: the original one and the original one with some random noise added in the first time steps. Figure 2.4 (B) shows the two different trajectories of three randomly selected nodes. Again, we see that, after few time steps, both trajectories converge to the same trajectory. Therefore, the input forgetting property is satisfied.

Another interesting feature that is worth noting, is the influence of the input in a network without feedback connections. Figure 2.5 shows the behavior of three randomly selected nodes for different types of input. By looking at them, we may infer that the dynamics of the reservoir are quite influenced by the dynamics of the input. Notice that if the network also had feedback connections, then the previous outputs will also play an important role on determining the reservoir's behavior.

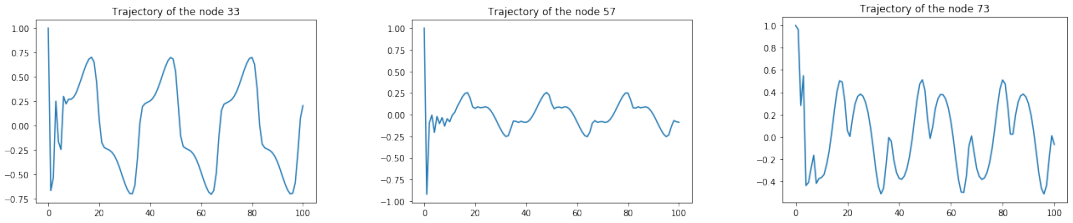
So far, we have defined the network and the input and we have analyzed the relationship between these parts of the network. Let us now determine a particular learning task. In what remains of this section, we are going to train the already defined network in order to predict the sequence

$$y_{\text{teach}}(n) = \frac{1}{2}\sin^7\left(\frac{n}{5}\right)$$

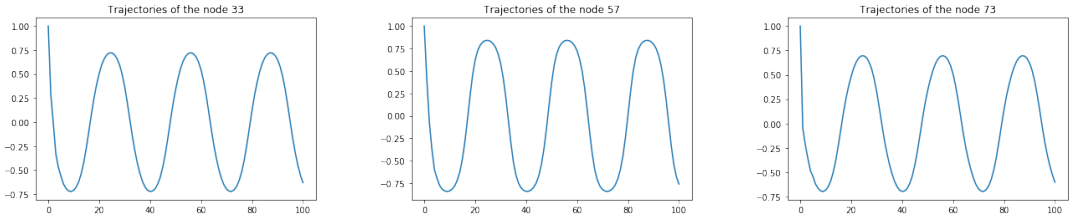
given the original input

$$u(n) = \sin\left(\frac{n}{5}\right).$$

Let us now recall that the notation $y(n)$ stands for the output produced by the network at time n whereas $y_{\text{teach}}(n)$ is the desired output at time n. Keep in mind that our available data consists, on the one hand, on the input sequence $\{u(1), u(2), ...\}$

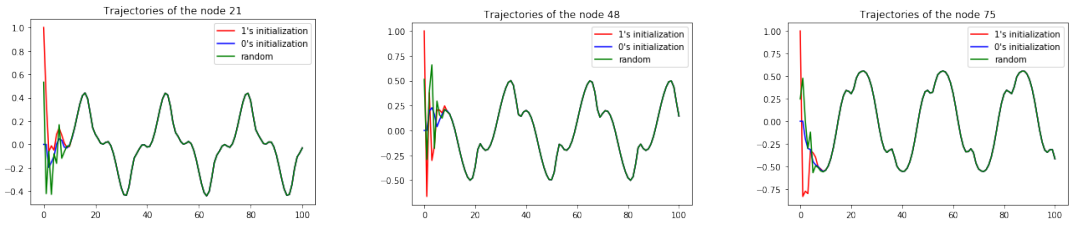


(A)

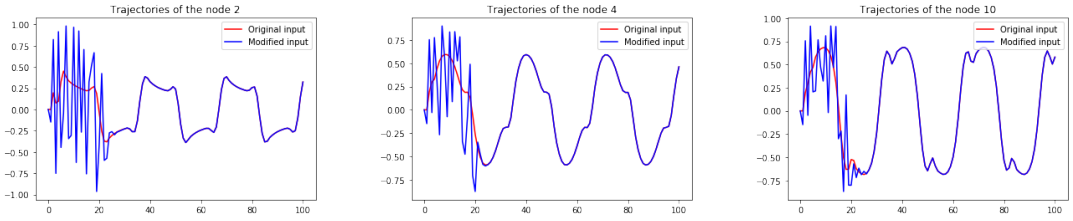


(B)

FIGURE 2.3: **(A)** Trajectories of three randomly selected nodes considering W to be randomly generated by taking values 0, -0.4 and 0.4 with probabilities 0.95, 0.025 and 0.025 respectively. **(B)** Trajectories of three randomly selected nodes where the values of the connections W where uniformly sampled from the interval [-1/12,1/12].

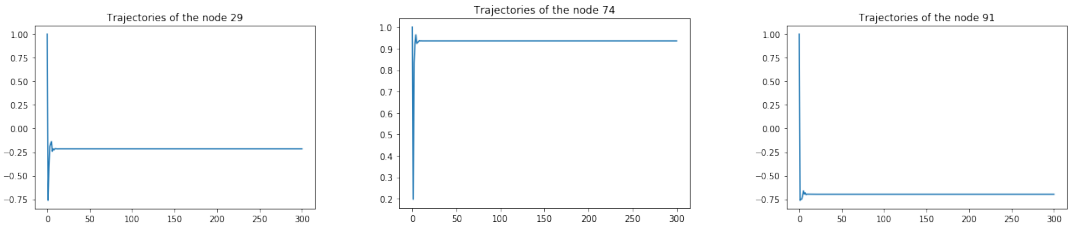


(A)

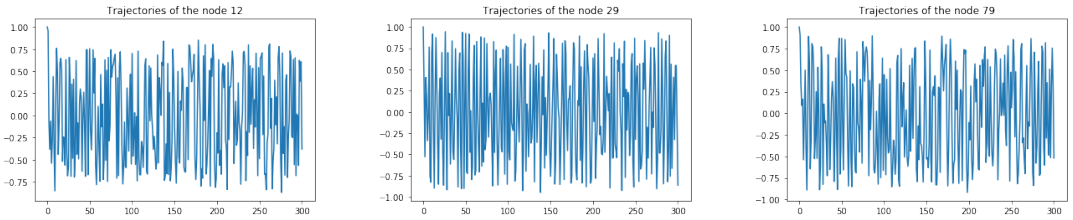


(B)

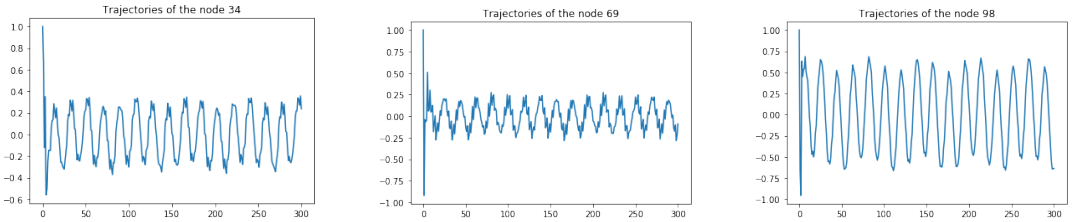
FIGURE 2.4: Trajectories of three randomly selected nodes. **(A)** Each plot shows three different trajectories corresponding to three different initializations: all 1's, all 0's and random. **(B)** Each plot shows two different trajectories corresponding to two different inputs. The *modified input* is the same as the *original input* but with some noise added in the first 20 time steps.



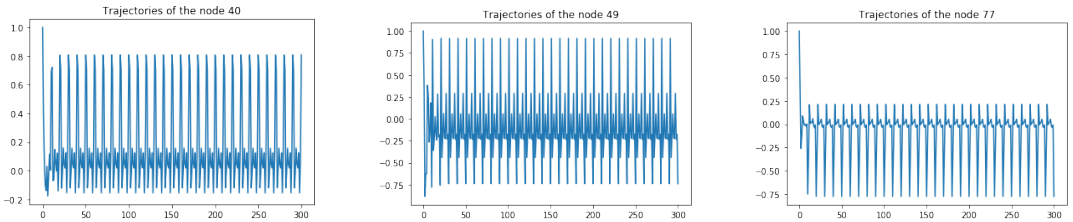
(A)



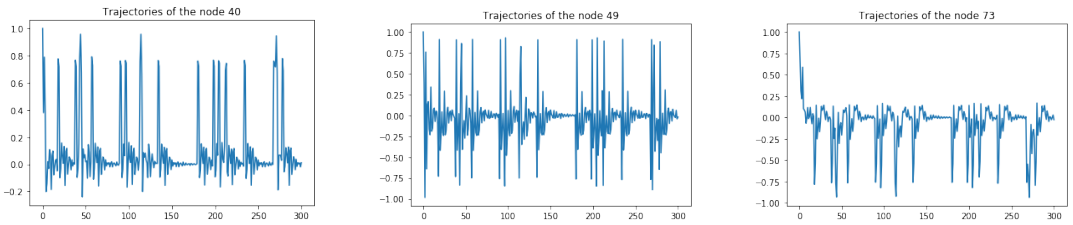
(B)



(C)



(D)



(E)

FIGURE 2.5: Trajectories of three randomly selected nodes for different types of inputs. **(A)** The input was constant. **(B)** Random input. **(C)** The input is a combination of sines and cosines. **(D)** Constant input that periodically spikes. **(E)** Constant input that spikes at random with probability 0.066.

and, on the other hand, on a few values of y_{teach}, $\{y_{\text{teach}}(1), ..., y_{\text{teach}}(T)\}$, in order to perform the supervised learning. Recall that T stands for the number of training time steps.

Let us now train the network for $T = 300$ time steps regarding a dismissal of $n_{\text{min}} = 100$ time steps. As already explained in Section 2.3, the training is performed in two steps:

(i) Compute the trajectories of the reservoir by means of teacher forcing. That is, for $1 \leq n \leq T$, update the states as follows

$$x(n) = \tanh(W^{\text{in}}u(n) + Wx(n-1)).$$

Notice that in this particular case, no teacher forcing is needed since there are not feedback connections. Recall that in case there were feedback connections, the update would be carried out as:

$$x(n) = \tanh(W^{\text{in}}u(n) + Wx(n-1) + W^{\text{back}}y_{\text{teach}}(n)).$$

(ii) Compute W^{out} by minimizing

$$\frac{1}{T - n_{\text{min}}} \sum_{n=n_{\text{min}}}^{T} \left[\tanh^{-1}(y_{\text{teach}}(n)) - W^{\text{out}}x(n)\right]^2.$$

After performing the training, we obtained a training MSE of $3.19 \cdot 10^{-24}$. Once the network is trained, it can generate the predictions. Recall that, in order to obtain the prediction at a single time step n, two steps are required:

(i) Update the state of the reservoir:

$$x(n) = \tanh(W^{\text{in}}u(n) + Wx(n-1))$$

Notice that this formula was also used for training. The reason is the fact that there is no feedback connections. Recall that, if there were feed back, the update would be performed as follows

$$x(n) = \tanh(W^{\text{in}}u(n) + Wx(n-1) + W^{\text{back}}y(n)).$$

(ii) Compute the output at time n:

$$y(n) = \tanh(W^{\text{out}}x(n)).$$

We tested the network for $301 \leq n \leq 600$ and got a testing MSE equal to $2.8 \cdot 10^{-4}$. Figure 2.6 shows the predictions against the true values. By looking at that picture, we see that the network has properly learnt the desired dynamics.

To finish, let us mention that the learning has been possible because the output and the input are quite similar. As we have already seen above, the dynamics of the reservoir is quite determined by the dynamics of the input. Consequently, if we tried to learn an output from a very different input, the learning would not have been successfully. Let us illustrate this fact by means of an example. Assume we

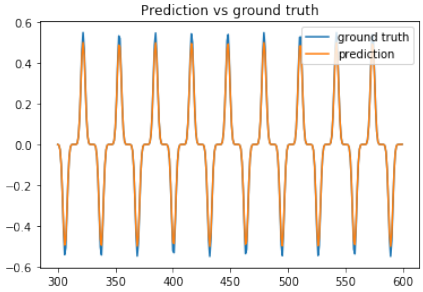


FIGURE 2.6: Comparison between the desired output and the network's prediction.

want to learn

$$y_{\text{teach}}(n) = \frac{1}{15}\left|\sin\left(\frac{n}{10}\right) + 7\sin\left(\frac{n}{3}\right) + \sin(3n)\cos(n)\right| + \frac{1}{\pi}\cos\left(\frac{n}{5}\right)$$

from the input

$$u(n) = \frac{1}{20}\left(\sin\left(\frac{n}{10}\right) + 7\sin\left(\frac{n}{3}\right) + \sin(3n)\cos(n)\right).$$

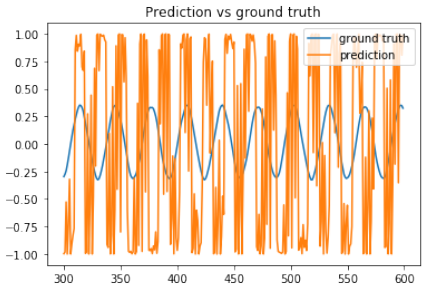


FIGURE 2.7: Comparison between the desired output and the network's prediction when the former behaved very differently with respect to the input.

Since the network only has input connections, the behavior of the reservoir is going to be highly influenced by the input. Since the output has nothing to do with the input, then it will not be contained in the subspace generated by the reservoir. When performing this experiment, we got a training error of 0.04, which leads us to think that the network is not properly learning. Finally, we got a testing MSE of 0.74. Figure 2.7 shows the comparison between the predictions and the desired output. It is clear that they have nothing to do. Let us remark again that the learning has not been possible since the dynamics of the reservoir were highly determined by the input, which in turn is very different from the output. Therefore, the output was not contained in the reservoir's subspace.

Chapter 3

Learning two fundamental dynamics

Along this chapter, we aim to train a neural network in order to learn two fundamental dynamics, such are the discrete periodic dynamics and the multiple attractor dynamics. The former is going to be developed in Section 3.1 and the latter in Section 3.2. In these approaches, we are going to train and test the network by means of the already introduced techniques (see Section 2.3). However, when dealing with the periodic dynamics, we are going to present an approach for preventing the network from degenerate: the insertion of noise.

All the experiments performed in this section can be found on the notebook *Periodic_Spiking_Dynamics.ipynb*.

3.1 Periodic dynamics

The main goal of this section is to train the neural network in order to make it capable of generating a discrete periodic sequence. Equivalently, this task can be thought of as learning how to cyclically play a melody. In order to achieve so, the network needs to learn how to cycle through a periodic attractor.

3.1.1 Setting the task

The periodic sequence we want the network to learn is a periodic repetition of the melody "House of the Rising Sun", which consists of 48 notes. Therefore, the signal has period 48. The notes of this melody are assigned numerical values ranging from -1 (g#) to 14 (a'), with halftone intervals corresponding to unit increments. This melody is depicted in Figure 3.1. Finally, since we are using *tanh* as the activation function, the outputs that the network is going to generate will belong to the interval [-1,1]. Therefore, in order to learn this melody, we need to squash it into such interval. In order to achieve so, the notes' numerical values are divided over 28.

As already stated above, in this section we are going to understand the learning task as being able of autonomously reproducing this sequence of period 48 during the maximum possible number of time steps. Let us now state this learning procedure more formally. The desired output, y_{teach}, is going to be the periodic concatenation of the melody. That is, at time step n, $y_{\text{teach}}(n)$ is going to be the note n%48 of the melody, where % stands for the modulo. Then, at testing time n, we want the hidden layer to input the output at time $n-1$ through the feedback connections and output $y(n)$, which should be as close as possible to $y_{\text{teach}}(n)$. Therefore, we are going to consider a neural network (following the topology of Section 2.1) without

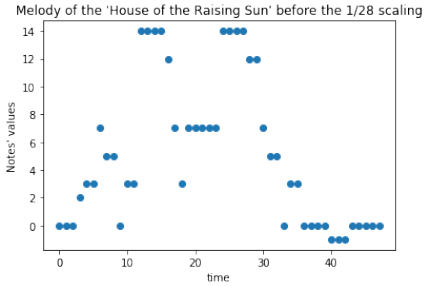


FIGURE 3.1: One period of the sequence to be learnt before squashing it into the interval [-1,1].

input but with feedback connections.

The next two sections provide more details regarding the network and the training and testing procedures.

3.1.2 Neural Network

As already mentioned in Section 3.1.1, the network we are going to implement do not have input but feedback connections, due to the task we want to solve. Let us now determine the reservoir and feedback connections. Recall that, as stated in Section 2.2, the main goal of these connections is to end up with a wider enough variety of hidden states trajectories in order to obtain a subspace containing y_{teach}.

On the one hand, the reservoir is going to be composed of 400 nodes. Its internal weights will be randomly chosen to be 0, 0.4 and -0.4 with probabilities of 0.9875, 0.00625 and 0.00625 respectively. Recall that sparsity encourages more different behaviors on the trajectories. In addition, 400 nodes appears to be enough so as to produce a rich reservoir of dynamics. On the other hand, the feedback weights are going to be randomly chosen following a uniform distribution on the interval [-2,2]. In Section 3.1.5, we are going to modify the reservoir and feedback connections in order to see how the predictions vary. By doing so, we will try to empirically identify the role they play on the learning procedure. However, we are going to see that the feedback choice is quite robust.

Finally, let us mention that the maximum eigenvalue and maximum singular value (in module) of the adjacency matrix of the reservoir, which is the matrix of internal connections W, are 0.93 and 1.96 respectively. Therefore, this shows that Proposition 2.2.5 is actually very restrictive.

3.1.3 Training and testing

As already stated in Section 3.1.1, the main goal is to train a network that is capable of autonomously generating a cyclic repetition of the melody. That is, we want y_{teach} to be the periodically concatenation of the melody.

The training was carried out for $T = 1500$ time steps taking a dismissal of $n_{\text{min}} = 500$. That is, we updated the reservoir by means of teacher forcing for $1 \leq n \leq T$ as follows

$$x(n) = \tanh(Wx(n-1) + W^{\text{back}} y_{\text{teach}}(n-1)).$$

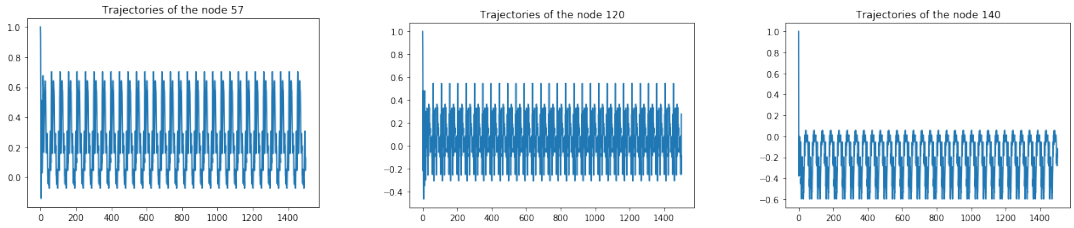


FIGURE 3.2: Trajectories of three randomly selected nodes during training.

Then, the output connections W^{out} are found by minimizing the square error:

$$MSE(y, y_{\text{teach}}) = \frac{1}{T - n_{\text{min}}} \sum_{n=n_{\text{min}}}^{T} \left[\tanh^{-1}(y_{\text{teach}}(n)) - W^{\text{out}} x(n) \right]^2,$$

Recall that this formulas were developed in Section 2.3.

In our particular task, we got a training MSE of $1.90 \cdot 10^{-28}$. Figure 3.2 illustrates the trajectories of three randomly selected nodes during training.

Regarding the testing, recall that, at time step n, the prediction is computed in two steps:

- Update the reservoir as follows:

$$x(n) = \tanh(Wx(n-1) + W^{\text{back}} y(n-1)),$$

  where $y(n-1)$ is the prediction of the network at time step $n-1$.

- Compute the prediction as

$$y(n) = \tanh(W^{\text{out}} x(n)).$$

As sated in Section 2.3, the test begins when the training ends. That is, the testing time starts at $T+1$ and regards $x(T)$ as the last internal state obtained when training and $y(T)$ as $y_{\text{teach}}(T)$. However, since our goal is not predicting the following time steps but being able to autonomously generate the melody, the test is going to be initialized in a slightly different way. Since the network is capable of autonomously running once the reservoir has converged (due to the input and state forgetting property), we will erase all the nodes' trajectories obtained during training. Then, we will perform some time steps by means of teacher forcing (as during training), which are going to be denoted as t_{dismiss}. Then, the test is going to be carried out for $n > t_{\text{dismiss}}$ and the value of $x(t_{\text{dismiss}} - 1)$ is going to be the last hidden state obtained during this procedure and $y(t_{\text{dismiss}} - 1)$ will be $y_{\text{teach}}(t_{\text{dismiss}} - 1)$.

We tested our network for 300 time steps after performing a teacher forcing procedure of $t_{\text{dismiss}} = 500$. Figure 3.3 illustrates the predictions during test, whose MSE was of 0.16. By looking at that image, we can see that, once the network has autonomously run for almost 125 time steps, it begins to degenerate. In other words, the periodic orbits at which the network converge are unstable. The reason for such

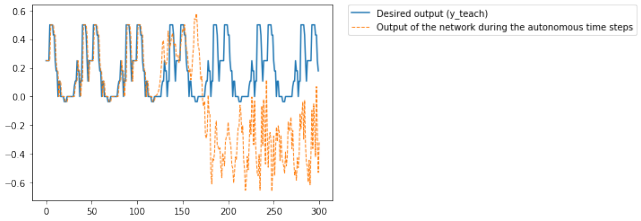


FIGURE 3.3: Testing the trained network for 300 time steps after considering $t_{\text{dismiss}} = 500$.

behaviour is because the equation we minimize when training, which is

$$\text{MSE}_{\text{train}} = \frac{1}{1000} \sum_{n=501}^{1500} \left(\tanh^{-1} y_{\text{teach}}(n) - W^{\text{out}} x(n) \right)^2,$$

is equivalent to

$$\text{MSE}_{\text{train}} = \frac{1}{48} \sum_{n=501}^{548} \left(\tanh^{-1} y_{\text{teach}}(n) - W^{\text{out}} x(n) \right)^2,$$

since y_{teach} and x are 48-periodic. Therefore, in order to determine the 400 values that compose W^{out}, we only dispose of 48 linearly independent arguments. Thus there are many possible values of W^{out} that would minimize the above equation. Then, when doing the python instruction ".fit", one of them is arbitrarily chosen. Section 3.1.4 will provide a technique for avoiding such degeneration.

3.1.4 Preventing the network from degenerating

As stated in Section 3.1.3, the degeneration of the network's behavior is due to the fact that the periodicity of y_{teach} and x render the computation of W^{out} underdetermined when performing the MSE. One way of avoiding such underdetermination is by introducing some noise when updating the state of the reservoir during training. By doing so, we will prevent x from being periodic.

Therefore, the updating is going to be carried out as follows

$$x(n) = \tanh \left[W x(n-1) + W^{\text{back}} (y_{\text{teach}}(n-1) + \nu(n)) \right],$$

where the noise $\nu(n)$ is uniformly sampled from [-0.001, 0.001]. Notice that noise is added to the y_{teach}, since the problem is caused by its periodicity. Now, the equation we want to minimize, which is

$$\text{MSE}_{\text{train}} = \frac{1}{1000} \sum_{n=501}^{1500} \left(\tanh^{-1} y_{\text{teach}}(n) - W^{\text{out}} x(n) \right)^2,$$

is composed of 1000 linearly independent arguments since now we actually have 1000 values of x.

Following this new equation, we obtained a training MSE of $6.86 \cdot 10^{-8}$, which is much lower than the one we obtained when no noise was introduced, which was of the order of 10^{-28}. Figure 3.4 shows the trajectories of three randomly selected nodes during training. Notice that they are quite similar to the ones we got when

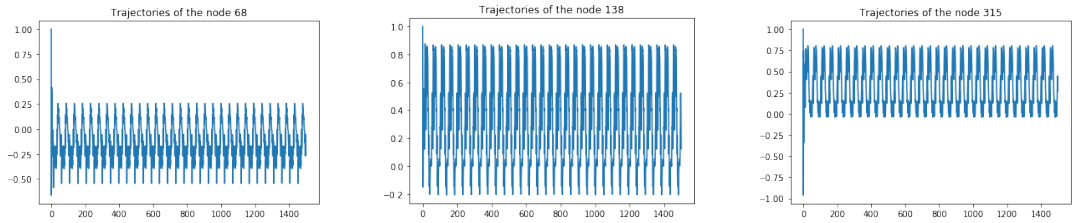


FIGURE 3.4: Trajectories of three randomly selected nodes during training. Some noise was introduced during training.

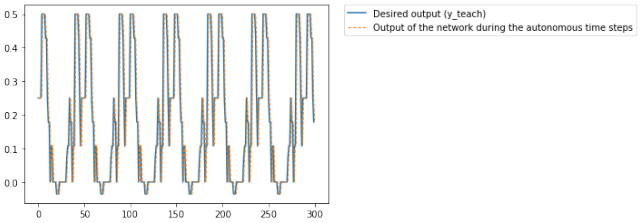


FIGURE 3.5: Testing the trained network for 300 time steps after considering $t_{\text{dismiss}} = 500$. Some noise was introduced during training.

no noise was introduced. Then, we tested the network for different values of t_{dismiss} and different testing times. The results are stated below:

- When testing the network for 300 time steps after regarding $t_{\text{dismiss}} = 500$, we got a MSE of $8.82 \cdot 10^{-8}$. Figure 3.5 shows the predictions, which coincides with the desired output. Notice that this was the experiment we performed in Section 3.1.3, but it now does not degenerate.
- When testing the network for 10000 time steps after regarding $t_{\text{dismiss}} = 500$, we got a MSE of $9.09 \cdot 10^{-8}$.
- When testing the network for 10000 time steps after regarding $t_{\text{dismiss}} = 30$, we got a MSE of $1.01 \cdot 10^{-7}$.

Notice that the result of the last experiment is very powerful. It says that, once the output weights have been learnt, then the network only needs to converge for the first 30 time steps and, afterwards, it can autonomously generate copies of the melody for 10000 time steps giving a MSE of $1.01 \cdot 10^{-7}$, which is very low.

The details regarding the training and testing of this task are already explained in Section 3.1.3.

3.1.5 Playing with the deterministic parameters

In Section 3.1.2, we determined the feedback and internal connections. Let us now see how the learning procedure is modified when changing these values. All these experiments and some more are developed in Section 1.3, 1.4 and 1.5 of the Notebook *Periodic_Spiking_Dynamics.ipynb*. In this section, we will summarize the results seen there.

Let us begin by modifying the feedback connections. Recall that the construction used in the above experiments regarded it to be randomly generated by uniformly taking values on [-2,2]. Then, we performed some experiments regarding W^{back} to be uniformly sampled from: [-0.5,0.5], [-0.01,0.01], [-4,4], [-6,6], {-1,1} and {-3,3}. The

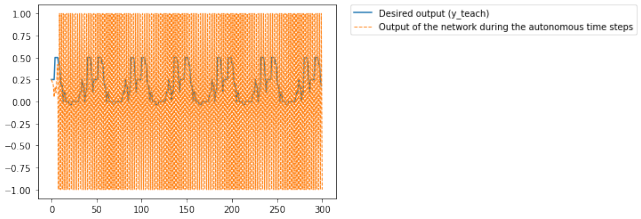


FIGURE 3.6: Testing the trained network for 300 time steps after considering $t_{\text{dismiss}} = 500$. All the components of the adjacency matrix of the reservoir takes the same non-null value.

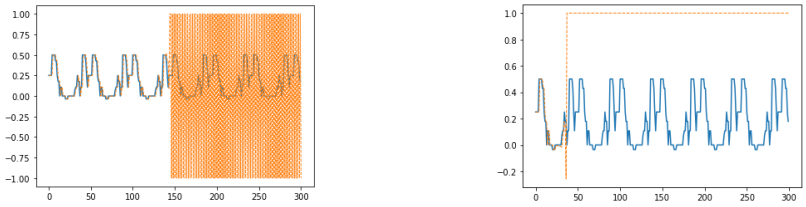
results obtained suggest that when we test the network for 300 time steps after regarding $t_{\text{dismiss}} = 500$, the network behaves quite robustly independently of the particular W^{back}. In addition, we see that W^{back} can be uniformly chosen from [-a,a] for $a \in [0.5, 4]$ without much influence on the network's behaviour. Finally, it seems that, for smaller values of a, the network needs to undergo the teacher forcing step (i.e, t_{dismiss}) for longer time in order to achieve good performance.

The second bunch of deterministic connections are the internal ones. As we already mentioned along the project, the reservoir needs to be *inhomogeneous*. That is, the nodes of the reservoir need to display trajectories quite different among them. As mentioned many times, this is achieved by regarding a sparse reservoir. In fact, we have performed various experiments where the network have been trained regarding different levels of sparsity. The cases we considered are: fully-connected with the same weights, fully-connected with different weights and non fully-connected with different weights. In the former case, we tested the network for 300 time steps regarding $t_{\text{dismiss}} = 500$. The comparison of the predictions with the desired output can be seen in Figure 3.6. Since all the trajectories are almost equal, then the network is not capable of learning. On the other hand, the other two experiments gave very similar results than with the sparse matrix but with a higher computational cost. Therefore, we may conclude that the best reservoir is constructed with only 1-2% of connectivity. By doing so, a wide variety of behaviors of the nodes are encouraged, which is key to make it capable of robustly learning the desired output and, at the same time, its low percentage of connectivity entails a lower computational cost.

Finally, we recover the original sparse matrix of connections W. Recall that it was randomly constructed taking values 0, -0.4 and 0.4 with probabilities of 0.9875, 0.00625 and 0.00625 respectively. Then, we wanted to analyze the role played by the numbers themselves. That is, why choosing 0.4 instead of 0.1 or 0.9, for instance. In fact, there is a common practice, as already mentioned in Section 2.4, consisting on rescaling the values of the matrix in order to obtain a particular spectral radius. In this particular case, the maximum eigenvalue (in module) was 0.93. Therefore, we performed different experiments considering different spectral radius.

The truth is that such maximum eigenvalue is quite related with the memory of the network. The closer it is to 1, the slower it is the fading out of the previous information. By looking at the periodic orbit we want to learn, which was depicted in Figure 3.1, we observe that the last 5 notes and the first 3 notes are the same. This implies that the network needs to have enough memory so as to see the same note 8 times in a row and not getting stuck on it. However, few memory is enough in order to achieve so.

We performed different experiments considering an spectral radius equal to 0.99,



(A) (B)

FIGURE 3.7: Testing the network for 300 time steps after $t_{\text{dismiss}} = 500$. The orange points are the predictions and the blue ones are the desired output. **(A)** Spectral radius of 0.15. **(B)** Spectral radius of 0.09.

0.79, 0.27, 0.15 and 0.09. The three former cases provided the same result as with the original spectral radius whereas the two latter cases gave a degenerated result. Figure 3.7 illustrates the predictions in these degenerated cases. Therefore, we infer that it is instrumental to have a little bit of memory.

3.2 Multiple attractor

In this section, we are going to train a network so as to learn a dynamics characterized by having multiple attractors. That is, we want the network to stabilize on certain stable points and, then, we want it to be capable of jumping from one stable point to another stable point.

We are going to consider a 20-dimensional input which is going to take constant value 0 in all its components for all time steps. However, the input is going to spike. That is, it will take value 0.5 every 200 time steps and the spike is going to occur at a single component at a time. So, in the first time step, only the first component is going to spike, while the other components will take value 0; then all components are going to take value 0 for 200 time steps; afterwards, the second component is going to take value 0.5, while the other ones are going to take value 0; and so on. So the spike occurs at one component at a time, in a sorted way and every 200 time steps. The components are also referred to as *channels*. Now, we want the output to provide a description of the input's spikes. The desired output is going to be 20-dimensional and its i-th channel will take value -0.5 until the i-th input channel spikes. Once the i-th input channel has spiked, then the i-th channel of the output will take value 0.5. Then, such channel will remain constant at value 0.5 until another input channel spikes. Once the other input channel spikes, then the i-th channel of the output will come back to the value -0.5.

Let us now construct the network. It will have input and feedback connections. The reservoir is going to consist on 100 nodes randomly connected with weights 0, 0.4 and -0.4 with probabilities 0.95, 0.025 and 0.025 respectively. Finally, these values are going to be rescaled so as to get an spectral radius of 0.22. As we have already mentioned in Section 3.1.5, such value is related with the memory. Since it takes such an small value, then the network will not have much memory. The reason why we want to impose such a short memory is the fact that we want to learn this task from the previous outputs and the inputs. That is, we do not want the network to

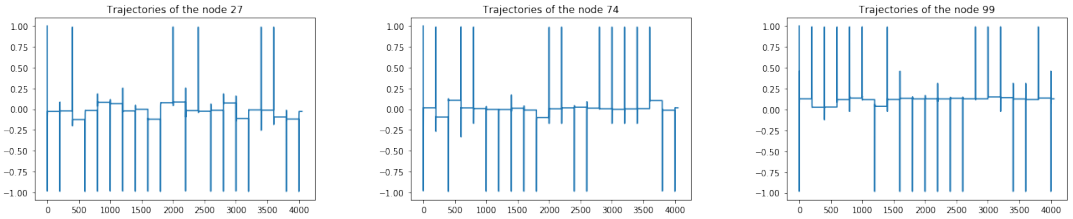


FIGURE 3.8: Trajectories of three randomly selected nodes during training.

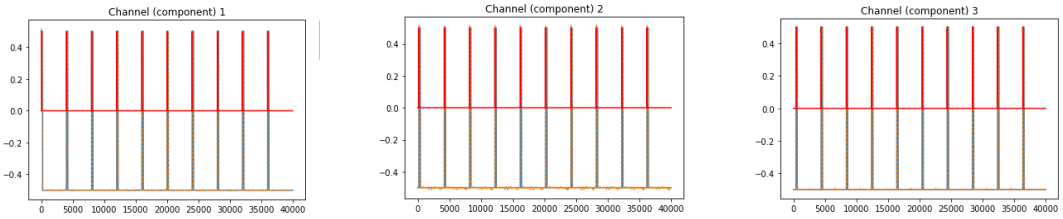


FIGURE 3.9: Input along with the predictions and the ground truth for channels 1, 2 and 3. The orange points are the predictions and the blue ones are the desired output.

memorize anything.

We trained the network for $T = 4050$ time steps and discarded the first $n_{\min} = 50$ time steps. The network was trained as specified in Section 2.3 and we obtained a set of 20 training MSE (one for each component) of the order of 10^{-7} or 10^{-8}. Figure 3.8 shows the trajectories of the three randomly selected nodes during training.

Then, we tested the network for 40000 time steps. The input during testing was constructed in the same way as during training: the spiked occurred every 200 time steps at one channel at a time, in a sorted way. The order of the 20 testing MSE obtained ranged from 10^{-11} to 10^{-8}. Figure 3.9 shows the behavior of the first three channels. In it, we can see the comparison of the desired output against the predicted output along with the input (for that particular component). We can see that the predictions coincide with the ground truth.

Then, we also tested the already trained network with a random input. That is, we considered an input such that, with probability 0.02, one of its channels, randomly selected with a uniform distribution, spiked. Therefore, the spikes were produced at random time steps at random channels. However, we still have the constraint that the spike has to occur at a single channel. We tested the network for 4000

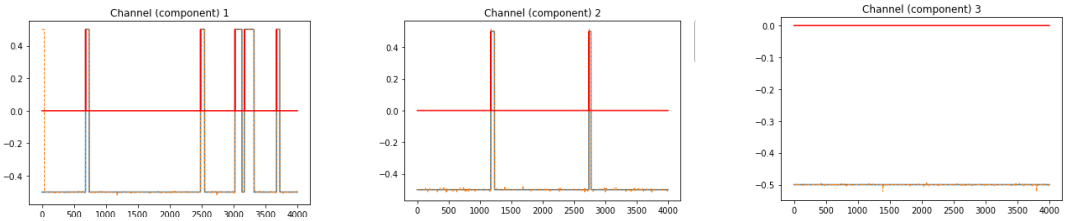


FIGURE 3.10: Input along with the predictions and the ground truth for channels 1, 2 and 3. The input at testing time was different than at training time. The orange points are the predictions and the blue ones are the desired output.

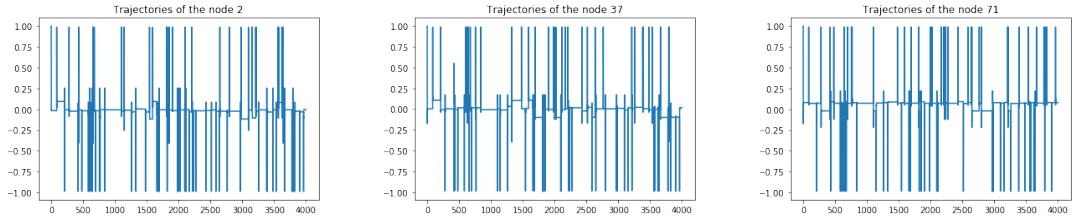


FIGURE 3.11: Trajectories of three randomly selected nodes.

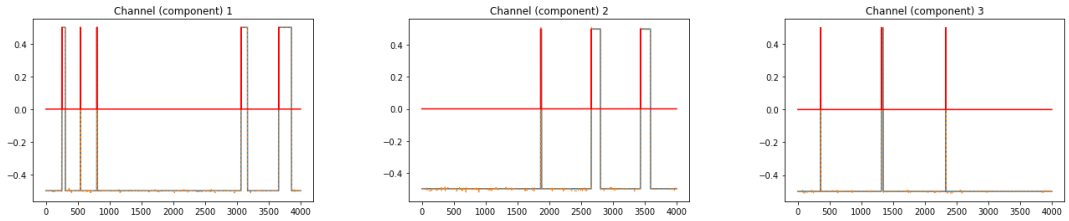


FIGURE 3.12: Input along with the predictions and the ground truth for channels 1, 2 and 3. The orange points are the predictions and the blue ones are the desired output.

time steps. Then, we got 20 testing MSE of the order of 10^{-4}. Figure 3.10 shows the behavior of the first three channels. Since the input spiked at random, the first channel spiked 5 times whereas the third channel never spiked. Notice that the network successfully predicts the desired output. Therefore, this means that the network has not memorized the task but it has actually learnt it. This could be achieved due to the small spectral radius.

Finally, we trained again the network considering the same random input as in the previous test. That is, an input that spiked with probability 0.02 at a random channel. Then, we got 20 training MSE whose orders ranged from 10^{-7} to 10^{-6}. Figure 3.11 shows the trajectories of 3 randomly selected nodes. Notice that it is quite similar to the previous case, but now the space between spikes is different. Then, we tested the network for 4000 time steps. Figure 3.12 shows the comparison of the predictions and the ground truth along with the inputs for the first three channels. Again, notice that the predictions are as desired. Let us remark again that the network has learnt but not memorized the task, which has been possible due to the small spectral radius.

Chapter 4

The Mackey-Glass system

In this chapter, we are going to introduce the so called *Mackey-Glass system*, which is a dynamical system that displays a chaotic behavior. Our main goal will be to train a neural network in order to make it capable of predicting chaotic trajectories. That is, we will train the network with T time steps of a particular trajectory and, afterwards, the network will predict the following time steps.

All the experiments performed in this chapter can be found on the attached notebook called *MackeyGlass.ipynb*. Actually, this notebook also contains some additional experiments, like the training and testing of the network for values of σ between 17 and 30.

4.1 Preliminaries on the Mackey-Glass system

The Mackey-Glass system is determined by the following *delay differential equation*

$$\dot{y}(t) = \frac{\alpha y(t-\sigma)}{1+y(t-\sigma)^{\beta}} - \gamma y(t),$$

where $\alpha, \beta, \gamma > 0$. This system has different behaviors depending on the parameters. For instance, it may have a fixed point, it may oscillate or it may display a chaotic dynamics. Recall that, in rough outlines, chaotic dynamics are characterized by the fact that the trajectories of two arbitrarily close initial points may differ a lot after some time steps. This is just an intuition of what chaos means but in any case it is a definition. See Greiner, 2010 for a more formal definition of chaos.

The parameters we are going to use are the standard ones. That is, $\alpha = 0.2$, $\beta = 10$ and $\gamma = 1$. Therefore, the equation becomes

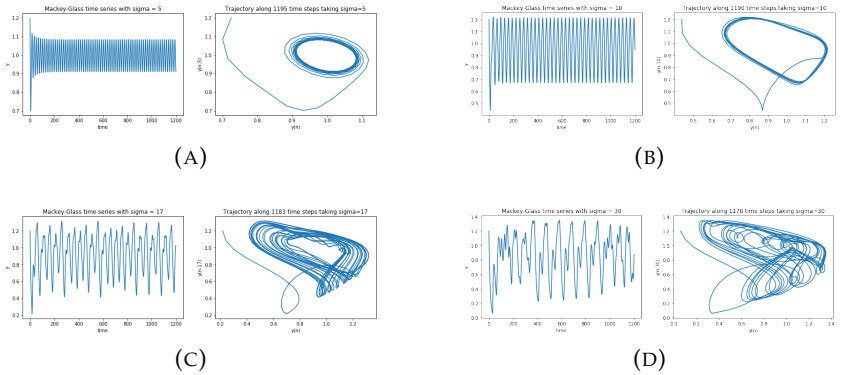
$$\dot{y}(t) = \frac{0.2\, y(t-\sigma)}{1+y(t-\sigma)^{10}} - 0.1\, y(t),$$

where $\sigma > 0$ stands for the time delay. Under such conditions, the system becomes chaotic for $\sigma > 16.8$.

It can be seen that the discrete approximation to the continuous solution of the system is

$$y(n+1) = y(n) + \delta \left[\frac{0.2\, y(n-\frac{\sigma}{\delta})}{1+y(n-\frac{\sigma}{\delta})^{10}} - 0.1\, y(n) \right], \tag{4.1}$$

where $\delta = 1/10$. Finally, in order to achieve that a unit in the discrete approach is the same as a unit in the continuous approach, a subsampling by 10 should be performed. In other words, after the subsampling, one can guarantee that one time step from $y(n)$ to $y(n+1)$ by means of the above formula corresponds to a unit time



(A) (B)

(C) (D)

FIGURE 4.1: Visualization of four different trajectories of the Mackey-Glass system for different values of σ. In all the cases, the initial condition is $y(0) = 1.2$. **(A)** $\sigma = 5$. **(B)** $\sigma = 10$. **(c)** $\sigma = 17$. **(D)** $\sigma = 30$.

step from $y(t)$ to $y(t+1)$ of the original continuous system. In addition, we are going to define $y(n) = 0$ for all $n < 0$.

There are two common ways of visualizing such systems. The first one would be the time series itself, which means y as a function of n. The second one consists in plotting the trajectory followed by any point $(y(n), y(n-\sigma))$.

In Figure 4.1, we can see both types of visualization for different values of the delay: $\sigma = 5, 10, 17, 30$. In all the plots, we are regarding $y(0) = 1.2$. Notice that, as already mentioned above, this system behaves differently for different values of σ and the chaotic behavior appears when $\sigma > 16.8$.

From now on, we will be dealing with a chaotic Mackey-Glass system. That is, we will only regard values of σ greater or equal than 17.

4.2 Setting the task

The main goal would be to train an echo states network so as to learn the dynamics given by the Mackey-Glass system. By "learning the dynamics" we mean training the network for a sequence of a certain size T in such a way that the trained network could afterwards predict the sequence for $t > T$. Therefore, we are going to consider y_{teach} as a certain trajectory of the Mackey-Glass system. In fact, we will perform experiments regarding different initial conditions.

Let us now construct the network. First of all, we will deal with a reservoir composed by 400 nodes. Such nodes are going to be randomly connected taking weights equal to 0, 0.4 and -0.4 with probabilities 0.9875, 0.00625 and 0.00625 respectively. Therefore, we will only regard 1.25% of connections. Finally, such weights are going to be rescaled in order to obtain an adjacency matrix with an spectral radius of 0.79. On the other hand, it is clear that our network needs to have feedback weights in order to get the information from previous outputs. Such connections (W^{back}) are going to be randomly sampled following a uniform distribution over [-0.56,0.56]. In addition, we are going to consider a constant input equal to 0.2, that is, $u(n) = 0.2$ for all n in order to introduce more variability within the internal states. The matrix giving the input connections (W^{in}) will be randomly chosen from the set {0, 0.14, -0.14} with probabilities 0.5, 0.25 and 0.25 respectively. It is quite common to take the

feedback connections within an interval and the input connections within a discrete set.

As already mentioned above, y_{teach} is going to be a trajectory of the Mackey-Glass system, which is constructed as explained in equation (4.1). However, since the activation function is going to be a *tanh*, we want our sequence to take values in the interval [-1,1]. That is the reason why such sequences will undergo the following transformation

$$y_{\text{teach}} \rightarrow \tanh(y_{\text{teach}} - 1).$$

Therefore, unless we specify the contrary, y_{teach} is going to denote such squashed version of the trajectory.

4.3 Training and testing

As stated above, the task consists in learning the dynamics of the Mackey-Glass system. In order to achieve so, different experiments are going to be carried out, each one considering a different type of training sequence:

- $\sigma = 17$ and $T = 3000$ time steps.
- $\sigma = 17$ and $T = 21000$ time steps.
- $\sigma = 30$ and $T = 3000$ time steps.
- $\sigma = 30$ and $T = 21000$ time steps.

Recall that σ is the parameter determining the degree of chaos, being 16.8 the boundary over which the system becomes chaotic. Therefore, $\sigma = 17$ entails a mildly chaotic system whereas $\sigma = 30$ entails a wilder chaotic system.

The training and testing procedures are going to be as exposed in Section 2.3. However, we will introduce another updating formula, which, during the training time, will be as follows:

$$x(n+1) = (1-\delta Ca)x(n) + \delta C\left[\tanh\left(W^{\text{in}}u(n+1) + Wx(n) + W^{\text{back}}y_{\text{teach}}(n) + \nu(n)\right)\right], \tag{4.2}$$

where we are going to take $\delta = 1, C = 0.49$ and $a = 0.9$. The noise $\nu(n)$, which will be uniformly sampled from the interval $[-0.00001, 0.00001]^{400}$, will only be introduced when dealing with $\sigma = 30$, since, as we are going to see, it entails a more stable result. On the other hand, no noise will be required when dealing with $\sigma = 17$.

As always, the learning is carried out by minimizing the MSE, which is

$$\text{MSE} = \frac{1}{T - n_{\text{min}}} \sum_{n=n_{\text{min}}}^{T} \left[\tanh^{-1}\left(y^{\text{teach}}(n)\right) - W^{\text{out}}x(n)\right]^2,$$

where T is the the number of training time steps and n_{min} is the number of discarded time steps. In all the experiments we are going to perform, we will discard the first 1000 time steps.

During testing time, the updating is going to be carried out as shown in equation (4.2) but without noise and using y instead of y_{teach}. The output of the network

Initial Condition	Errors	T=3 000	T=21 000
y(0)=1.2	NRMSE_{84}	0.00109	0.00135
	$\text{MSE}_{\text{test}}(T/1000)$	0.00012	$2.91 \cdot 10^{-5}$
	$\text{MSE}_{\text{test}}(T/5000)$	0.03192	0.06040
y(0)=1	NRMSE_{84}	0.00299	0.00150
	$\text{MSE}_{\text{test}}(T/1000)$	$0.3.34 \cdot 10^{-5}$	$5.21 \cdot 10^{-5}$
	$\text{MSE}_{\text{test}}(T/5000)$	0.08973	0.09480
y(0)=1.5	NRMSE_{84}	0.00104	0.00126
	$\text{MSE}_{\text{test}}(T/1000)$	0.00029	$4.23 \cdot 10^{-6}$
	$\text{MSE}_{\text{test}}(T/5000)$	0.03177	0.03585
y(0)=5	NRMSE_{84}	0.00159	0.00146
	$\text{MSE}_{\text{test}}(T/1000)$	0.00145	0.00014
	$\text{MSE}_{\text{test}}(T/5000)$	0.06083	0.07123

FIGURE 4.2: Performance comparison of two different values of T regarding $\sigma = 17$

will be a linear combination of the internal states of the reservoir. That is, the output will be computed as

$$y(n) = W^{\text{out}} x(n),$$

where $W^{\text{out}} \in M_{1\times N}(\mathbb{R})$ is the learnt matrix.

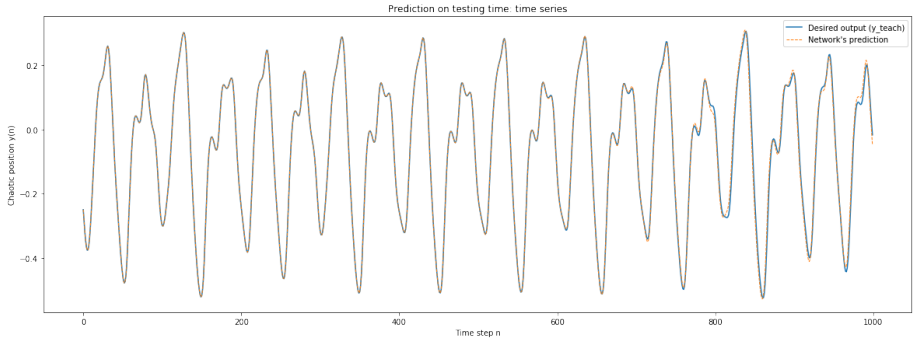
Finally, apart from using the MSE, we are going to use the NRMSE_{84}, which was introduced in Section 2.3. It measures the accuracy of the predictions in the long term, which is extremely useful in chaotic dynamics. Recall that, in a chaotic system, two close points may end up being far apart.

4.4 Results

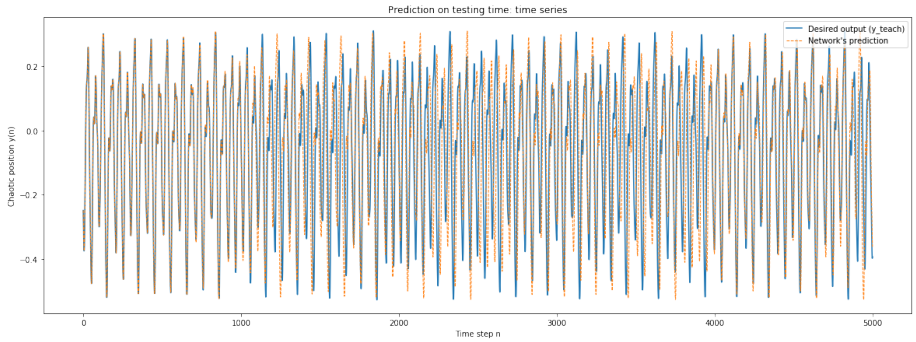
All the experiments performed can be found on the attached *MackeyGlass.ipynb* notebook. In it, we have played with different initial conditions as well with different values of the t_{dismiss} parameter, which allowed us not only to predict but also to replicate the dynamics seen during training. However, this section will just provide the main results, all regarding the prediction task.

4.4.1 Mildly chaotic system: $\sigma = 17$

As stated in Section 4.2, the network has been trained regarding $T = 3000$ and $T = 21000$. In addition, we considered different initial conditions, which were $y(0) = 1, 1.2, 1.5, 5$. The results are exposed on Figure 4.2. In it, we are using the notation of $\text{MSE}_{\text{test}}(T, M)$ to indicate that it corresponds to the prediction of the M time



(A)



(B)

FIGURE 4.3: Network's prediction after being trained for $T = 3000$ time steps. In both cases, the initial condition is $y(0) = 1.2$. **(A)** Predicting the next 1000 steps. **(B)** Predicting the next 5000 steps.

steps that follow the T steps used for training. By looking at the already mentioned figure, we see that when 1000 steps are predicted, the results are more accurate for $T = 21000$ than for $T = 3000$. However, it appears that the precision achieved for $T = 3000$ is good enough in most of the experiments. If we now regard the experiments for which 5000 steps were predicted, we see that the results obtained for $T = 21000$ are slightly worse than for $T = 3000$ but quite equivalent. Another good metrics for measuring the precision of the predictions is the NRMSE_{84}. Notice that, in some experiments, such metrics gets better for $T = 21000$ whereas, in some others, it gets worse. Despite such slight variations, both metrics are quite similar for both values of T.

In addition, Figure 4.3 shows the prediction of 1000 and 5000 time steps regarding $T = 3000$ and $y(0) = 1.2$. All the plots obtained for other initial conditions and for $T = 21000$ are quite similar, thus we are only showing it once. In any case, all the images can be found on notebook *MackeyGlass.ipynb*. By looking at that figure, we can see that the network predicts quite precisely the next 1000 time steps that follow the training although it can just approximate the next 5000. In addition, it is worth noting that, even though the prediction may degenerate, it still follows a dynamics that resembles that of a Mackey-Glass system. In addition, the degenerated prediction may end up recovering precision and getting closer to the original one whenever it encounters less chaotic behaviors. The latter can be seen in Figure 4.3 (B), where it degenerates after 1000 time steps but then it recovers precision at time step 4000.

As we have already discussed, the network predicts quite precisely the next 1000 time steps that follow the training and it then degenerates. On the other hand, it

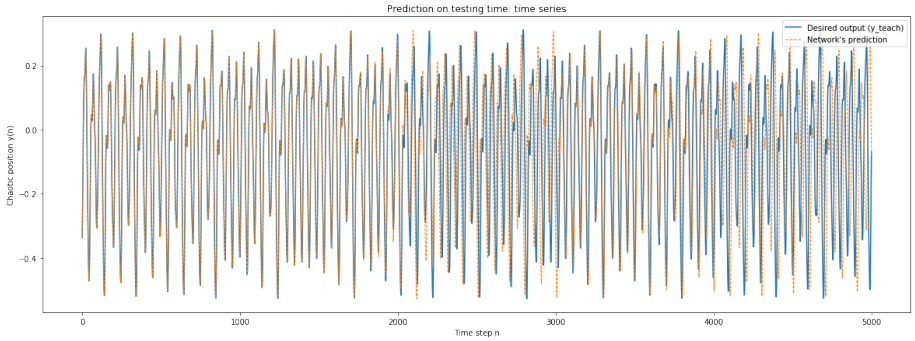


FIGURE 4.4: Prediction of 5000 time steps after increasing the network's capacity.

is true that such degeneration still resembles a Mackey-Glass dynamics and, after some time steps, the predictions may recover precision. Therefore, it seems that the problem does not lie on the quantity of data available (since the results for $T = 3000$ and $T = 21000$ have appeared to be similar) but on the network's capacity. In other words, a reservoir with 400 nodes is not capable of learning the dynamics at the extent of precisely predicting 5000 time steps. Consequently, we incremented the network's capacity by regarding a reservoir with 1500 nodes. After training the network with $T = 7000$, we could quite accurately predict 2000 time steps, which can be visually inferred by looking at Figure 4.4. In addition, the value of the NRMSE_{84} was divided by 100, which implies that this network is 100 times more precise in the long term than any of the previous ones.

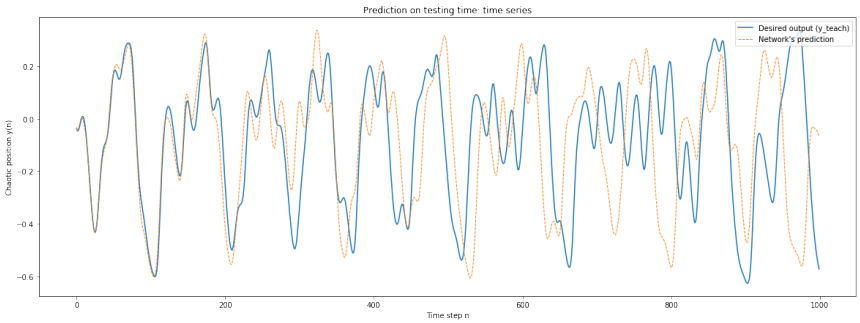
4.4.2 Wildly chaotic system: $\sigma = 30$

Similarly than in Section 4.4.1, we performed different experiments, which are summarized in Figure 4.5. In addition, Figure 4.6 (A) depicts the prediction for 1000 time steps regarding $T = 3000$ and $y(0) = 1.2$. Again, since all the obtained plots appear to behave similarly, we will only provide one. What we observe is quite similar to the previous case. That is, the network degenerates after 200 or 250 time steps although the degenerated prediction still resembles a Mackey-Glass dynamics. In addition, the predictions may recover accuracy as time goes by. Apart from that, the accuracy got for $T = 3000$ and $T = 21000$ is quite similar. Therefore, all these observations suggest that, in order to improve the learning, we need to increase the network's capacity. Consequently, we increased the number of nodes from 400 to 1500. After training the network for 7000 time steps, the prediction is as shown in Figure 4.6 (B). Notice that, now, the network could precisely predict around 800 time steps. In addition, the value of the NRMSE_{84} has been divided by 10. Therefore, this network is 10 times more precise in the long term than any of the previous ones.

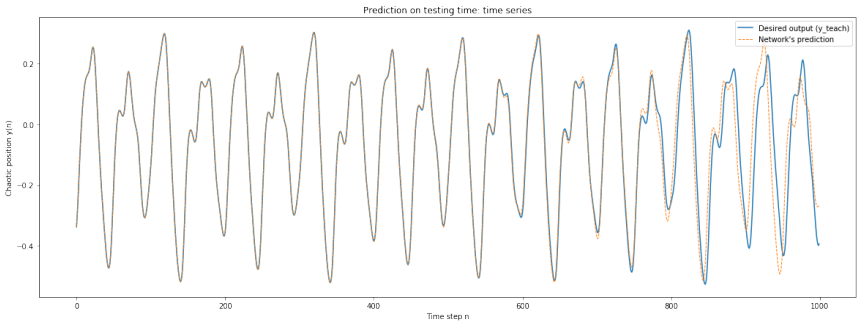
Finally, let us remark that, when dealing with $\sigma = 30$, some noise needed to be introduced. Recall that it was uniformly chosen from $[-0.00001, 0.00001]^{400}$. In the already mentioned notebook, we performed different experiments regarding the noise to be on the interval $[-a, a]^{400}$ for different values of a. After performing such experiments, we could see the importance of choosing the appropriate noise. For instance, we saw that, if the value of a is too big, then the prediction will end up degenerating, although it will still follow a trajectory resembling that of a Mackey-Glass system. On the other hand, if we do not introduce noise or the value of a is too

Initial Condition	Errors	T=3 000	T=21 000
y(0)=1.2	NRMSE_{84}	0.90456	0.37840
	$\text{MSE}_{\text{test}}(T/200)$	0.00271	0.00157
	$\text{MSE}_{\text{test}}(T/1000)$	0.07000	0.10662
y(0)=1	NRMSE_{84}	0.79282	0.38680
	$\text{MSE}_{\text{test}}(T/200)$	0.01376	0.00057
	$\text{MSE}_{\text{test}}(T/1000)$	0.12254	0.08265
y(0)=1.5	NRMSE_{84}	1.12072	0.36563
	$\text{MSE}_{\text{test}}(T/200)$	$3.96 \cdot 10^{-5}$	0.00011
	$\text{MSE}_{\text{test}}(T/1000)$	0.07439	0.06813
y(0)=5	NRMSE_{84}	0.71455	0.40585
	$\text{MSE}_{\text{test}}(T/200)$	0.00017	0.00064
	$\text{MSE}_{\text{test}}(T/1000)$	0.05028	0.11481

FIGURE 4.5: Performance comparison of two different values of T regarding $\sigma = 17$



(A)



(B)

FIGURE 4.6: **(A)** Predicting the next 1000 steps. **(B)** Predicting the next 1000 steps after increasing the network's capacity.

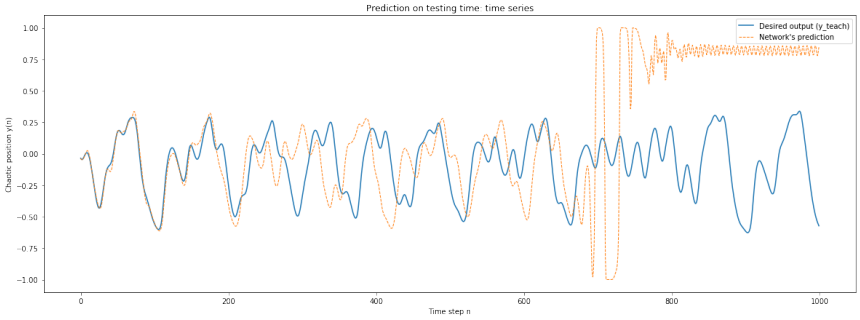


FIGURE 4.7: Predicting 1000 time steps for $\sigma = 30$ when no noise is added.

small, the predictions will degenerate. In this case, the prediction will be far from a Mackey-Glass one. Figure 4.7 illustrates the predictions when no noise was introduced.

Therefore, we found two possible ways of degenerating: the one that still follows a trajectory that resembles a Mackey-Glass dynamics and the one that gets stuck at ± 1. The former can be solved by increasing the network's capacity and the value of training samples whereas the latter could be work out by introducing some noise.

Chapter 5

The standard map

The main goal of this chapter is to use all the knowledge acquired during the previous chapters in order to learn the dynamics of the so called *standard map*. That is, we want to train an echo states network in order to make it capable of predicting the different trajectories of such map. We decided to deal with this system since it displays a very challenging behavior and we did not find any literature regarding the application of the reservoir computing on it.

We start the chapter by providing some notions on the standard map (Section 5.1), which follow *Chirikov standard map* and *Kicked Rotator*. Afterwards, we expose the network and training parameters and procedures that gave the best performance (Sections 5.2, 5.3 and 5.4). Finally, Section 5.5 provides an overview of the different strategies we tried before reaching the best result. Such approaches are worthy to mention and study since, although they may had not worked in this particular case of study, they belong to the set of typical tunning strategies. Therefore, they illustrate a way of systematically choose the most suitable strategies and parameters.

All the experiments performed in this chapter can be found on the attached notebooks called *StandardMap.ipynb* and *StandardMap_Parallel.ipynb*. Along this chapter, we are going to detail the main conclusions we reached as well as the most remarkable features observed. However, we are not going to discuss all the experiments performed, since they can be found on the already mentioned notebooks.

5.1 Preliminaries on the standard map

The *Chirikov standard map* (or simply *standard map*) is an area-preserving map

$$\begin{array}{rccc} F : [0,2\pi] \times [0,2\pi] \times \mathbb{R} & \to & [0,2\pi] \times [0,2\pi] \\ (p,\theta,t) & \mapsto & F(p,\theta,t) \end{array}$$

defined by means of the following differential equation

$$\begin{cases} \dot{p} &= p + K\sin(\theta) \\ \dot{\theta} &= \theta + \dot{p} \end{cases},$$

where θ and p determine the angular position and the angular momentum. The parameter K is a non-negative scalar giving the degree of chaos. When $K = 0$, the orbits are linear. Then, as this value increases, periodic and quasi-periodic orbits appear. Finally, if we increase the value even more, so does the probability of observing chaotic dynamics for the appropriate initial conditions.

This map is generated by the Hamiltonian

$$H(p,\theta,t) = \frac{p^2}{2} + K\cos(\theta)\delta_1(t),$$

where $\delta_1(t)$ is a periodic $\delta-$function with period 1 in time.

In our study, we are not going to use the continuous EDO but its discrete version, which is

$$\begin{cases} p_{n+1} &= p_n + K\sin(\theta_n) \\ \theta_{n+1} &= \theta_n + p_{n+1} \end{cases}, \tag{5.1}$$

where p_n and θ_n are taken modulo 2π. Therefore, we are regarding F to be defined on the torus.

The standard map describes the dynamics followed by a *kicked rotator*. A *kicked rotator* is usually imagined as a particle constrained to move on a circle in a system with no friction nor gravity that is periodically kicked. Thus, its parameters are:

- θ, which gives the angular position within the circle. It is measured in radians.
- p, which gives its angular momentum. It can be positive or negative, where positive indicates counter-clockwise motion and negative indicates clockwise motion.
- K, which stands for kick strength. For low kick strengths, the particle motion is fairly regular. However, if the kicks are strong enough, $K > K_c \approx 0.971635\ldots$ the system becomes chaotic and has a positive Maximal Lyapunov exponent (MLE).

The particle starts out at a random position on the circle with a random initial momentum and is periodically kicked by an "homogeneous field". Depending on the position the particle is located within the circle, the kick affects the particle's motion differently. In particular, in order to compute how effective the kick is at a certain position θ, its value K is multiplied by $\sin(\theta)$.

Figure 5.1 illustrates the phase space of the standard map for different values of the kick. In our particular case, the phase space is a two dimensional space where each axis stands for one of the standard's map variables. That is, the x-axis corresponds to θ whereas the y-axis corresponds to p. The phase space depicts all the orbits (i.e, trajectories) for any initial condition given by (θ_0, p_0). By looking at the image, we see that, when $K = 0$, each trajectory is a set of points that lie on the same line. The reason is that, in such case, the value of p is constant and the value of θ is proportional to p (module 2π). As the value of K increases, quasi-periodic and periodic orbits appear ($K = 0.5, 0.9$). When the value of K reaches 0.9, we observe trajectories that seem to be slightly chaotic. This would be the case of the light blue and pink curves. For $K = 1.2$, such chaotic trajectories are easier to distinguish. That is, we see a wide range of the space filled with points of different colors. Finally, it is clear that, as the value of K keeps increasing, so does the probability of finding chaotic trajectories.

5.2 Setting the task

The main goal is to train an echo states network so as to learn the dynamics given by the standard map. Similarly to Chapter 4, the learning task consists on training

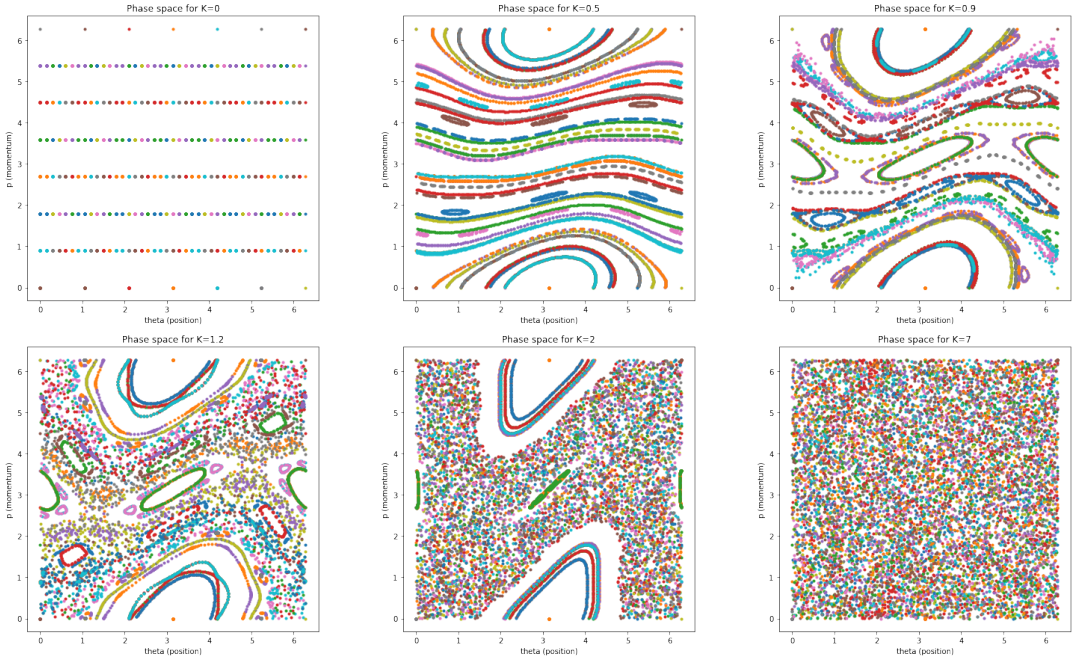


FIGURE 5.1: Phase space of the standard map for different kick values. The first row corresponds to non-chaotic behaviors whereas the second one depicts some chaotic orbits.

the network for a sequence of a certain size T in order to make it capable of predicting the sequence for $t > T$. Therefore, the desired output, denoted as y_{teach}, will represent a certain trajectory of the standard map. That is,

$$y_{\text{teach}}(n)^T = (\theta(n), p(n)),$$

will correspond to the position of the particle at time n. Many experiments regarding different initial conditions and values of K will be carried out.

In our case of study, we will deal with a reservoir composed by 3000 nodes. Such nodes are going to be randomly connected taking weights equal to 0, 0.4 and -0.4 with probabilities 0.9875, 0.00625 and 0.00625 respectively. Therefore, we will only regard 1.25% of connections. Finally, such weights are going to be rescaled in order to obtain an adjacency matrix with an spectral radius of 1.35.

The matrix giving the feedback connections (W^{back}) is going to be constructed by randomly taking uniformly distributed numbers within the interval [-0.56,0.56]. In addition, we will also input a constant value of 0.5 in order to encourage even more a wider variety of dynamics (i.e, u(n)=0.5 for all n). Its matrix of connections (W^{in}) will take values 0, 0.7 and -0.7 with probabilities 0.5, 0.25 and 0.25 respectively.

These values of the parameters appeared to be the ones that gave the best performance after some trial and error. In Section 5.5, we will provide an overview of the different strategies we tried before reaching the best result.

As already mentioned, y_{teach} will be a trajectory of the standard map, which is constructed as explained in equation (5.1). Recall that the parameters θ and p are taken modulo 2π. Therefore, the trajectory will take values on the interval [0,2π]. However, since the activation function is going to be a *tanh*, we want our sequence to

take values in the interval [-1,1]. That is the reason why such sequences will undergo the following transformation

$$y_{\text{teach}} := \frac{y_{\text{teach}}}{2\pi} - 0.5.$$

Therefore, unless we specify the contrary, y_{teach} is going to denote such squashed version of the trajectory.

5.3 Training and testing

The network is going to be trained for $T = 3000$ time steps regarding a dismissal of $n_{\min} = 500$ steps.

The training and testing procedures are going to be as exposed in Section 2.3. However, we will now introduce some noise during training in order to prevent the predictions from degenerating in the long term. Therefore, the update of the reservoir while training is going to be as follows

$$x(n) = \tanh\left(W^{\text{in}}u(n) + Wx(n-1) + W^{\text{back}}y^{\text{teach}}[:, n-1] + \nu[:, n]\right), \tag{5.2}$$

where we are going to take ν as a centered Gaussian noise with standard deviation of 0.001. On the contrary, the update during testing will be:

$$x(n) = \tanh\left(W^{\text{in}}u(n) + Wx(n-1) + W^{\text{back}}y[:, n-1]\right). \tag{5.3}$$

In equations (5.2) and (5.3), we are considering y_{teach} and y as matrices of sizes $y^{\text{teach}} \in M_{2\times T}(\mathbb{R})$ and $y^{\text{teach}} \in M_{2\times T'}(\mathbb{R})$ respectively, where T' is the testing time steps.

Recall that in the experiments we carried out so far (in the previous chapters), the output of the network was computed as a linear combination of the internal states of the reservoir. That is, the output was computed as

$$y(n) = W^{\text{out}}x(n),$$

where $W^{\text{out}} \in M_{L\times N}(\mathbb{R})$ was the matrix to be learnt. However, in this case of study we will slightly modify such approach in order to get more accurate outputs. The strategy we will be following is:

$$y(n) = W^{\text{out}}\tilde{x}(n), \tag{5.4}$$

where

$$\tilde{x}(n) := \begin{pmatrix} x(n)_{1:\frac{N}{2}} \\ x(n)^2_{\frac{N}{2}+1:N} \end{pmatrix} \tag{5.5}$$

stands for the vertical concatenation of vectors $x(n)_{1:\frac{N}{2}}$ and $x(n)^2_{\frac{N}{2}+1:N}$. In addition, the notation of $x(n)_{a:b}$ represents the slice of vector $x(n)$ from component a to component b. Therefore, in equation (5.5), the above vector stands for the first half of $x(n)$ whereas the below one stands for the second half of $x(n)$ to the square.

By following this approach, we want to end up having a set of nodes' trajectories in such a way that the desired output belongs, not to the subspace generated by them, but to the subspace generated by them and its squares. Notice that this

approach has more probabilities of yielding good results.

Finally, the learning will consist on a *ridge regression* instead of the classical linear regression. Thus, we will need to minimize the following function

$$\text{RIDGE} = ||\tanh^{-1}\left(y^{\text{teach}}\right) - W^{\text{out}}\tilde{x}||_2^2 + \alpha||W^{\text{out}}||_2^2,$$

where $y^{\text{teach}} \in M_{2\times(T-n_{\text{min}})}(\mathbb{R})$, $W^{\text{out}} \in M_{2\times N}(\mathbb{R})$ and $\tilde{x} \in M_{N\times(T-n_{\text{min}})}(\mathbb{R})$. Recall that T is the number of time steps considered and n_{min} is the number of discarded time steps. In addition, we will take α=10.

The choice of the insertion of noise in equation (5.2), the computation of the output as a quadratic combination (equation (5.4)) and the Ridge regression instead of the classical linear regression were done after some trial and error. Section 5.5 provides an overview of the discarded approaches.

5.4 Experiments

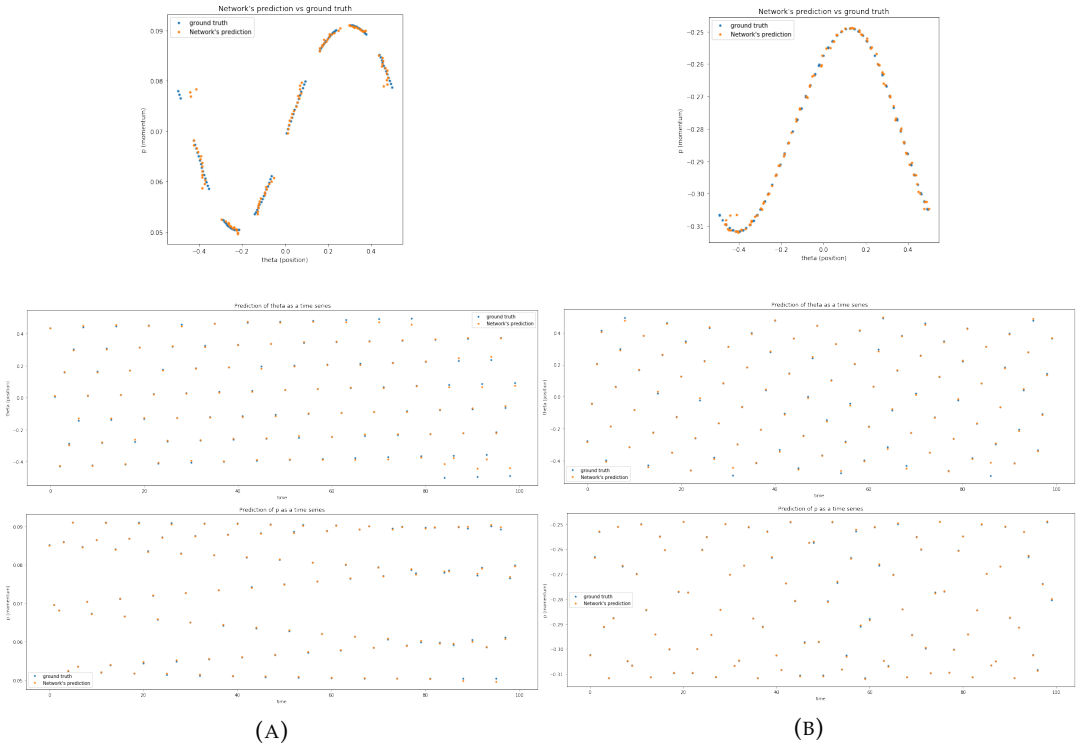
Along this section, we are going to present the main conclusions from the different experiments performed. All of them can be found in the notebook *Standard-Map.ipynb*.

The kick values we are going to consider in these experiments are: 0, 0.1, 0.25, 0.5, 0.75, 0.9, 1.2, 1.5, 2, 3, 5 and 7. Although the first 5 cases will not display a chaotic dynamics, they are worthy cases of study since their trajectories are not going to be trivial. The reason is the fact that the values of θ and p are taken modulo 2π. Therefore, if the initial condition is different from

$$(\theta_0, p_0) = \left(\frac{2\pi}{a}, \frac{2\pi}{b}\right) \quad \text{for } a, b \in \mathbb{Z}_{>1},$$

then, even though the points are reduced to belong to [0,2π], two different points of the trajectory will not perfectly coincide when taken in such modulo. Notice that this is still true even if we squash, since squashing does no cause overlapping of points. The initial conditions we have considered are: (0, 0.75), (0,0), ($2\pi/7$, $2\pi/8$), (0,0.9), (0.9,0), (0.5,3), (1,3.5) and (1,1.2).

What can be inferred after carrying out the experiments is the fact that the network can properly predict at least a few time steps for all initial condition and all value of k lower than 0.9. Afterwards, the predictions may degenerate in two different ways according to the NRMSE_{84}: the mistakes are amplified as time goes by OR the dynamics end up catching again the true trajectory. In the former case, such metrics are large whereas, in the latter case, such metrics are smaller. Figure 5.2 depicts two cases where the first 100 predictions coincide almost perfectly with the desired values. However, the NRMSE_{84} for both parameters in (A) are lower than 2 whereas in (B), the NRMSE_{84} for θ takes value 15, which is very large. The reason for these two different behaviors is the fact that the network is trained in order to learn a dynamics. Therefore, it may be possible to learn a dynamics that is close to the original one but that, at some points, it differs a little bit. This would correspond to the cases where the metrics NRMSE_{84} is lower. On the contrary, it may also occur that the network commits an error when predicting and, then, this error is propagated along



(A) (B)

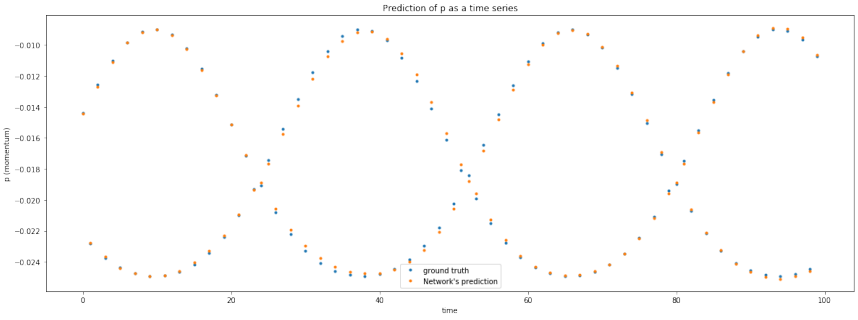
FIGURE 5.2: Comparison of the prediction against the ground truth for two different initial conditions and $K = 0.25$. **(A)** (θ_0, p_0)=(1,3.5). **(B)** (θ_0, p_0)=(1,1.2).

time. This would correspond to the cases where the metrics NRMSE_{84} is higher.

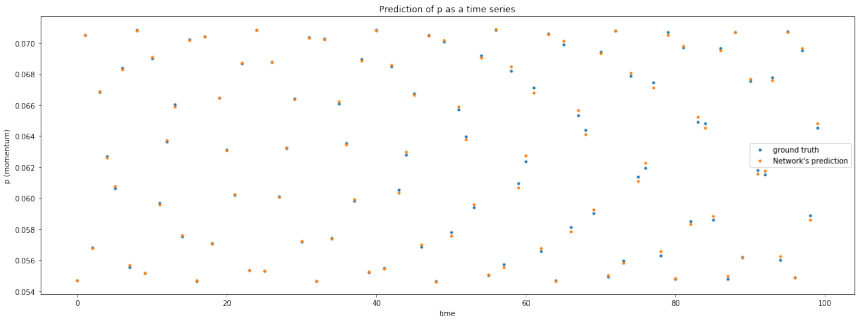
Finally, let us also mention the fact that being able to predict the trajectories of the standard map for non-chaotic values of K do not appear to be a trivial task. Figure 5.3 depicts different orbits for different non-chaotic values of K and different initial conditions. Notice that non of the behaviors depicted there seem trivial. Notice that even images (A) and (D) do not follow a clearly distinguishable pattern. What is more, the orbits seem to have some subtle random behavior.

Let us now tackle the chaotic case. Recall that the trajectories may become chaotic whenever the kick value is higher than 0.98. The experiments showed that, when the orbits displayed a chaotic behavior, then the network was not able to properly predict. By looking at the second row of Figure 5.1, it can be seen that, if the value of K is greater than 0.98 but not too large, then not all orbits are necessarily chaotic. Therefore, in our experiments, we could quite accurately predict orbits for chaotic values of K as long as the orbits were non-chaotic. On the contrary, if the orbits displayed a chaotic pattern, then they could not be learnt. Figure 5.4 shows one non-chaotic orbit properly predicted and a chaotic orbit whose prediction completely missed the ground truth. In both cases, K=1.2, thus a chaotic value of K was considered.

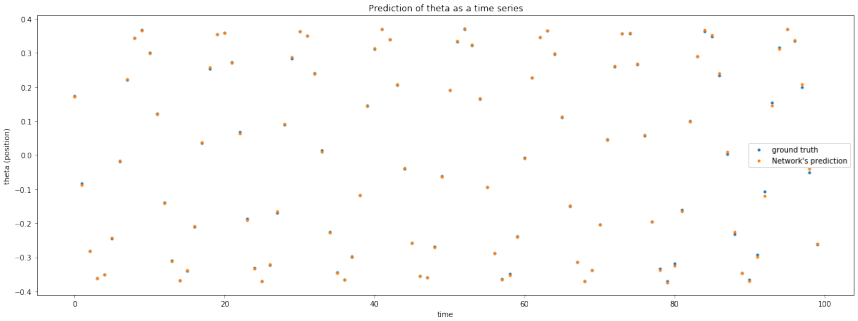
What appears to be the main cause of the poor predicting capability of the network when dealing with an standard's map chaotic orbit is the fact that its dynamics is not continuous. Such discontinuity makes the orbits look like a point cloud without structure (see Notebook *StandardMap.ipynb*). Thus learning the dynamics become a hard task.



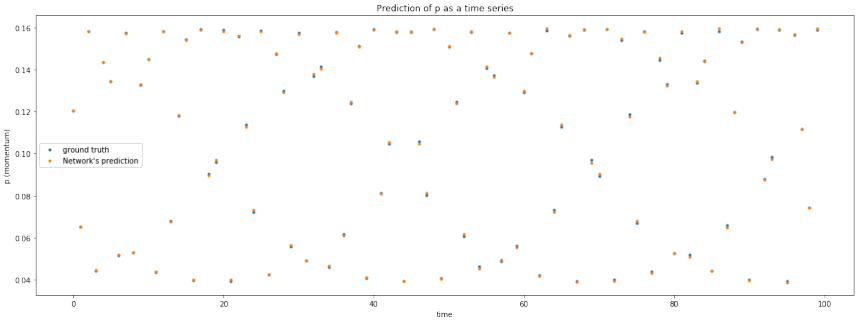
(A)



(B)

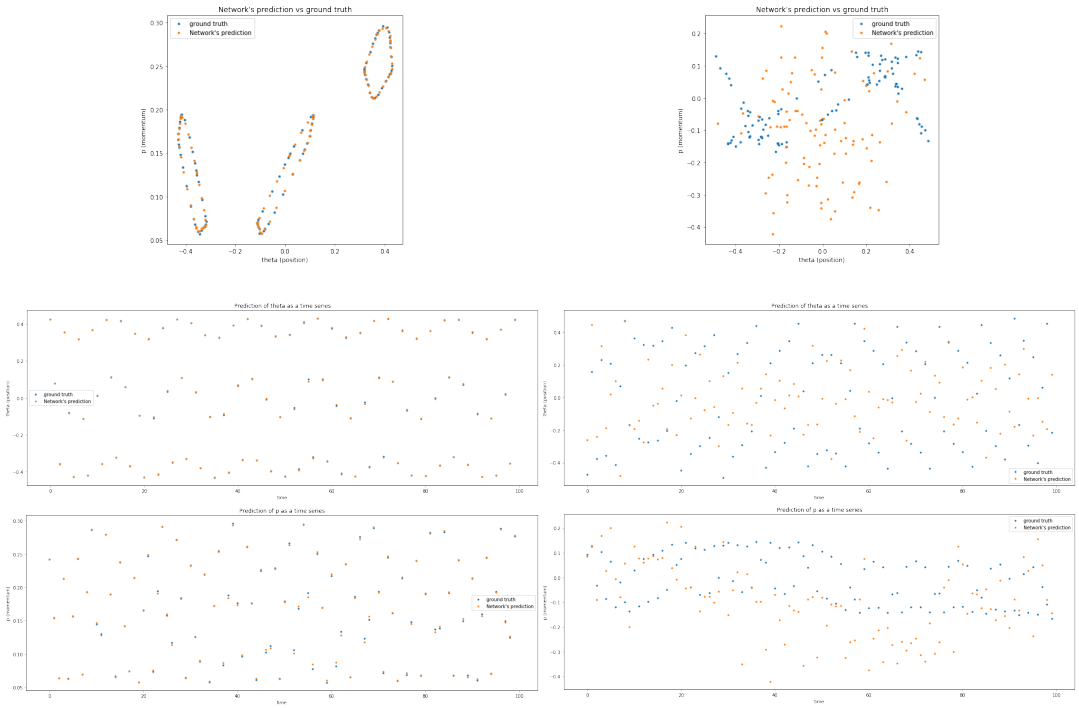


(C)



(D)

FIGURE 5.3: **(A)** (θ_0, p_0)=(0.5,3) and K=0.1. **(B)** (θ_0, p_0)=(1,3.5) and K=0.1. **(C)** (θ_0, p_0)=(0.9,0) and K=0.75. **(D)** (θ_0, p_0)=(1,3.5) and K=0.75.



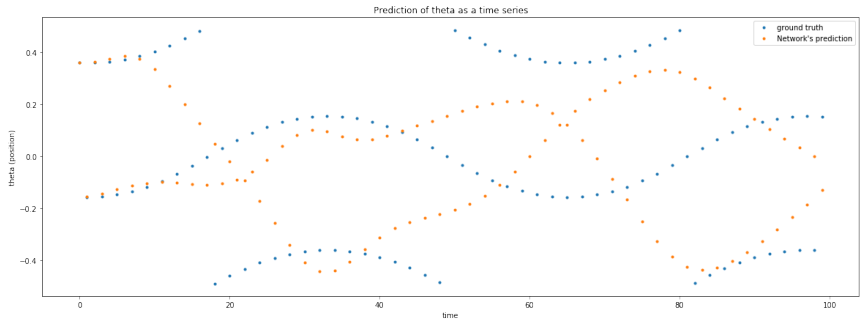
(A) (B)

FIGURE 5.4: **(A)** Orbit of (θ_0, p_0)=(1,3.5) taking $K = 1.2$. **(B)** Orbit of (θ_0, p_0)=(0,0.9) taking $K = 1.2$.

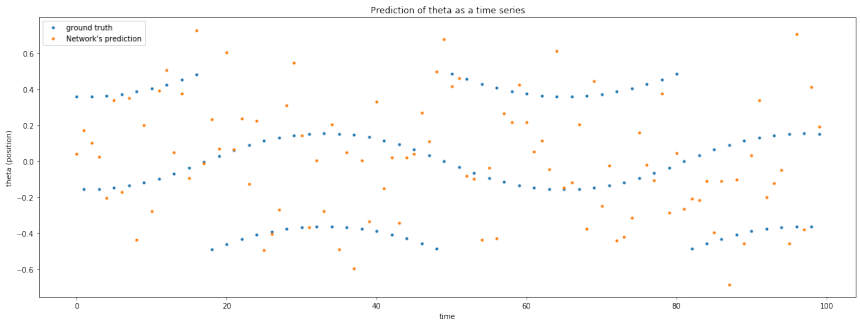
Another widely used approach in the area of reservoir computing consists on training one reservoir for each bunch of variables instead of training a single reservoir. The main advantage of this technique is that each reservoir only takes into account the relationship among the variables that belong to the same bunch. This method is very useful when the parameter space is high dimensional and each component is only influenced by its neighboring components. At first, we may think that this approach is not suitable for our particular case of study, since our parameter space is only two dimensional and both variables are highly related. However, we still tried to train two reservoirs in parallel, each one regarding a single variable of the standard map. The experiments can be found on the notebook *Standard-Map_Parallel.ipynb*. What we saw is that the only case where the same performance can be achieved with less computational cost is when the value of K is equal to 0. Recall that, in this case, the value of p is constant, thus the only parameter to be learnt is θ. On the other cases, the outcome may depend on the initial condition. If it is true that, in some cases, training a single parameter allows to reduce a little bit the value of NRMSE_{84}, it is also true that there is not much difference on the short term prediction. Despite these few cases, in the majority of experiments, the results appeared to be better when both parameters were jointly trained. As we previously mentioned, this was the expected behavior.

5.5 Other trials with poor performance or too much computational cost

As already mentioned, Section 5.4 showed the best performance we were able to obtain by tunning all the different components of the network as well as the training.



(A)



(B)

FIGURE 5.5: Two different predictions for the same trajectory that differ on the choice of the spectral radius.**(A)** Spectral radius of 0.99. **(B)** Spectral radius of 3.

Let us now provide an overview of the different strategies we tried before reaching the best result. Such approaches are worthy to mention and study since, although they may had not worked in this particular case of study, they belong to the set of typical tunning strategies.

Let us now review the different trials carried out. All of them can be found on Section 3 of the notebook *StandardMap.ipynb*.

- **Parameters of the reservoir**. Within these group, we will focus on the number of nodes and the spectral radius, since the other parameters of the reservoir appeared to be quite robust. On the one hand, we saw that, if the number of nodes was too low, then the network is not capable of learning complex dynamics. However, it is also true that we saw that, even regarding N=400, the results were not as bad as one would expect, since, in the short term, the predictions lied on a reasonable neighborhood of the ground truth. Despite this fact, the predictions quickly degenerated. Then, if the number of nodes was too large, the NRMSE_{84} could decrease a little bit, entailing an improvement on the long term predictions. Despite this fact, not much was improved, which leads us to think that a huge number of nodes does not guarantee a good performance.

  On the other hand, we also tunned the spectral radius. We saw that, if it was low, the network learnt some simple continuous dynamics, which had nothing to do with the desired one. On the contrary, if it was too big, it seemed that the network was not learning any particular dynamics but a quite random cloud of points. This fact is illustrated in Figure 5.5.

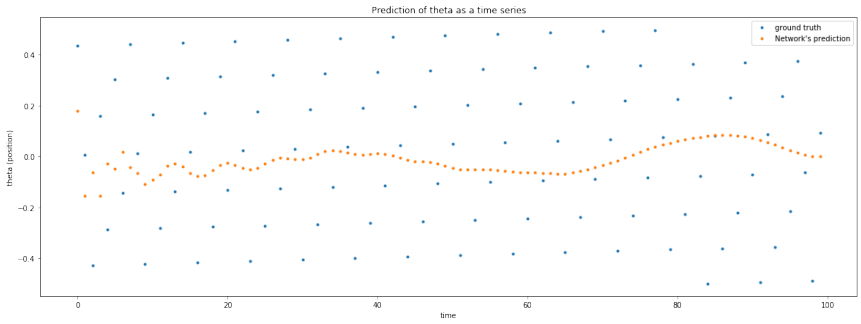


FIGURE 5.6: Predictions regarding the updating formula exposed in equation (5.6).

- **Input**. At first, one may decide to not define an input, which would make sense since the only main feature of the network is the feedback. However, not introducing an input entails a poor performance not only in the long term but also in the short term. On the other hand, the choice of the input value has to be the appropriate, since a too lower or too higher value may not provide the desired result.

- **Inserting noise during training**. We carried out some experiments where we played with the noise: changing the standard deviation of a centered Gaussian noise, changing the Gaussian noise for a uniformly distributed noise and even removing the noise. As we already saw when dealing with the Mackey-Glass system, the lack of noise leads the predictions to degenerate in the sense of ending taking value -1 or 1. On the other hand, we saw that the Gaussian noise gave the best performance. In addition, its parameters had to be chosen in order not to be too large or too low. In the former, the noise's influence was too big so as to properly learn the desired dynamics whereas in the latter, the influence of the noise was too subtle to make a difference.

- **Formula for updating the reservoir**. In this particular task, we updated the reservoir by means of the following equation

$$x(n+1) = \tanh\left(W^{\text{in}}u(n) + Wx(n) + W^{\text{back}}y^{\text{teach}}[:,n] + \nu[:,n]\right),$$

where y_{teach} is replaced by the output of the network (y) during testing time. This strategy, which is the most common one, was already implemented when coping with other tasks. However, recall that when dealing with the Mackey-Glass system, the updating was done as follows

$$x(n+1) = (1-\delta Ca)x(n) + \delta C\tanh\left(Wx(n) + W^{\text{back}}y^{\text{teach}}[:,n]\right), \tag{5.6}$$

where δ=1, C=0.49 and a=0.9. Notice that, when $\delta = C = a = 1$, the second formula reduces to the first one. Figure 5.6 depicts the predictions when the update was done by means of equation (5.6) (and adding some noise). Notice that the predictions seem to be an average of the real ones. The reason why this approach does not work in our case is the fact that it is very suitable when dealing with slow continuous dynamics, which is not our case.

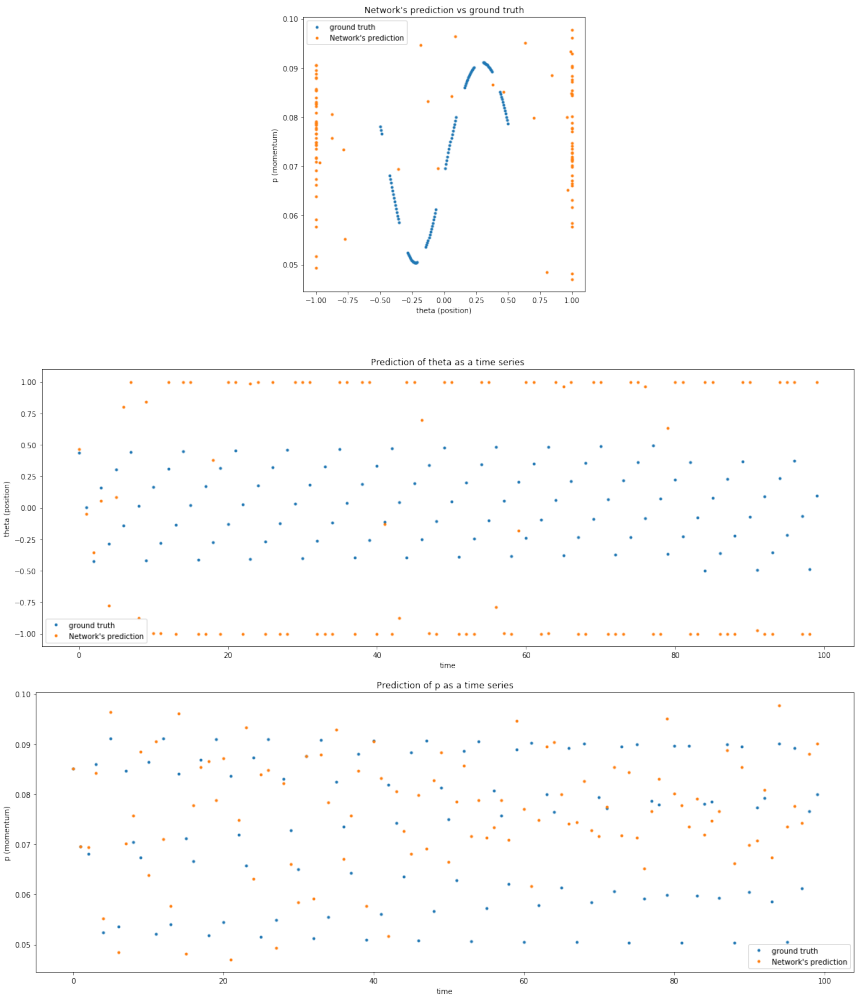


FIGURE 5.7: Predictions when the loss function was the classical MSE.

- **Formula for obtaining the output**. Recall that, in our case, the output is computed from the internal states of the reservoir as follows

$$y(n) = W^{\text{out}}\tilde{x}(n),$$

where

$$\tilde{x}(n) := \begin{pmatrix} x(n)_{1:\frac{N}{2}} \\ x(n)^2_{\frac{N}{2}+1:N} \end{pmatrix}$$

stands for the vertical concatenation of vectors $x(n)_{1:\frac{N}{2}}$ and $x(n)^2_{\frac{N}{2}+1:N}$. However, this was not the first strategy that we tried. We began with classical linear combination. That is,

$$y(n) = W^{\text{out}}x(n).$$

Once we saw that the quadratic combination provided a better outcome, we also decided to try a cubic combination. That is,

$$y(n) = W^{\text{out}}\tilde{x}(n),$$

where

$$\tilde{x}(n) := \begin{pmatrix} x(n)_{1:\frac{N}{3}} \\ x(n)^2_{\frac{N}{3}+1:\frac{2N}{3}} \\ x(n)^3_{\frac{2N}{3}+1:N} \end{pmatrix}$$

stands for the vertical concatenation of the three vectors.

The main difference between the three approaches is the long term prediction. The linear combination approach appears to degenerate quite fast. Then, the cubic one seems to lower the NRMSE_{84} with respect to the quadratic. We decided to work with the quadratic one since it provided a good enough result and it implied less complexity of the reservoir's subspace.

- **Loss function**. The last choice we made was regarding the loss function. Recall that we performed a ridge regression. However, we began trying the classical linear regression. The poor performance of this approach can be seen in Figure 5.7. In addition, we also carried out experiments regarding different values of α. We saw that, if it was to low, then the predictions degenerated in the sense that they ended up taking value ± 1. On the contrary, if it was too large, it also ended up degenerating, although in a more smoother way.

Chapter 6

Conclusions

The main goal of this project was to understand the internal functioning of the reservoir computing approach as well as to analyze its implementation on the task of learning a dynamics, which included both chaotic and non-chaotic dynamics. Let us now summarize both aspects.

Reservoir computing is a framework that inputs a time step of a sequential signal, denoted by $u(n)$, along with the network's previous output, denoted by $y(n-1)$. Then, both are projected onto a higher dimensional space by the maps W^{in} and W^{back} respectively. Afterwards, the new point is run through a deterministic dynamical system called "reservoir", which is defined over the previous mentioned higher dimensional space. Finally, a linear combination of the states of the reservoir is performed by means of a map denoted as W^{out}. The result of this linear combination is the output $y(t)$. The only connections to be learnt are the ones given by W^{out}, since all the others have to be deterministically set in advanced.

While training, the reservoir is updated by means of teacher forcing and, after discarding the initial time steps (state and input forgetting properties), a linear regression is performed in order to obtain the desired output as a linear combination of the states of the reservoir. Therefore, the connections W^{in}, W^{back} and W have to be deterministically chosen in order to guarantee that the subspace generated by the trajectories of the nodes of the reservoir contains the desired output.

As we have seen along the project, whenever the reservoir computing framework is implemented, many choices regarding the parameters and the procedures have to be done. By means of the experiments we have carried out, we tried to identify the role played by such elements. Let us now summarize our findings:

- *REGARDING THE RESERVOIR*
  - **Number of nodes of the reservoir**. They determine the network's capacity.
  - **The connections W^{in} and W^{back}**. They are usually randomly chosen.
  - **The internal connections of the reservoir,** W.
    - The most important feature of W is its sparsity, since it encourages a wider range of nodes' behaviors. Therefore, it guarantees the inhomogeneity of the reservoir
    - The reservoir needs to have echo states. This is guaranteed whether $\sigma_{\max} > 1$ or $\lambda_{\max} < 1$. However, these two conditions appeared to be too restrictive since, in most of our experiments, they were not satisfied.

    - The memory issue. If the value of $\lambda_{\max}$ is low, then the inputs' influence fades out faster. On the contrary, if it is higher, then such influence fades out slower.

- *TRAINING PROCEDURE*
  - If the predictions degenerate quite fast or they degenerate taking values 1 or -1, then **introducing some noise** appears to be a good idea. The choice of such noise is quite case dependent and has to be tunned by trial and error.
  - As already mentioned, the output is a linear combination of the internal states of the reservoir. Thus the loss function is usually the MSE. However, there may be situations where it should be **regularized** in order to get the appropriate result (as happened when dealing with the standard map, for instance). In such cases, the ridge regression appears to be the most popular choice.
- *METRICS*
  - **MSE**. It is the most common way of measuring the training and testing errors.
  - **NRMSE$_{84}$**. This measure solely depends on the trained network. It measures the error in the long term, which is very useful when dealing with chaotic dynamics.
- *MAIN CONCERNS WHEN DEALING WITH CHAOTIC DYNAMICS*
  - In a chaotic orbit, the uncertainty increases as time goes by, because of the definition of chaos. Therefore, the long term predictions appear to be a hard task.

Many research work is being developed in the area of reservoir computing. As already mentioned in Chapter 1, chaos theorist Edward Ott and its team could predict the chaotic dynamics of the Kuramoto-Sivashinsky equation up to 8 Lyapunov times by means of this framework (Pathak et al., 2018b and Pathak et al., 2017). Recall that the reservoir computing is based on solely learning from past data, thus the differential equation giving rise to the dynamics do not need to be modelled. However, in paper Pathak et al., 2018a, the same team used, not only the reservoir computing resource but they also tried to model the system. By means of this *hybrid* approach, the predicting capability was increased up to 12 Lyapunov times. Finally, let us mention that the reservoir computing and the echo states network is not a field only devoted to deal with chaotic dynamics but also to many other areas. As reported in Schrauwen, Verstraeten, and Campenhout, 2007, such framework has been successfully applied in the context of reinforcement learning, digital signal processing (such as speech recognition or noise modeling) and on the Brain Machine Interfacing. In addition, in many areas, such as chaotic time series prediction and isolated digit recognition, reservoir computing techniques have outperformed state-of-the-art approaches. Finally, it is worth mentioning the work of Hart, Hook, and Dawes, 2020, where they expose the mathematics behind the echo states network. In particular, they proved that, under suitable conditions on the dynamical system and on the echo states network, there exists a linear readout layer that allows the ESN to predict the next observation of a dynamical system arbitrarily well. This beautiful result mathematically backs up the reservoir computing framework.

Bibliography

Aoun, Mario and Mounir Boukadoum (Oct. 2015). "Chaotic Liquid State Machine". In: *International Journal of Cognitive Informatics and Natural Intelligence* 9, pp. 1–20. DOI: 10.4018/IJCINI.2015100101.

Chirikov standard map. http://www.scholarpedia.org/article/Chirikov_standard_map. Accessed: 2020-05-1.

Greiner, W. (2010). *Lyapunov Exponents and Chaos. In: Classical Mechanics.* Springer, Berlin, Heidelberg.

Hart, Allen, James Hook, and Jonathan Dawes (2020). "Embedding and approximation theorems for echo state networks". In: *Neural Networks* 128, 234–247. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2020.05.013. URL: http://dx.doi.org/10.1016/j.neunet.2020.05.013.

Jaeger, H. (Jan. 2010). "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note". In: *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report* 148.

Jaeger, Herbert and Harald Haas (2004). "Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication". In: *Science* 304.5667, pp. 78–80. ISSN: 0036-8075. DOI: 10.1126/science.1091277. eprint: https://science.sciencemag.org/content/304/5667/78.full.pdf. URL: https://science.sciencemag.org/content/304/5667/78.

Jiang, Junjie and Ying-Cheng Lai (2019). *Model-free prediction of spatiotemporal dynamical systems with recurrent neural networks: Role of network spectral radius*. arXiv: 1910.04426 [cs.LG].

Kicked Rotator. https://ccl.northwestern.edu/netlogo/models/KickedRotator. Accessed: 2020-05-1.

Machine Learning's 'Amazing' Ability to Predict Chaos. https://www.quantamagazine.org/machine-learnings-amazing-ability-to-predict-chaos-20180418/. Accessed: 2020-02-1.

Pathak, Jaideep et al. (2017). "Using machine learning to replicate chaotic attractors and calculate Lyapunov exponents from data". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 27.12, p. 121102. ISSN: 1089-7682. DOI: 10.1063/1.5010300. URL: http://dx.doi.org/10.1063/1.5010300.

Pathak, Jaideep et al. (2018a). "Hybrid Forecasting of Chaotic Processes: Using Machine Learning in Conjunction with a Knowledge-Based Model". In: *CoRR* abs/1803.04779. arXiv: 1803.04779. URL: http://arxiv.org/abs/1803.04779.

Pathak, Jaideep et al. (Jan. 2018b). "Model-Free Prediction of Large Spatiotemporally Chaotic Systems from Data: A Reservoir Computing Approach". In: *Phys. Rev.*

Lett. 120 (2), p. 024102. DOI: 10.1103/PhysRevLett.120.024102. URL: https://link.aps.org/doi/10.1103/PhysRevLett.120.024102.

Schrauwen, Benjamin, David Verstraeten, and Jan Campenhout (Jan. 2007). "An overview of reservoir computing: Theory, applications and implementations". In: pp. 471–482.

Vlachas, P.R. et al. (2020). "Backpropagation algorithms and Reservoir Computing in Recurrent Neural Networks for the forecasting of complex spatiotemporal dynamics". In: *Neural Networks* 126, pp. 191 –217. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2020.02.016. URL: http://www.sciencedirect.com/science/article/pii/S0893608020300708.