Universidad Autónoma de Madrid
Escuela Politécnica Superior
Departamento de Ingeniería Informática

Advanced Methods for Recurrent Neural Networks Design

Master's thesis presented to apply for the Master in Computer Engineering and Telecommunications degree and the Master in Mathematics and Applications degree

By

Carlos María Alaíz Gudín

under the direction of

José Ramón Dorronsoro Ibero

Madrid, November 10, 2010

Contents

Acknowledgements

The author is kindly supported by the FPU-MEC grant reference AP2008-00167. He has been also partially supported by Spain's TIN 2007-66862 and Cátedra IIC Modelado y Predicción.

He also would like to thank Graz University of Technology for receiving him during his two months visit supported by FPU-MEC grant reference AP2008-00167ESTANCIA-2010.

Abstract

Including temporal information in machine learning to model temporal series can be a difficult task. Several solutions, like the addition of delays as synthetic inputs, has been presented to adapt classical algorithms to this kind of problems. Nevertheless, the more correct treatment seems to be the use of models that retain a temporal context intrinsically. It is with this background where the RNNs arise. These models can trade with temporal information by their own definition, but the classical algorithms to train them suffer from important disadvantages, mainly the computational cost. The Reservoir Computing (RC) paradigm tries to solve this problem, proposing simpler approaches that performs as well as the classical ones.

In this work, a description of the state of the art in RNNs is given. The evolution of the algorithms from the classical ones to the RC approaches is studied, unifying all this techniques with the same theoretical model. This will permit to analyze the consecutive simplifications that form the path from the classical approaches to the new paradigm, in an attempt to justify its surprising efficiency. In the same line some standard experiments comparing several algorithms and contrasting the theory with their real behaviour are included. Finally, the application of these new approaches to the prediction of wind power is presented.

Chapter 1

Introduction

The models most commonly used in machine learning consist on optimal transformations of the inputs into the desired outputs, that is, a mathematical function whose result is defined only by the instantaneous value of the variables. This type of methods are powerful enough for a big variety of problems, but suffer from a big limitation: they do not include explicitly any temporal information.

One of the simplest solutions to this problem is just to use as inputs the original inputs in previous timesteps. So the new set of inputs will be the original inputs plus a set of delays of them. This approach can capture any temporal dependency if the number of delays is big enough (this is a consequence of Takens Theorem [1], see appendix A.1). The disadvantage is that the number of variables produced by this projection of the temporal series into an equivalent multidimensional space is usually very big, so this technique may result into very complex models, and a bad efficiency.

To solve this problem Recurrent Neural Networks (RNNs) arise naturally. The general RNNs derive from the widely used Feedforward Networks, or Multilayer Perceptron (MLPs), but with feedback connections, as it will be defined in the following section. This allows a RNN to retain a temporal context so it may come to be a powerful tool to model temporal series.

As a first approach to RNNs, Hopfield Networks (HN) [2] should be cited. In this model, some connections between units of the internal layers transform it into a dynamical system. This system evolves as the units are updated (one at a time), and it tends to certain stable states. This permits to use HNs as associative memories, in the sense that a set of input patterns will drive the network to the same final state, but used as temporal processing models.

In this work, the state of the art in the field of RNN is described. It is mainly focused on the Reservoir Computing approach, as it will be explained, but the classical algorithms and the evolution of the paradigms are also studied. The goal is to justify somehow the apparition of the ESN algorithm and its efficiency. In fact, this technique works surprisingly well, as it will be seen, so the natural development of this field should be to understand these results. As it will be explained in section 1.2, the evolution of this techniques can be summarized in the scheme of figure 1.0.1. Conceptually we will explain this relationship between the different algorithm, in which consecutive simplifications permits to arrive to the simple ESN algorithm. We will also try to relate these algorithms through the experimental part of the work, although there are some difficulties to compare the algorithms, as it can be the different behaviour of the online techniques versus the batch ones.

$$\begin{tikzcd}
\boxed{\text{Classical Algorithms}} \arrow[r] & \boxed{\text{APRL}} \arrow[r] & \boxed{\text{APRL Online}} \arrow[d] \\
\boxed{\text{BPDC-L}} \arrow[d] & \boxed{\text{BPC-S}} \arrow[l] & \boxed{\text{BPDC-C}} \arrow[l] \\
\boxed{\text{ESN}} & &
\end{tikzcd}$$

Figure 1.0.1: Evolution of the RNN algorithms.

The document is structured as follows. In chapter 1, the general definition of RNNs and the model that will be used are given. Moreover the classical algorithms and the ESN/LSM approaches are described. In chapter 2, a new formulation of the problem due to Atiya and Parlos and the algorithm APRL are studied. BPDC, a natural derivation of the APRL approach, is described next in chapter 3. In this chapter, some techniques for the construction of the reservoir are briefly considered. In chapter 4 several experiments are reported, and a final discussion will is given in chapter 5.

1.1 Definitions and General Descriptions

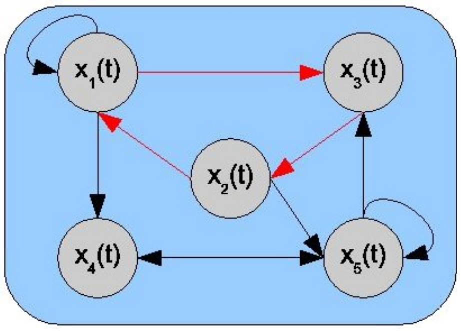
1.1.1 Recurrent Neural Networks

A Neural Network (NN) is a computational model with a certain inspiration in biological neural networks. It consists on a group of process units, called neurons, that can store certain information through the strength of the connections between them (the weights).

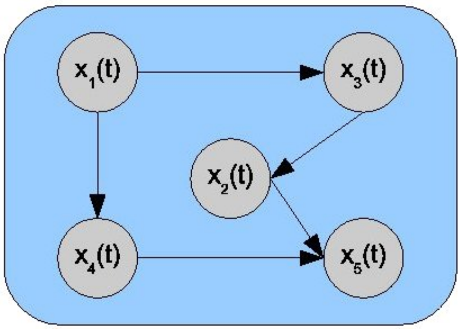
The simplest type of Neural Networks are the already mentioned Feedforward Networks (see [3] for a brief survey), that are characterized by the absence of directed cycles. In these models the information flows from the input units, through the hidden ones, to the output units, so it moves only in one direction. Thus the value of a unit is a function of the previous units connected to it, usually a nonlinear transformation, called the activation function, of a weighted summation.

A Recurrent Neural Network is a Neural Network with at least one cycle in the synaptic connections. This converts the model into a dynamical system, that is, the state of the network does not depend only on the current value of the inputs, but also on the previous complete time series.

As standard MLPs, a RNN is completely defined by the weights of the connections, given the activation functions of the units.



(a) Recurrent NN.



(b) Feedforward NN.

Figure 1.1.1: Difference between a RNN and a MLP. For the RNN, a cycle is marked in red. The MLP connections are defined by an acyclic directed graph.

The main differences between a MLP and a RNN are:

- In the MLPs the information flows from the input to the output, usually through one or more hidden layers. In the RNNs, at least one cyclic path is present (see figure 1.1.1).
- The MLPs implement static input-output functions, while RNNs represent dynamical systems.
- MLPs are *universal approximators* for any nonlinear map (this is the *universal approximation theorem* [4]). RNNs are *universal approximators* for any dynamical system (they can simulate any Turing machine [5]).
- For training MLPs there is a standard algorithm, BackPropagation (BP). For the RNNs there is no clear winner.
- MLPs have been successfully applied to many practical applications. RNNs have been much less applied in practical situations.

In this work the RNNs will be used to solve regression problems with temporal dependencies. A classical regression problem consists on finding a map that converts the inputs into the desired outputs. For this, a training dataset is used, i.e., a set of input-output pairs $\{(u_1, d_1), \ldots, (u_K, d_K)\}$, usually considered as independent, identically distributed samples. This data is supposed generated by a deterministic map ϕ and some non-biased noise, i.e. $d_i = \phi(u_i) + \epsilon_i$. The problem is to find an approximation of such a deterministic map $\tilde{\phi} \approx \phi$ to predict the values of the unknown outputs for some known inputs. In particular, the temporal ordering of the samples is irrelevant.

On the other hand, in a modeling problem with temporal dependencies one or more training sequences, of the form $\{(u_k, d_k)\}_{k=1}^{K}$, are used. In this case, the order of the data is crucial because they are samples of some kind of dynamical system. So the output does not depend only on the current input, but on all the input sequence until that moment. A possible model for this situation could be $d(k) = \phi(u(1), \dots, u(k)) + \epsilon_k$. The goal is then to reproduce that system in order to predict the unknown outputs given an input series.

As stated in [6], the classical methods from training RNN suffer from the following problems:

- The iterative change of the parameters can drive the network dynamic through bifurcations where the gradient information is ill-defined, so the convergence can not be assured [7]. This means that, in contrast to classical MLP, where the error surface is continuous in weight space, in a RNN a change of the weights can modify drastically the behaviour of the system, thus the error surface may be discontinuous.

- The gradient information required for capturing long time dependencies exponentially fades over time (gradient vanishment), so it is very difficult to get a long-range memory without specialized architectures.

- Local minima problems, in which the algorithms get trapped, arise in many problems.

- If not properly set, this kind of algorithms may have high computational costs and can only be applied to small networks.

- Some of these methods usually depends on global control parameters that may be difficult to choose.

In any case, and while not yet very well understood, recurrent networks are a significant extension of standard MLPs that enhances their data processing ability. They are much closer to the powerful biologic neural models and can be a source of new insights even in practical situations.

1.2 Evolution of the RNN Paradigms

In this subsection, a brief overview of the evolution of the main RNN paradigms will be given. These will be discussed in more detail in subsequent sections.

The first algorithms for training RNNs were based on the exact computation of the error gradient, and the iterative update of the weights through a classical gradient descent. The first proposed methods were BackPropagation Through Time (BPTT) and Real Time Recurrent Learning

(RTRL). BPTT [8] is based on the transformation of the RNN into a MLP, where classical Back-Propagation algorithm can be applied. RTRL [9] computes the error gradient in a recursive way using the information of the previous timestep.

A major change in perspective was the Atiya–Parlos Recurrent Learning (APRL) method [10], that arises from the reformulation of the problem of training a RNN. They proposed a new interpretation of training, which is treated as a constraint optimization problem and were able to unify many of the classical methods. APRL is based on the computation of the error gradient with respect to the states, instead of directly with respect to the weights. Once the desired change of the states is known, a proportional change on the weights that produces the states' change is computed. This algorithm partially solves many of the problems listed above, with a faster convergence to the solution.

The next step was BackPropagation Decorrelation (BPDC) [11], developed after an analysis of the behaviour of APRL learning that simplifies some of its terms and results in a much more efficient algorithm. But the most important contribution of this method is that, motivated by the different dynamics of the weights update, it proposes to update only the weights that connect the output. This only requires a linear cost.

More or less simultaneously with BPDC, the Echo State Network (ESN) approach [12] appeared, that focuses only on the training of the output weights. In fact, as it will be explained, it considers a part of the RNN, called the reservoir, that contains only the connections between the hidden units, as a fixed transformation that projects the temporal series. Then, a readout (such as a simple linear regression) is used to compute the output from the state of the reservoir.

The same idea takes place under the Liquid State Machines (LSM) approach [13], but using models more realistic in a biological sense. In this case, a group of spiking neurons (the liquid) processes the inputs, and then the readout (that can be quite complex, such as an MLP) is applied.

1.3 The Reference Model

A definition of the selected RNN model for this work will be given here, so that it can be referred in the following chapters.

The units will be denoted by x_i, with $i = 1, \dots, N$, so $x_i(k)$ denotes the value of the unit i at time k. The inputs are a subset of the units (given by an index set I) whose values are fixed, so

$$x_r(k) = u_r(k), \quad \forall r \in I, \forall k, 1 \leq k \leq K.$$

That is, the values of these units at each timestep k are the values of the corresponding inputs of the sample. The number of inputs will be denoted by $M = |I|$. These input units do not receive feedback connections (in fact, they do not depend on the values of other units but only on the sample).

The outputs are treated in the same way. They are identified by an index set O, so x_i will be an output iff $i \in O$. The desired outputs (i.e., the targets) will be denoted by y_i, with $i \in O$. The number of outputs is given by $L = |O|$. There may be feedback connections arising from the output units.

The update equation of this main model is

$$x(k+1) = (1 - \Delta t)x(k) + \Delta t W f(x(k)), \tag{1.3.1}$$

where Δt is the time constant. When $\Delta t = 1$, then the units themselves have not memory. When Δt is small, close to 0, the value of a unit mostly depends on its previous value, so it changes slowly keeping a certain memory.

This expression, when $\Delta t = 1$, corresponds to the discrete model,

$$x(k+1) = W f(x(k)), \tag{1.3.2}$$

and when $\Delta t \to 0$, approximates the dynamic of the continuous time system

$$\frac{\partial x}{\partial t} = -x + W f(x). \tag{1.3.3}$$

The activation function f is assumed to be, in general, a standard sigmoid differentiable function, centered at zero, like the hyperbolic tangent. It should be noticed that f can change for one unit to another (in fact, this is a way of adapting the reservoir, as it will be explained in subsection 3.4.1), but this dependency will not be expressed explicitly.

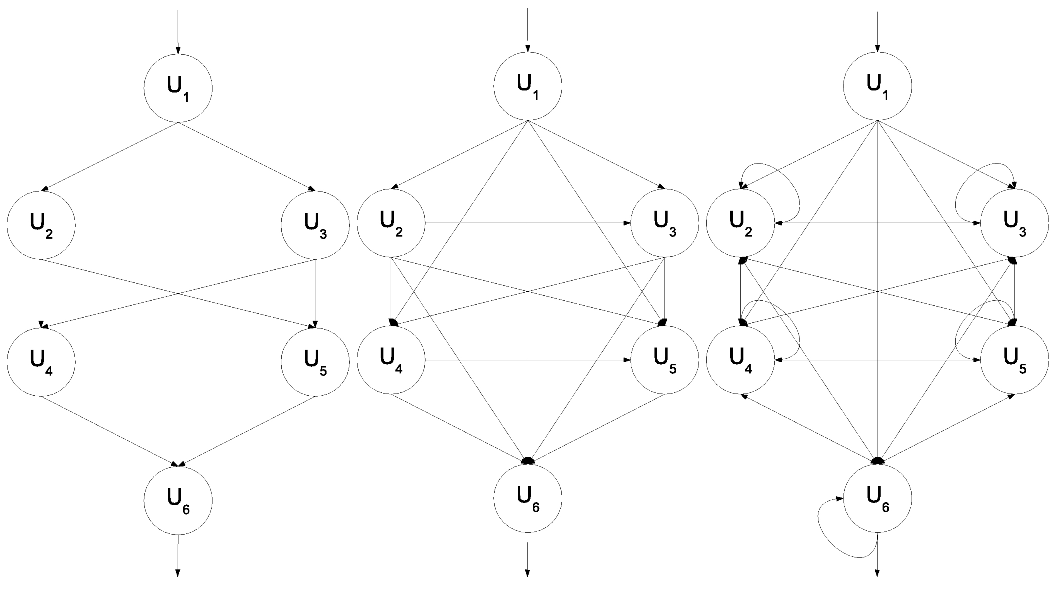
In the model just given, the $x(k)$ values correspond to the activations in standard MLPs. The model can also be expressed in terms of the outputs $o(k) = f(x(k))$ as

$$o(k+1) = f((1 - \Delta t)x(k) + \Delta t W o(k)),$$

or also as the discrete model

$$o(k+1) = f(W o(k)).$$

The weight matrix W represents the topology of the neural network. In the classical models of MLP, where the units are separated into layers, and each layer only receives connections from the previous ones (or from the input layer in the case of the first hidden layer) and we assume a left-to-right ordering of the network units, W is a matrix with blocks under its diagonal. In the general FFNN, where the only restriction is the absence of directed cycles, W is an lower triangular matrix, because the units can be orderer so that the activation in one of them only depends on the previous units. The weight matrix of the RNN can have any form, but in can be divided into blocks in function of the type of units involved in the connections. Thus there can be connections from the input units to the reservoir or to the output ones, from the reservoir to the reservoir itself or to the output units, and from the output units to the reservoir or the output ones. As explained before, the input units do not receive any connection, but W will be considered as a $N \times N$ matrix for simplicity (although a $(N - M) \times N$ matrix would be enough). In figure 1.3.1 an example of these architectures is shown. In figure 1.3.2 the corresponding weight matrices are shown.



(a) MLP. (b) FFNN. (c) RNN.

Figure 1.3.1: Examples of the three different architectures, with one input $i = 1$ and one output $i = 6$.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ X & 0 & 0 & 0 & 0 & 0 \\ X & 0 & 0 & 0 & 0 & 0 \\ 0 & X & X & 0 & 0 & 0 \\ 0 & X & X & 0 & 0 & 0 \\ 0 & 0 & 0 & X & X & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ X & 0 & 0 & 0 & 0 & 0 \\ X & X & 0 & 0 & 0 & 0 \\ X & X & X & 0 & 0 & 0 \\ X & X & X & X & 0 & 0 \\ X & X & X & X & X & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ X & X & X & X & X & X \\ X & X & X & X & X & X \\ X & X & X & X & X & X \\ X & X & X & X & X & X \\ X & X & X & X & X & X \end{pmatrix}$$

(a) MLP. (b) FFNN. (c) RNN.

Figure 1.3.2: Weight matrices for the different architectures of figure 1.3.1.

Under the reservoir computing paradigm, the internal units (those that are neither inputs nor outputs) form the reservoir. The set of weights that interconnect the reservoir will be denoted by the $(N - M - L) \times (N - M - L)$ matrix W^{Res}. The properties of this matrix have a big influence in the dynamic of the RNN.

For these models, the squared error (that the model will try to minimize) is given by the expression

$$E = \frac{1}{2} \sum_{k=1}^{K} \sum_{s \in O} \left[x_s(k) - y_s(k) \right]^2 . \tag{1.3.4}$$

When computing the derivative of the error, the following instantaneous error will be used:

$$e_s(k) = \begin{cases} x_s(k) - y_s(k), & s \in O, \\ 0, & s \notin O. \end{cases}$$

1.4 Classical Algorithms

In this section, the model given by equation (1.3.2) (the discrete one) will be used for both the BPTT and RTRL classical algorithms, so the dynamic considers $\Delta t = 1$ and $x(k+1) = Wf(x(k))$.

1.4.1 BackPropagation Through Time

The most used algorithm for training classical Feedforward Neural Networks, that is, MLPs, is BackPropagation [14]. It consists on the computation of the gradient of the squared error (see equation (1.3.4)) in a recursive manner that permits to optimize the operations.

In the original model of MLP, each unit i in layer k produces an output, o_i^k, that is given by the activation function, f, applied to its activation x_i^k, i.e. $o_i^k = f(x_i^k)$. The activation is computed using the outputs of the previous layer units (or the inputs if $k = 1$). Denoting the number of such units by N_{k-1}, the expression is $x_i^k = \sum_{j=1}^{N_{k-1}} w_{ij}^k o_j^{k-1}$. Bias effects can be achieved adding an extra input unit with a constant 1 value.

The main idea is to estimate first the derivatives of the error with respect to the activations of the units, i.e., $\delta_i^k = \frac{\partial E}{\partial x_i^k}$. These are computed from the last layer (the output one, $k = K$), resulting in

$$\delta_i^K = \frac{\partial E}{\partial x_i^K} = \frac{\partial E}{\partial o_i^K} \frac{\partial o_i^K}{\partial x_i^K} = (o_i^K - y_i^K) f'(x_i^K),$$

to the first layer, $k = 1$, using the previous results in each step through the formula

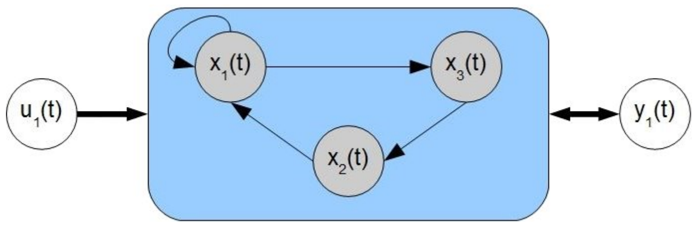
$$\delta_i^k = \frac{\partial E}{\partial x_i^k} = \sum_{j=1}^{N_{k+1}} \frac{\partial E}{\partial x_j^{k+1}} \frac{\partial x_j^{k+1}}{\partial x_i^k} = \sum_{j=1}^{N_{k+1}} \delta_j^{k+1} f'(x_j^{k+1}) w_{ji}^{k+1}.$$

Once these values are known for each unit, the derivatives with respect to the weights $\frac{\partial E}{\partial w_{ij}^k}$ can easily be calculated using the expression

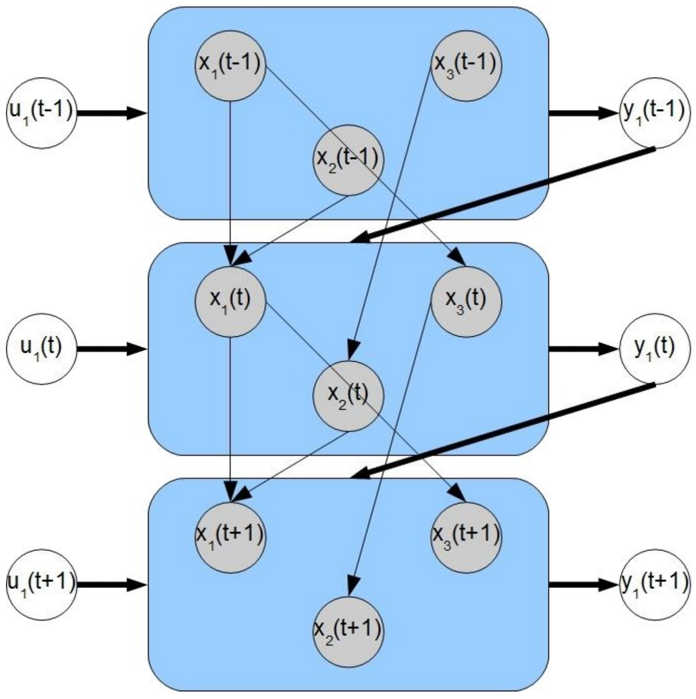
$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial x_i^k} \frac{\partial x_i^k}{\partial w_{ij}^k} = \delta_i^k o_j^{k-1}.$$

Following the same philosophy, BackPropagation Through Time (BPTT) [8] unfolds a recurrent network in temporal steps, so the original recurrent network is transformed into a much larger feedforward network. Then, the classical BP is applied to train the weights of the network.

Conceptually, the first step of the algorithm is to transform the RNN into a MLP. For this task, at every timestep of the training sequence, a new layer is constructed as a copy of the reservoir (the set of internal units of the RNN). The connections between the units of the reservoir are transformed into connections from a layer to the following one. Each layer has also connections from the inputs at the respective time, and produces the corresponding output of the model. This is illustrated in figure 1.4.1.



(a) RNN.



(b) MLP.

Figure 1.4.1: Transformation of a RNN into a MLP, unfolding the network over the timesteps of the training sequence.

Formally, the unfolding of the network consists on the following steps:

- The state of the RNN at time k is represented by the k-th layer.
  - The input $i \in I$ at time k, $x_i(k) = u_i(k)$, is encoded by the value of the i-th input to the k-th layer.
  - The reservoir unit j at time k, $x_j(k)$, is encoded by the value of the j-th internal unit in the k-th layer.
- The dynamic of the resulting MLP is given by

$$x_i(k+1) = x_i^{k+1} = \sum_{j=1}^{N} w_{ij} f(x_j^k) = \sum_{j=1}^{N} w_{ij} f(x_j(k)).$$

Once the RNN has been so converted, the classical BP algorithm is used, resulting in the BPTT method. More precisely, the gradient of the error function with respect to the weights is computed in a recursive way, but this time over the timesteps of the training series, instead of over the layers of the network. This recursion is based on the equation that relates the state in the current timestep with the previous state, i.e. in this discrete model $x(k+1) = Wf(x(k))$.

More exactly, let $\delta_i(k)$ be the derivative of the error with respect to the activation of the j-th unit at time k:

$$\delta_i(k) = \frac{\partial E}{\partial x_i(k)}.$$

Starting with the last step of the series, $k = K$, it is clear that the value of the i-th unit $x_i(K)$ only influences the error in that exact timestep (it has no influence through other units because there are no more timesteps afterwards), so

$$\delta_i(K) = \frac{\partial E}{\partial x_i(K)} = \frac{1}{2}\frac{\partial}{\partial x_i(K)}\sum_{k=1}^{K}\sum_{s\in O}[x_s(k) - y_s(k)]^2 = e_i(K), \tag{1.4.1}$$

where we have $e_i(K) = x_i(K) - y_i(K)$, if $i \in O$, and $e_i(K) = 0$, if $i \notin O$.

In order to define the recursion, the following expression will be used:

$$\begin{aligned}\delta_i(k) &= \frac{\partial E}{\partial x_i(k)} = \frac{\partial^{\mathrm{E}} E}{\partial x_i(k)} + \sum_{j=1}^{N}\frac{\partial E}{\partial x_j(k+1)}\frac{\partial x_j(k+1)}{\partial x_i(k)} \\ &= e_i(k) + \sum_{j=1}^{N}\delta_j(k+1)w_{ji}f'(x_i(k)) \\ &= e_i(k) + f'(x_i(k))\sum_{j=1}^{N}\delta_j(k+1)w_{ji}, \end{aligned} \tag{1.4.2}$$

where the first term $\frac{\partial^{\mathrm{E}} E}{\partial x_i(k)}$ represents the direct effect of $x_i(k)$ in the error (that is why it only appears if the unit is an output of the RNN, $i \in O$), and the second one is the indirect error propagated through the posterior timesteps. In what follows, the notation $\frac{\partial^{\mathrm{E}}}{\partial x_i(k)}$ stands for the explicit partial derivative with respect to $x_i(k)$ [1]. With this equation, the $\delta_i(k)$ can be calculated recursively over all $k = 1, \dots, K$. The gradient of the error with respect to the weights will be then given by

[1] In general, for a function f that depends on a set of variables $\{x, a_1(x), \dots, a_N(x)\}$, the derivative of f w.r.t. x is computed using the chain rule as

$$\frac{\partial f}{\partial x} = \frac{\partial^{\mathrm{E}} f}{\partial x} + \sum_{i=1}^{N}\frac{\partial f}{\partial a_i(x)}\frac{\partial a_i(x)}{\partial x}.$$

In this work, for simplicity, the dependence of the variables is not made explicit most of the times (so, in the example, a_i would be used instead of $a_i(x)$). Sometimes, when the function does not depend explicitly on the variable, the corresponding term $\frac{\partial^{\mathrm{E}} f}{\partial x}$ will be omitted directly.

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \sum_{k=1}^{K} \frac{\partial E}{\partial x_i(k)} \frac{\partial x_i(k)}{\partial w_{ij}} \\ &= \sum_{k=1}^{K} \delta_i(k) \frac{\partial x_i(k)}{\partial w_{ij}} \\ &= \sum_{k=1}^{K} \delta_i(k) f(x_j(k-1)). \end{aligned} \tag{1.4.3}$$

Summarizing, the algorithm follows the steps:

1. Compute a forward pass with the current weights.
2. Compute the δ terms for $k = K$ using
$$\delta_i(K) = e_i(K).$$
3. Compute the δ terms for $k < K$ using
$$\delta_i(k) = e_i(k) + f'(x_i(k)) \sum_{j=1}^{N} \delta_j(k+1) w_{ji}.$$
4. Compute the gradient of the error using
$$\frac{\partial E}{\partial w_{ij}} = \sum_{k=1}^{K} \delta_i(k) f(x_j(k-1)).$$
5. Update the weights using the gradient of the error.

It can be seen that BPTT computes the gradient through backward recursion over time.

Turning out attention to the cost of the algorithm, it is determined by the computation of the gradient. The more complex task is to calculate the $\delta_i(k)$. Each of these terms requires N multiplications (in the summation). This must be done for each unit $i = 1, \ldots, N$ and for every timestep, $k = 1, \ldots, K$, so the complexity of the algorithm comes to be $O(KN^2)$.

1.4.2 Real Time Recurrent Learning

Real Time Recurrent Leaning is the second classical algorithm for Recurrent Neural Networks. It was describe by Williams and Zipser in [9]. It consists on the computation of the exact error gradient in a recursive way. In this case, the derivative of the states with respect to the weights are computed first, using the results of a timestep k to get the derivatives at the next timestep $k + 1$.

Since the initial states $x_i(0)$ of the units are fixed (so they are independent of the weights), the first partials are immediate:

$$\frac{\partial x_i(0)}{\partial w_{ml}} = 0.$$

Given the update equation of the model used $x(k+1) = Wf(x(k))$, that is, $x_i(k+1) = \sum_j w_{ij} f(x_j(k))$, the derivative of the states can easily be computed in a recursive way, resulting in

$$\begin{aligned}\frac{\partial x_i(k+1)}{\partial w_{ml}} &= \delta_{i,m} f(x_l(k)) + \sum_{j=1}^{N} \frac{\partial x_i(k+1)}{\partial x_j(k)} \frac{\partial x_j(k)}{\partial w_{ml}} \\ &= \delta_{i,m} f(x_l(k)) + \sum_{j=1}^{N} w_{ij} f'(x_j(k)) \frac{\partial x_j(k)}{\partial w_{ml}}, \end{aligned} \tag{1.4.4}$$

where $\delta_{i,m}$ is the Kronecker's delta. It should be noticed that the first term corresponds to the explicit dependence of $x_i(k+1)$ on w_{ml} (so it only appears if $m = i$, i.e., w_{ml} connects x_i) and the second one corresponds to the dependence through the previous states (it is the term of the recursion).

Now the gradient of the error can be computed as

$$\begin{aligned}\frac{\partial E}{\partial w_{ml}} &= \sum_{k=1}^{K} \sum_{i=1}^{N} \frac{\partial E}{\partial x_i(k)} \frac{\partial x_i(k)}{\partial w_{ml}} \\ &= \sum_{k=1}^{K} \sum_{i=1}^{N} e_i(k) \frac{\partial x_i(k)}{\partial w_{ml}} \\ &= \sum_{k=1}^{K} \sum_{i \in O} e_i(k) \frac{\partial x_i(k)}{\partial w_{ml}}. \end{aligned} \tag{1.4.5}$$

Although the summation is over all the units, only the terms corresponding to the output units $i \in O$ will appear because $e_i(k) = 0$ for $i \notin O$.

The complete algorithm is:

1. Compute a forward pass with the current weights.
2. Compute the terms $\frac{\partial x_i(k)}{\partial w_{ml}}$ for $k = 0$ using
$$\frac{\partial x_i(0)}{\partial w_{kl}} = 0.$$
3. Compute the terms $\frac{\partial x_i(k)}{\partial w_{ml}}$ for $k = 1, \ldots, K$ using
$$\frac{\partial x_i(k)}{\partial w_{ml}} = \delta_{i,m} f(x_l(k-1)) + \sum_{j=1}^{N} w_{ij} f'(x_j(k-1)) \frac{\partial x_j(k-1)}{\partial w_{ml}}.$$

4. Compute the gradient of the error using

$$\frac{\partial E}{\partial w_{ml}} = \sum_{k=1}^{K} \sum_{i=1}^{N} e_i(k) \frac{\partial x_i(k)}{\partial w_{ml}} = \sum_{k=1}^{K} \sum_{i \in O} e_i(k) \frac{\partial x_i(k)}{\partial w_{ml}}.$$

5. Update the weights using the gradient of the error.

It is observed that RTRL computes the gradient through forward recursion on time. Hence, one advantage of this algorithm is that it can be used in an online way, so the weights are being adapted as the new training patterns are introduced. In this case, the instantaneous error

$$E(k) = \frac{1}{2} \sum_{s \in O} [x_s(k) - y_s(k)]^2$$

is used. The weights at time k are updated with the gradient of $E(k)$, which is given by

$$\frac{\partial E(k)}{\partial w_{ml}} = \sum_{i=1}^{N} e_i(k) \frac{\partial x_i(k)}{\partial w_{ml}}.$$

With respect to costs, the more complex task in the algorithm is the computation of the gradient, i.e., to solve the recursive expression to get the term $\frac{\partial x_i(k+1)}{\partial w_{ml}}$. There are N^3 terms of this form, and each of them require an order of N multiplications, so the total cost is $O(N^4)$. This is the cost for each timestep. If the algorithm is applied in a batch mode, then the total cost for an update is $O(KN^4)$. If it is used online only the instantaneous error gradient is needed, so the complexity is $O(N^4)$ for each update of the weights. So the cost grows with N^4. It scales very bad compared to the cost of BPTT, so it is suitable only for an online use.

1.5 Echo State Networks and Liquid State Machines

The idea of handling differently the weights connecting the reservoir and those that define the output of the model arises simultaneously in two approaches, the Echo State Networks [12] by Jaeger et al, and another one known as Liquid State Machines [13] by Maass et al.

1.5.1 The ESN Algorithm

In this approach, the model is trained in batch mode. The reservoir weights (given by the matrix W^{Res}) are not modified at all. They are fixed at the beginning, usually with a random initialization. Sometimes they are scaled so the speed of the RNN fits the dynamic of the problem, as it will be explained in subsection 1.5.4. Then the original algorithm only trains the output weights w_{si}, with $s \in S$, as it can be viewed in figure 1.5.1.

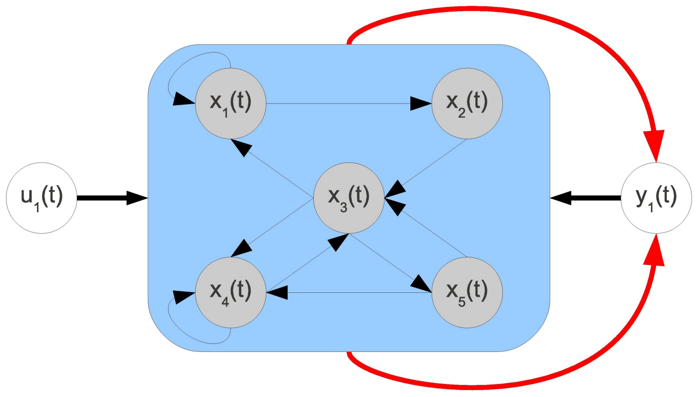


Figure 1.5.1: In the ESN approach, only the output weights (the red ones) are modified.

1.5.2 Algorithm

The first step needed to calculate the output weights is the computation of the inner states in the RNN, using the desired output (this technique is known as *teacher forcing*). So, following the update given in section 1.3 by equation (1.3.1), but fixing the outputs to their desired value, i.e., $x_s(k) = y_s(k)$ for all $s \in S$, the inner states $x(k)$ are stored in a $K \times N$ matrix X, so the k-th row corresponds to the transpose vector $x^T(k)$,

$$X = \begin{pmatrix} x^T(1) \\ \vdots \\ x^T(K) \end{pmatrix} = \begin{pmatrix} x_1(1) & \cdots & x_N(1) \\ \vdots & \ddots & \vdots \\ x_1(K) & \cdots & x_N(K) \end{pmatrix}.$$

It is important to realize that these states are obtained "assuming" that the RNN estimates the output perfectly.

Intuitively, the idea is to compute the states of the reservoir when the RNN is producing the desired output. Once the states are obtained, the mentioned matrix X is used to look for the output weights that reconstruct the desired output with these states.

When the RNN has been trained, the output $s \in O$ at each timestep $k+1$ is computed with the expression

$$x_s(k+1) = [Wf(x(k))]_s = \sum_{j=1}^{N} w_{sj} f(x_j(k)) = w_s^T f(x(k)) = f(x^T(k)) w_s, \qquad (1.5.1)$$

where w_s^T is the $1 \times N$ vector of weights that connect the output s, i.e. the s-th row of W.

On the other hand, the s-th desired output is collected into a $K \times 1$ vector

$$Y_s = \begin{pmatrix} y_s(2) \\ \vdots \\ y_s(K+1) \end{pmatrix}.$$

Formally, the problem is to find the set of output weights $\{w_{si}, s \in S\}$ that minimize the error E given by equation (1.3.4). Each of the outputs can be trained independently, so the problem is equivalent to minimize

$$\|f(X)w_s - Y_s\|^2,$$

where we recall that $f(.)$ is applied componentwise. It should be noticed that the vector Y_s represents the output at timesteps $2, \ldots, K+1$, while X represents the states at timesteps $1, \ldots, K$. This is because the output $x_s(k+1)$ at time $k+1$ is computed using the value of the units in the previous step k (see equation (1.5.1).

The minimization of the previous squared error is just a problem of linear regression, and the solution is simply

$$w_s = O^+ Y_s,$$

where $O = f(X)$ is the $K \times N$ matrix with the outputs of the units (i.e., after applying the activation function) at each timestep, and O^+ denotes the pseudoinverse of O, i.e., $O^+ = (O^T O)^{-1} O^T$. In other words, the problem can be reduced to the minimization of

$$\|Ow_s - Y_s\|^2 = (Ow_s - Y_s)^T (Ow_s - Y_s) = w_s^T O^T O w_s - 2 w_s^T O^T Y_s + Y_s^T Y_s.$$

Taking the gradient with respect to w_s,

$$\nabla_{w_s}(\|Ow_s - Y_s\|^2) = 2O^T O w_s - 2O^T Y_s = 0 \implies w_s = (O^T O)^{-1} O^T Y_s = O^+ Y_s.$$

The final complete algorithm is:

1. Iterate the dynamic of the network, forcing the outputs to the desired states ($x_s(k) = y_s(k)$) for $k = 1, \ldots, K$ and using the expression given by equation (1.3.1),

$$x(k+1) = (1 - \Delta t) x(k) + \Delta t W f(x(k)).$$

2. Collect the states into the matrix X for steps $k_0, \ldots, K$. The first $k_0 - 1$ steps are not used because of their dependence on the initial state of the RNN (this is usually called the washout of the network).

3. Compute the matrix $O = f(X)$, the outputs of the units after the application of the transfer function.

4. For each $s \in S$.

   (a) Collect the desired output of unit s in the vector Y_s for the steps $k_0 + 1, \ldots, K + 1$.

   (b) Compute $w_s^T = O^+ Y_s$.

5. Substitute each w_s in the corresponding row of the weight matrix W.

1.5.3 Mathematical Formalization: Echo States Property

One important characteristic of the RNN (before training it, i.e., independent of the output weights) is the Echo States Property (ESP), defined as:

Definition 1 *[Echo States Property] Assume a RNN with the input vector $u(k)$ and the output vector $y(k)$ from compact subsets U and Y, respectively. The RNN has* echo states *(and therefore satisfies the* echo states property*) with respect to U and Y if for every left-infinity sequence $(u(k), y(k))$, with $k = \dots, -2, -1, 0$, and for all state sequences $x(k)$, $x'(k)$ compatible with the teacher sequence, that is, with*

$$\begin{aligned} x(k+1) &= (1-\Delta t)x(k) + \Delta t W f(x(k)), \quad x_s(k) = y_s(k) \quad \forall s \in S, \\ x'(k+1) &= (1-\Delta t)x'(k) + \Delta t W f(x'(k)), \quad x'_s(k) = y_s(k) \quad \forall s \in S, \end{aligned}$$

it holds that $x(k) = x'(k)$ for all $k \leq 0$.

Clearly, the previous definition does not depend on the output weights, because it is based in the sequence obtained teacher-forcing the RNN.

The intuitive interpretation of the ESP is that the current state of the network, when it has been running during a long time, depends only on the sequence of inputs and the forced outputs. Thus, the RNN forgets its initial state.

The ESP permits to express the present state of the RNN as a function of all the previous inputs and (desired) outputs of the dynamical system being modeled. Usually in engineering problems, any deterministic stationary system that one wants to solve will be assumed to depend only on the previous inputs and outputs. This is because if the unknown initial state has any influence, it could not be captured by the model and it should be obviated. Thus this kind of systems can be in principle represented with such a function of the inputs and desired outputs, so this property gives a way of relating the assumed desired outputs of the system with the real output of the RNN.

There is no known necessary and sufficient condition for the ESP. In [15], it was shown that a sufficient condition for the non existence of echo states is that the spectral radius, that is, the largest absolute eigenvalue of the reservoir weight matrix W^{Res} (the matrix of the connections between the units that are not inputs or outputs, as defined before) is greater than 1, i.e., $|\lambda_{\text{max}}(W^{\text{Res}})| > 1$. This ensures that the RNN has not the ESP w.r.t. any interval $U \times D \ni (\mathbf{0}, \mathbf{0})$. There are also sufficient conditions, such as $\sigma_{\text{max}}(W^{\text{Res}}) < 1$ for $\tanh$ activations, where σ_{max} is the largest singular value [2] of the matrix; or a less restrictive version, $\inf_{D \in \mathcal{D}} \sigma_{\text{max}}(D W^{\text{Res}} D^{-1}) < 1$, with $\mathcal{D}$ the set of diagonal matrices of the proper dimension (see [16] for further details).

[2]A singular value σ and pair of singular vectors u and v of a matrix A are a nonnegative scalar and two nonzero vectors so that

1.5.4 Practical Issues

The first task is to initialize the reservoir. Although there is not an exact method for creating a reservoir that satisfies the ESP, in the practice the following algorithm seems to guarantee this property:

1. Generate a reservoir weight matrix W_0.
2. Normalize it so its spectral radius comes to be 1, $W_1 = 1/|\lambda_{\text{max}}|W_0$.
3. Scale it to an spectral radius $\alpha < 1$, $W = \alpha W_1$.

Then the RNN resulting from this procedure usually behaves as an ESN, independently of the connections with the input and output units.

This α has a very big importance, because the dynamic of the RNN depends on it [15]. When α is close to 1, the reservoir stores the information for a long time, so the RNN produces a slow dynamic. On the other side, if α is small the reservoir is fast, and it will forget quickly the past information. So the selection of α should depend on the timescale of the problem being modeled (see [15], or [17] for an experimental analysis). Moreover, in [18] the processing power of a system working at the edge of stability, that is, where $\alpha \simeq 1$, is studied.

We briefly discuss some practical recommendations for improving ESN training.

Internal States Analysis It is important to check the values of the internal states. Figures of the evolution of the states will provide a visual analysis of the dynamic achieved. Fast oscillations indicates that the spectral radius is too big, and should be scaled down, multiplying the weight matrix by a certain factor smaller than 1 (this is equivalent to choose an smaller α during the construction of the reservoir). When the states take values close to the limits of the activation function (in the case of the $\tanh$, 1 or -1) the reservoir is saturated (due to big inputs or outputs). This can be solved by scaling the input or feedback weights. Nevertheless, this behaviour can be desirable when the desired output oscillates at each step from a given value to another one. In this case, the RNN has to change its state completely in only one step, so it should work in the nonlinear region of the activation function, where it performs almost as a discrete activation.

Model Size The larger the network is, the more complex results it can produce. So difficult problems will require big networks (but the overfitting problem should not be forgotten).

$$\begin{aligned} Av &= \sigma u \\ A^T u &= \sigma v. \end{aligned}$$

The singular values of A coincide with the square root of the eigenvalues of $A^T A$, provided that

$$A^T A v = A^T \sigma u = \sigma A^T u = \sigma^2 v.$$

Selection of the Spectral Radius As it has been mentioned, the spectral radius sets the speed of the network. A big α, $\alpha \approx 1$, implies a slow dynamic, and a small one, $\alpha \approx 0$, a short-memory network.

Addition of Noise When the network is being trained to work in a generative way (that is, forecasting several consecutive steps into the future, usually using a vector of delays as training inputs, and replacing it by the corresponding output for testing), problems of stability may appear. They can be solved by adding noise to the states of the network during sampling [15] (i.e. when the matrix X is computed). This can also be useful when the model suffers from overfitting (however, the best idea in these cases is to reduce the network size, or to try some kind of regularization, as using ridge regression instead of standard linear regression).

Use of a Bias When the output has not zero mean, it is a good idea to use a fixed input ("bias") during both training and test. The model will be kept unaltered, and the algorithms for training and test will be exactly the same, but the data will be modified to add a fixed input $u_r(k) = C, \forall k$. The magnitude of C can change the behaviour of the RNN, so if C is very big it will shift the value of the units to the extremes of the activation function. A small C will not have influence in the nonlinearity degree of the RNN.

Input Range The range of the input values can be shifted and scaled, in order to put it in a region where the network works well for the problem at hand. If the problem is very nonlinear, a range far away from zero can improve the results. The input should also be modified to avoid symmetric problems. This is because, with a standard sigmoid network with a symmetric activation function (such as $\tanh$), if the output is $y(n)$ with an input $u(n)$, another sequence $-u(n)$ will produce $-y(n)$, so problems like $y(n) = u(n)^2$ cannot be solved.

1.5.5 Liquid State Machines

The Liquid State Machine (LSM) approach developed by Maas et al. ([13]) is partially similar to the ESN one. This model was created trying to emulate the principal computational properties of neural microcircuits.

It conceives a reservoir (called "liquid") as a large RNN that will be fixed during all the training. So, it is independent of the problem. As in the ESN approach, the "liquid" is often randomly initialized. The information stored in this RNN is used by a trainable readout mechanism (though dependent of the task), in an analogous way as in the ESN.

The main differences between both approaches are:

- LSM is focused in realistic models, that can represent biological phenomena. Typically the connectivity is topologically and metrically constrained by biological reasons. ESN is conceived more like an engineering tool.

- In the LSM approach, the values of the internal units are assumed to be continuous over time, so the dynamic of the system is usually defined with a differential equation. Thus the system is defined continuously, and a discretization is used to simulate it. In the case of the ESN approach, the system is generally discrete over time, so the values can change completely at each timestep.
- The "liquid" of LSM is made of spiking neurons, whereas the ESN units are typically simple sigmoid ones.
- LSM considers lots of readout mechanism (including feedforward networks), while ESN typically uses a simple linear layer.

LSM used to model spike trains and not get in engineering problems. Thus we will not further consider them more.

Chapter 2

Atiya-Parlos Recurrent Learning

In the year 2000, Atiya and Parlos published a work [10] in which the problem of training a RNN was rewritten under a different approximation. This allows to unify the classical algorithms, and to propose a new one, namely the Atiya-Parlos Recurrent Learning (APRL).

In this chapter, this APRL algorithm and the unification of both BPTT and RTRL using the new approach will be explained.

2.1 Definitions

In this chapter, the model that will be used is the one explained in section 1.3, namely the general one given by (1.3.1),

$$x(k+1) = (1 - \Delta t)x(k) + \Delta t W f(x(k)).$$

2.1.1 Definition of the Problem

From the point of view of Atiya-Parlos [10], the dynamic of the network is not included in the algorithm explicitly, but as a constrain of the optimization problem. Therefore, the objective of the algorithm will be to solve the problem

$$\text{minimize } E \text{ with } g \equiv 0, \tag{2.1.1}$$

where g is the constraint that represents the dynamic of the RNN (see below).

The error E is defined as the squared error of the outputs (denoted with the index set O, as done before) over all the K timesteps, that is,

$$E = \frac{1}{2} \sum_{k=1}^{K} \sum_{s \in O} \left[x_s(k) - y_s(k) \right]^2 .$$

The update of the state of the network $x(t) = (x_1(t), \ldots x_N(t))^T$ (a vector of length N, the number of units) follows the equation

$$x(k+1) = (1-\Delta t)x(k) + \Delta t W f(x(k)).$$

The constraint function g will be satisfied if the dynamic of the network is respected, so

$$g(k+1) = -x(k+1) + (1-\Delta t)x(k) + \Delta t W f(x(k)) = 0. \tag{2.1.2}$$

In terms of the vector components,

$$g_i(k+1) = -x_i(k+1) + (1-\Delta t)x_i(k) + \Delta t \sum_{j=1}^{N} w_{ij} f(x_j(k)) = 0. \tag{2.1.3}$$

2.1.2 Definition of the Virtual Target

One of the principles of APRL is the use of a virtual target. The idea is to calculate the weight changes that produce a desired state change, which is obtained solving the problem of minimizing E with respect to the states of the RNN over all the timesteps. This technique will be formalized next.

For notation, let w_i^T be the rows of W, so w_i is a $N \times 1$ vector. The variables involved are collected into vectors of the form:

$$\begin{aligned}
\mathbf{x} &= (x^T(1), \ldots, x^T(K))^T, \\
\mathbf{g} &= (g^T(1), \ldots, g^T(K))^T, \\
\mathbf{w} &= (w_1^T, \ldots, w_N^T)^T.
\end{aligned}$$

The desired states update will be given by

$$\Delta\mathbf{x} = -\left(\frac{\partial E}{\partial \mathbf{x}}\right)^T = -\begin{pmatrix} \left(\frac{\partial E}{\partial x(1)}\right)^T \\ \vdots \\ \left(\frac{\partial E}{\partial x(K)}\right)^T \end{pmatrix} = -\begin{pmatrix} e^T(1) \\ \vdots \\ e^T(K) \end{pmatrix} \in \mathbb{R}^{NK\times 1},$$

where $e(k)$ is a $1 \times N$ row vector (so $e^T(k)$ is a $N \times 1$ vector) given by

$$e(k) = \frac{\partial E}{\partial x(k)} = \left(\frac{\partial E}{\partial x_1(k)}, \cdots, \frac{\partial E}{\partial x_N(k)}\right) = \Big(e_1(k), \cdots, e_N(k)\Big)$$

and

$$e_s(k) = \begin{cases} x_s(k) - y_s(k), & s \in O, \\ 0, & s \notin O. \end{cases}$$

For posterior notation, we write

$$\frac{\partial E}{\partial \mathbf{x}} = (e(1), \ldots, e(K)). \tag{2.1.4}$$

In order to estimate the change in the weights that will produce the desired change $\frac{\eta}{\Delta t}\Delta\mathbf{x}$ in the states, the constraint $\mathbf{g} = 0$ is used, that is, $\Delta\mathbf{w}$ is estimated as the variation that compensates the constraint when the states move $\Delta\mathbf{x}$:

$$0 = \Delta\mathbf{g} \approx \frac{\partial \mathbf{g}}{\partial \mathbf{w}}\Delta\mathbf{w} + \frac{\eta}{\Delta t}\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\Delta\mathbf{x} \implies \frac{\partial \mathbf{g}}{\partial \mathbf{w}}\Delta\mathbf{w} \approx -\frac{\eta}{\Delta t}\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\Delta\mathbf{x}.$$

Finally, the pseudo-inverse is used to solve the equation, so

$$\Delta\mathbf{w}^{\text{batch}} = -\frac{\eta}{\Delta t}\left[\left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)^T\left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)\right]^{-1}\left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)^T\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\Delta\mathbf{x}. \tag{2.1.5}$$

2.1.3 Dimension Values

The dimensions of the matrices and vectors that will be used are described here:

- $\mathbf{x}$: It is a vector of dimension NK (N units over K timesteps).
- $\Delta\mathbf{x}$: It is a vector of dimension NK (the update of $\mathbf{x}$).
- $\mathbf{g}$: It is a vector of dimension NK, because it is the constraint over $\mathbf{x}$.
- $\mathbf{w}$: It is a vector of dimension N^2, it represents the weights matrix in a single vector.
- $\Delta\mathbf{w}$: It is a vector of dimension N^2 (the update of $\mathbf{w}$).
- E: It is a scalar, the error over all units and all timesteps.
- $\frac{\partial \mathbf{g}}{\partial \mathbf{x}}$: It is a matrix of dimensions $NK \times NK$.
- $\frac{\partial \mathbf{g}}{\partial \mathbf{w}}$: It is a matrix of dimensions $NK \times N^2$.
- $\frac{\partial E}{\partial \mathbf{x}}$: It is a row vector of dimensions $1 \times NK$. It can be decomposed into K vectors of dimensions $1 \times N$ as $\frac{\partial E}{\partial \mathbf{x}} = (e(1), \ldots, e(K))$.
- $\frac{\partial E}{\partial \mathbf{w}}$: It is a row vector of dimensions $1 \times N^2$.

2.2 Atiya-Parlos Batch Recurrent Learning

In the following sections, the algorithm of APRL will be presented. Later (see chapter 3) it will be further analyzed and simplified to arrive to the new Backpropagation Decorrelation (BPDC) online algorithm. First of all, the terms that appear in the right side of equation (2.1.5) will be computed, to get the desired update of the weights $\Delta\mathbf{w}$ that will define the batch algorithm.

Let $f_k = (f(x_1(k)), \ldots, f(x_N(k)))^T$ be the vector of activations at the step k. The partial of the constraint function $\mathbf{g}$ with respect to $\mathbf{w}$ will be given by

$$\begin{aligned}
\frac{\partial \mathbf{g}}{\partial \mathbf{w}} &= \begin{pmatrix} \frac{\partial g(1)}{\partial w_1} & \cdots & \frac{\partial g(1)}{\partial w_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial g(K)}{\partial w_1} & \cdots & \frac{\partial g(K)}{\partial w_N} \end{pmatrix} \\
&= \begin{pmatrix} \frac{\partial g_1(1)}{\partial w_{11}} & \cdots & \frac{\partial g_1(1)}{\partial w_{1N}} & \cdots & \frac{\partial g_1(1)}{\partial w_{N1}} & \cdots & \frac{\partial g_1(1)}{\partial w_{NN}} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{\partial g_N(1)}{\partial w_{11}} & \cdots & \frac{\partial g_N(1)}{\partial w_{1N}} & \cdots & \frac{\partial g_N(1)}{\partial w_{N1}} & \cdots & \frac{\partial g_N(1)}{\partial w_{NN}} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \frac{\partial g_1(K)}{\partial w_{11}} & \cdots & \frac{\partial g_1(K)}{\partial w_{1N}} & \cdots & \frac{\partial g_1(K)}{\partial w_{N1}} & \cdots & \frac{\partial g_1(K)}{\partial w_{NN}} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{\partial g_N(K)}{\partial w_{11}} & \cdots & \frac{\partial g_N(K)}{\partial w_{1N}} & \cdots & \frac{\partial g_N(K)}{\partial w_{N1}} & \cdots & \frac{\partial g_N(K)}{\partial w_{NN}} \end{pmatrix} \in \mathbb{R}^{NK \times N^2}.
\end{aligned}$$

Differentiating in the expression (2.1.3) gives

$$\begin{aligned}
\frac{\partial \mathbf{g}}{\partial \mathbf{w}} &= \begin{pmatrix} f(x_1(0)) & \cdots & f(x_N(0)) & \cdots & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & \cdots & f(x_1(0)) & \cdots & f(x_N(0)) \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ f(x_1(K-1)) & \cdots & f(x_N(K-1)) & \cdots & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots & \cdots & \cdots \\ 0 & \cdots & 0 & \cdots & f(x_1(K-1)) & \cdots & f(x_N(K-1)) \end{pmatrix} \\
&= \begin{pmatrix} f_0^T & & & \\ & f_0^T & & \\ & & \ddots & \\ & & & f_0^T \\ \vdots & \vdots & \vdots & \vdots \\ f_{K-1}^T & & & \\ & f_{K-1}^T & & \\ & & \ddots & \\ & & & f_{K-1}^T \end{pmatrix} \in \mathbb{R}^{NK \times N^2}.
\end{aligned}$$

Using a compact notation,

$$\frac{\partial \mathbf{g}}{\partial \mathbf{w}} = \begin{pmatrix} \text{diag}[f_0]^T \\ \vdots \\ \text{diag}[f_{K-1}]^T \end{pmatrix}. \tag{2.2.1}$$

Notice that, by definition of diag[.] (see section B.2), we have $\mathrm{diag}[f_{K-1}^T] = \mathrm{diag}[f_{K-1}]^T$.

Let $C_K = \sum_{k=0}^{K-1} f_k f_k^T$ be an approximation of the auto-correlation matrix (except for a constant) of the activations (assuming they are centered at zero) over the steps $1, \dots, K-1$. From the expression above, it is easy to see that

$$
\begin{aligned}
\left[\left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)^T \left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)\right] &= \begin{pmatrix} \mathrm{diag}[f_0] & \cdots & \mathrm{diag}[f_{K-1}] \end{pmatrix} \begin{pmatrix} \mathrm{diag}[f_0]^T \\ \vdots \\ \mathrm{diag}[f_{K-1}]^T \end{pmatrix} \\
&= \begin{pmatrix} C_K & & & \\ & C_K & & \\ & & \ddots & \\ & & & C_K \end{pmatrix} \\
\Longrightarrow \left[\left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)^T \left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)\right]^{-1} &= \begin{pmatrix} C_K^{-1} & & & \\ & C_K^{-1} & & \\ & & \ddots & \\ & & & C_K^{-1} \end{pmatrix} \in \mathbb{R}^{N^2 \times N^2}.
\end{aligned}
$$

In the same way it is calculated the derivative with respect to the states:

$$
\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial g(1)}{\partial x(1)} & \cdots & \frac{\partial g(1)}{\partial x(K)} \\ \vdots & \ddots & \vdots \\ \frac{\partial g(K)}{\partial x(1)} & \cdots & \frac{\partial g(K)}{\partial x(K)} \end{pmatrix} \in \mathbb{R}^{NK \times NK}.
$$

The following equalities are obtained from the equation (2.1.3):

$$
\begin{aligned}
\frac{\partial g_i(k+1)}{\partial x_i(k+1)} &= -1, \\
\frac{\partial g_i(k+1)}{\partial x_j(k+1)} &= 0, \quad i \neq j, \\
\frac{\partial g_i(k+1)}{\partial x_i(k)} &= \Delta t w_{ii} f'(x_i(k)) + (1 - \Delta t), \\
\frac{\partial g_i(k+1)}{\partial x_j(k)} &= \Delta t w_{ij} f'(x_j(k)), \quad i \neq j, \\
\frac{\partial g_i(l)}{\partial x_j(k)} &= 0, \quad l \neq k, k+1.
\end{aligned}
$$

Expressed in vector notation, the preceding equations become

$$\begin{aligned}
\frac{\partial g(k)}{\partial x(k)} &= -I_{N\times N}, \\
\frac{\partial g(k+1)}{\partial x(k)} &= \Delta t \begin{pmatrix} w_{11}f'(x_1(k)) & \cdots & w_{1N}f'(x_N(k)) \\ \vdots & \ddots & \vdots \\ w_{N1}f'(x_1(k)) & \cdots & w_{NN}f'(x_N(k)) \end{pmatrix} + (1-\Delta t)I_{N\times N} \\
&= \Delta t A_k + (1-\Delta t)I_{N\times N},
\end{aligned}$$

where A_k is defined to simplify the notation as

$$A_k = \begin{pmatrix} w_{11}f'(x_1(k)) & \cdots & w_{1N}f'(x_N(k)) \\ \vdots & \ddots & \vdots \\ w_{N1}f'(x_1(k)) & \cdots & w_{NN}f'(x_N(k)) \end{pmatrix}.$$

So, using the above matrix notation, we have

$$\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \begin{pmatrix} -I_{N\times N} & 0 & \cdots & 0 & 0 \\ \Delta t A_1 + (1-\Delta t)I_{N\times N} & -I_{N\times N} & \cdots & 0 & 0 \\ 0 & \Delta t A_2 + (1-\Delta t)I_{N\times N} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & \Delta t A_{K-1} + (1-\Delta t)I_{N\times N} & -I_{N\times N} \end{pmatrix}. \quad (2.2.2)$$

For the weights update in (2.1.5), the product $-\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\Delta \mathbf{x}$, denoted by γ, is needed, so

$$\begin{aligned}
-\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\Delta \mathbf{x} &= \begin{pmatrix} \frac{\partial g(1)}{\partial x(1)} & \cdots & \frac{\partial g(1)}{\partial x(K)} \\ \vdots & \ddots & \vdots \\ \frac{\partial g(K)}{\partial x(1)} & \cdots & \frac{\partial g(K)}{\partial x(K)} \end{pmatrix} \begin{pmatrix} e^T(1) \\ \vdots \\ e^T(K) \end{pmatrix} \\
&= \begin{pmatrix} \gamma(1) \\ \vdots \\ \gamma(K) \end{pmatrix} \in \mathbb{R}^{NK\times 1},
\end{aligned}$$

where $\gamma(k) = (\gamma_1(k), \cdots, \gamma_N(k))^T$ is given by

$$\gamma(k) = -e^T(k) + (1-\Delta t)e^T(k-1) + \Delta t A_{k-1} e^T(k-1).$$

Combining this result with the formula for $\frac{\partial \mathbf{g}}{\partial \mathbf{w}}$, the following equality is obtained

$$
\begin{aligned}
-\left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)^T \frac{\partial \mathbf{g}}{\partial \mathbf{x}} \Delta \mathbf{x} &= \begin{pmatrix} [f_0] & & \cdots & [f_{K-1}] & & \\ & \ddots & \cdots & & \ddots & \\ & & [f_0] & \cdots & & [f_{K-1}] \end{pmatrix}^T \begin{pmatrix} \gamma(1) \\ \vdots \\ \gamma(K) \end{pmatrix} \\
&= \begin{pmatrix} \sum_{k=0}^{K-1} f_k \gamma_1(k+1) \\ \vdots \\ \sum_{k=0}^{K-1} f_k \gamma_N(k+1) \end{pmatrix} \in \mathbb{R}^{N^2 \times 1},
\end{aligned}
$$

where $f_k\gamma_i(k+1)$ is a $N \times 1$ vector that results from the product of the vector f_k and the scalar $\gamma_i(k+1)$.

Taking into account the term obtained for $\left[\left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)^T \left(\frac{\partial \mathbf{g}}{\partial \mathbf{w}}\right)\right]$, the batch weights update becomes

$$
\begin{aligned}
\Delta \mathbf{w}^{\text{batch}} &= \frac{\eta}{\Delta t} \begin{pmatrix} C_K^{-1} & & & \\ & C_K^{-1} & & \\ & & \ddots & \\ & & & C_K^{-1} \end{pmatrix} \begin{pmatrix} \sum_{k=0}^{K-1} f_k \gamma_1(k+1) \\ \vdots \\ \sum_{k=0}^{K-1} f_k \gamma_N(k+1) \end{pmatrix} \\
&= \frac{\eta}{\Delta t} \begin{pmatrix} C_K^{-1} \sum_{k=0}^{K-1} f_k \gamma_1(k+1) \\ \vdots \\ C_K^{-1} \sum_{k=0}^{K-1} f_k \gamma_N(k+1) \end{pmatrix} \in \mathbb{R}^{N^2 \times 1}.
\end{aligned}
$$

In terms of each component of $\mathbf{w}$,

$$
\Delta w_{ij}^{\text{batch}}(K) = \frac{\eta}{\Delta t} \left[C_K^{-1} \sum_{k=0}^{K-1} f_k \gamma_i(k+1) \right]_j . \qquad (2.2.3)
$$

It should be noticed that the update formulae for $\Delta \mathbf{w}^{\text{batch}}$ can be computed in a recursive form. More precisely, we change from the vector notation for the weights $\mathbf{w}$ to the matrix notation W defined in section 1.3). We then have

- The regularized version $\hat{C}_K = \sum_{k=0}^{K-1} f_k f_k^T + \epsilon I_{N \times N}$ of the matrix C_K (to avoid numerical problems with its inversion) can be expressed in terms of $\hat{C}_{K-1}$ as

$$
\hat{C}_K = \hat{C}_{K-1} + f_{K-1} f_{K-1}^T .
$$

- From this last formula and the small rank adjustment matrix inversion lemma (see appendix A.2.2), the inverse of $\hat{C}_K$ can be expressed as

$$
\hat{C}_K^{-1} = \hat{C}_{K-1}^{-1} - \frac{\left[\hat{C}_{K-1}^{-1} f_{K-1}\right] \left[\hat{C}_{K-1}^{-1} f_{K-1}\right]^T}{1 + f_{K-1}^T \hat{C}_{K-1}^{-1} f_{K-1}} . \qquad (2.2.4)
$$

- For the other term of the update equation, let $B(K) = \sum_{k=0}^{K-1} \gamma(k+1) f_k^T$. The order has been reversed to express the update as a matrix of dimensions $N \times N$, instead of a $N^2 \times 1$ vector. More precisely, in scalar terms we have $B(K)_{ij} = \sum_{k=0}^{K-1} \gamma_i(k+1)[f_k^T]_j$. With this definition, it is clear that

$$B(K) = B(K-1) + \gamma(K) f_{K-1}^T.$$

So the update (ignoring the effect of ϵ) can be expressed as the matrix

$$\Delta W^{\text{Batch}} = \frac{\eta}{\Delta t} B(K) \hat{C}_K^{-1}.$$

In fact, notice that at the (i, j) component, the right hand side matrix gives

$$\begin{aligned}
[\Delta W^{\text{Batch}}]_{ij} &= \frac{\eta}{\Delta t} \left[B(K) \hat{C}_K^{-1} \right]_{ij} \\
&= \frac{\eta}{\Delta t} \sum_{l=1}^{N} [B(K)]_{il} [\hat{C}_K^{-1}]_{lj} \\
&= \frac{\eta}{\Delta t} \sum_{l=1}^{N} [\sum_{k=0}^{K-1} \gamma(k+1) f_k^T]_{il} [\hat{C}_K^{-1}]_{lj} \\
&= \frac{\eta}{\Delta t} \sum_{l=1}^{N} \sum_{k=0}^{K-1} \gamma_i(k+1) [f_k^T]_l [\hat{C}_K^{-1}]_{lj},
\end{aligned}$$

which coincides with (2.2.3) since we have

$$\begin{aligned}
\Delta w_{ij}^{\text{batch}}(K) &= \frac{\eta}{\Delta t} \left[C_K^{-1} \sum_{k=0}^{K-1} f_k \gamma_i(k+1) \right]_j \\
&= \frac{\eta}{\Delta t} \sum_{l=1}^{N} [C_K^{-1}]_{jl} \sum_{k=0}^{K-1} [f_k]_l \gamma_i(k+1) \\
&= \frac{\eta}{\Delta t} \sum_{l=1}^{N} [C_K^{-1}]_{lj} \sum_{k=0}^{K-1} [f_k]_l \gamma_i(k+1),
\end{aligned}$$

where the symmetry of C_K^{-1} has been used.

2.3 Unification of BPTT and RTRL

In this section the classical algorithms of BPTT and RTRL will be unified in terms of the constraint problem defined by (2.1.1). Here we will consider $\Delta t = 1$, i.e., the discrete model of the RNN, as this is the one used by BPTT and RTRL. For this task, the trained network (that is, the weights that minimize the error function satisfying the mentioned constraint) is found using gradient descent over the weights. These two algorithms differ in the way in which this gradient is computed, but both of them use the following two formulae in the reformulation of Atiya-Parlos:

- For the derivative of the error,

$$\frac{\partial E(\mathbf{x}(\mathbf{w}))}{\partial \mathbf{w}} = \frac{\partial^{\mathrm{E}} E(\mathbf{x}(\mathbf{w}))}{\partial \mathbf{w}} + \frac{\partial E(\mathbf{x}(\mathbf{w}))}{\partial \mathbf{x}}\frac{\partial \mathbf{x}(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial E(\mathbf{x}(\mathbf{w}))}{\partial \mathbf{x}}\frac{\partial \mathbf{x}(\mathbf{w})}{\partial \mathbf{w}}, \tag{2.3.1}$$

  where the first term equals zero because of the absence of an explicit dependence.

- For the derivative of the constraint,

$$\begin{aligned} 0 = \frac{\partial \mathbf{g}(\mathbf{w},\mathbf{x}(\mathbf{w}))}{\partial \mathbf{w}} &= \frac{\partial^{\mathrm{E}} \mathbf{g}(\mathbf{w},\mathbf{x}(\mathbf{w}))}{\partial \mathbf{w}} + \frac{\partial \mathbf{g}(\mathbf{w},\mathbf{x}(\mathbf{w}))}{\partial \mathbf{x}}\frac{\partial \mathbf{x}(\mathbf{w})}{\partial \mathbf{w}} \implies \\ \frac{\partial \mathbf{x}(\mathbf{w})}{\partial \mathbf{w}} &= -\left(\frac{\partial \mathbf{g}(\mathbf{w},\mathbf{x}(\mathbf{w}))}{\partial \mathbf{x}}\right)^{-1} \frac{\partial^{\mathrm{E}} \mathbf{g}(\mathbf{w},\mathbf{x}(\mathbf{w}))}{\partial \mathbf{w}}, \end{aligned} \tag{2.3.2}$$

  where it has been used that $g(\mathbf{w},\mathbf{x}(\mathbf{w})) = 0$.

Substituting (2.3.2) into (2.3.1), and with a simplified notation, the following equation for the error gradient is obtained:

$$\begin{aligned} \frac{\partial E}{\partial \mathbf{w}} &= \frac{\partial E}{\partial \mathbf{x}}\frac{\partial \mathbf{x}}{\partial \mathbf{w}} \\ &= -\frac{\partial E}{\partial \mathbf{x}}\left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\right)^{-1}\frac{\partial \mathbf{g}}{\partial \mathbf{w}}. \end{aligned} \tag{2.3.3}$$

2.3.1 BackPropagation Through Time

In this approach, the term $\hat{\delta} = \frac{\partial E}{\partial \mathbf{x}}\left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\right)^{-1}$, a vector of size $1 \times NK$, is evaluated first, and then used to get the desired gradient. By definition,

$$\frac{\partial E}{\partial \mathbf{x}} = \frac{\partial E}{\partial \mathbf{x}}\left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\right)^{-1}\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \hat{\delta}\frac{\partial \mathbf{g}}{\partial \mathbf{x}}.$$

Denoting the $1 \times N$ components of $\hat{\delta}$ by $\hat{\delta}(k)$, i.e.

$$\hat{\delta} = \begin{pmatrix} \hat{\delta}(1) & \cdots & \hat{\delta}(K) \end{pmatrix},$$

and using the formulae (2.1.4) and (2.2.2), the expression becomes

$$\Big(e(1),\cdots,e(K)\Big) =$$
$$\Big(\hat\delta(1),\cdots,\hat\delta(K)\Big)\begin{pmatrix} -I_{N\times N} & 0 & 0 & \cdots & 0 & 0 \\ A_1 & -I_{N\times N} & 0 & \cdots & 0 & 0 \\ 0 & A_2 & -I_{N\times N} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_{K-1} & -I_{N\times N} \end{pmatrix}.$$

Notice than now $\Delta t = 1$ and we drop the identity matrix in A_k. Therefore we have

$$\begin{aligned} e(K) &= -\hat\delta(K), \\ e(k) &= -\hat\delta(k) + \hat\delta(k+1)A_k, \quad k = 1,\dots,K-1. \end{aligned}$$

or, equivalently,

$$\begin{aligned} \hat\delta(K) &= -e(K), \\ \hat\delta(k) &= -e(k) + \hat\delta(k+1)A_k, \quad k = 1,\dots,K-1. \end{aligned}$$

And $\hat\delta$ can be computed iterating the above formulae backwards in time. If we develop the last formulae at the $\hat\delta_i(k)$ component level we obtain

$$\begin{aligned} \hat\delta_i(K) &= -e_i(K), \\ \hat\delta_i(k) &= -e_i(k) + \sum_{j=1}^{N}[\hat\delta(k+1)]_j[A_k]_{ji} \\ &= -e_i(k) + \sum_{j=1}^{N}\hat\delta_j(k+1)w_{ji}f'(x_i(k)), \end{aligned}$$

where the first term coincides with (1.4.1) and the second expression with (1.4.2), up to a minus sign. So $\hat\delta = -\delta$, where δ is defined as in the classical BPTT algorithm.

Once the term $\hat\delta$ is known, the gradient of the error with respect to the weights can be computed using (2.3.3) and the expression for $\frac{\partial \mathbf{g}}{\partial \mathbf{w}}$ given by (2.2.1), so

$$\frac{dE}{d\mathbf{w}} = -\hat\delta\frac{\partial \mathbf{g}}{\partial \mathbf{w}} = -\hat\delta\begin{pmatrix} \text{diag}[f_0]^T \\ \vdots \\ \text{diag}[f_{K-1}]^T \end{pmatrix} = -\sum_{k=1}^{K}\tilde\delta(k)\text{diag}[f_{k-1}]^T.$$

That is exactly the formula used in BPTT (see subsection 1.4.1) expressed in a vectorial form, as it can be seen, focusing our attention in the component $(i-1)N + j$, that corresponds to the update of w_{ij}:

$$\begin{aligned}
\left[\frac{dE}{d\mathbf{w}}\right]_{(i-1)N+j} &= -\left[\sum_{k=1}^{K}\tilde{\delta}(k)\mathrm{diag}[f_{k-1}]^T\right]_{(i-1)N+j} \\
&= -\sum_{l=1}^{N}\sum_{k=1}^{K}[\tilde{\delta}(k)]_l[\mathrm{diag}[f_{k-1}]^T]_{l,(i-1)N+j} \\
&= -\sum_{l=1}^{N}\sum_{k=1}^{K}\tilde{\delta}_l(k)\begin{pmatrix} f_{k-1}^T & & & \\ & f_{k-1}^T & & \\ & & \ddots & \\ & & & f_{k-1}^T \end{pmatrix}_{l,(i-1)N+j} \\
&= -\sum_{k=1}^{K}\tilde{\delta}_i(k)[f_{k-1}]_j \\
&= \sum_{k=1}^{K}\delta_i(k)[f_{k-1}]_j,
\end{aligned}$$

which is exactly the expression (1.4.3), where the change of sign compensates the one of the $\hat{\delta}$ expression.

2.3.2 Real Time Recurrent Learning

For this algorithm, the first step is to compute the matrix $Z = \left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\right)^{-1}\frac{\partial \mathbf{g}}{\partial \mathbf{w}}$ of dimensions $NK \times N^2$. As before, it is clear that

$$\frac{\partial \mathbf{g}}{\partial \mathbf{w}} = \frac{\partial \mathbf{g}}{\partial \mathbf{x}}\left(\frac{\partial \mathbf{g}}{\partial \mathbf{x}}\right)^{-1}\frac{\partial \mathbf{g}}{\partial \mathbf{w}} = \frac{\partial \mathbf{g}}{\partial \mathbf{x}}Z.$$

Again denoting the $N \times N^2$ blocks of Z as $Z(k)$,

$$Z = \begin{pmatrix} Z(1) \\ \vdots \\ Z(K) \end{pmatrix},$$

and using the equalities (2.2.1) and (2.2.2) into the expression $\frac{\partial \mathbf{g}}{\partial \mathbf{w}} = \frac{\partial \mathbf{g}}{\partial \mathbf{x}}Z$, the following result is obtained:

$$\begin{pmatrix} \mathrm{diag}[f_0]^T \\ \vdots \\ \mathrm{diag}[f_{K-1}]^T \end{pmatrix} = \begin{pmatrix} -I_{N\times N} & 0 & 0 & \cdots & 0 & 0 \\ A_1 & -I_{N\times N} & 0 & \cdots & 0 & 0 \\ 0 & A_2 & -I_{N\times N} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & A_{K-1} & -I_{N\times N} \end{pmatrix}\begin{pmatrix} Z(1) \\ \vdots \\ Z(K) \end{pmatrix}.$$

From this expression, the following formulae for Z are obtained:

$$\begin{aligned}\text{diag}[f_0]^T &= -Z(1),\\ \text{diag}[f_k]^T &= A_k Z(k) - Z(k+1), \quad k = 1, \dots, K-1.\end{aligned}$$

or, equivalently,

$$\begin{aligned}Z(1) &= -\text{diag}[f_0]^T,\\ Z(k+1) &= A_k Z(k) - \text{diag}[f_k]^T, \quad k = 1, \dots, K-1.\end{aligned}$$

This is the classical recursive expression for RTRL (see section 1.4.2), for in fact we have

$$\frac{\partial x_i(k)}{\partial w_{kl}} = -\left(Z(k)\right)_{i,(k-1)N+l},$$

i.e., $Z(k)$ is the minus derivative of the states at time k with respect to the weights. This is clear if we develop the right side of the expressions and take the corresponding $(i, (k-1)N+l)$ component:

$$\begin{aligned}\frac{\partial x_i(1)}{\partial w_{kl}} = [Z(1)]_{i,(k-1)N+l} &= -[\text{diag}[f_0]^T]_{i,(k-1)N+l}\\ &= -\begin{pmatrix} f_0^T & & & \\ & f_0^T & & \\ & & \ddots & \\ & & & f_0^T \end{pmatrix}_{i,(k-1)N+l}\\ &= -\delta_{ik}[f_0]_l.\end{aligned}$$

Where δ_{ik} stands for Kornoecker's delta. For the other terms we have:

$$\begin{aligned}\frac{\partial x_i(k+1)}{\partial w_{kl}} = [Z(k+1)]_{i,(k-1)N+l} &= [A_k Z(k) - \text{diag}[f_k]^T]_{i,(k-1)N+l}\\ &= \sum_{l=1}^{N} [A_k]_{i,l}[Z(k)]_{l,(k-1)N+l} - \delta_{ik}[f_k]_l\\ &= \sum_{l=1}^{N} w_{i,l}[f_k']_l - \frac{\partial x_l(k)}{\partial w_{kl}}\delta_{ik}[f_k]_l.\end{aligned}$$

Once Z has been computed, using (2.3.3) and the expression (2.1.4) for $\frac{\partial E}{\partial \mathbf{x}}$, the gradient becomes

$$\frac{dE}{d\mathbf{w}} = \frac{\partial E}{\partial \mathbf{x}} Z = (e(1), \dots, e(K))Z = \sum_{k=1}^{K} e(k)Z(k).$$

Turning the attention to the component $(i-1)N+j$ that represents the update of w_{ij}, we get the formula

$$\begin{aligned}
\left[\frac{dE}{d\mathbf{w}}\right]_{(i-1)N+j} &= \left[\sum_{k=1}^{K} e(k)Z(k)\right]_{(i-1)N+j} \\
&= \sum_{l=1}^{N}\sum_{k=1}^{K}[e(k)]_l[Z(k)]_{l,(i-1)N+j} \\
&= \sum_{l=1}^{N}\sum_{k=1}^{K} e_l(k)\frac{\partial x_l(k)}{\partial w_{ij}} \\
&= \sum_{k=1}^{K}\sum_{l\in O} e_l(k)\frac{\partial x_l(k)}{\partial w_{ij}},
\end{aligned}$$

which is the same as obtained in (1.4.5) with the classical approach.

Chapter 3

BackPropagation Decorrelation

The algorithm of BPDC was developed by Steil at [11]. It emerges as an evolution of the online version of APRL. In that paper, it was presented as an algorithm based on three basic principles:

- The differentiation of the error function with respect to the states (not the weights, as usually) in order to obtain a "virtual teacher" target. This new objective will be used to update the weights.

  This idea was introduced by Atiya and Parlos (see chapter 2), and it has been claimed that their technique (APRL) outperforms the classical BPTT algorithm.

- The use of the temporal memory of the network dynamic.

- The use of the recurrent neural network as a kind of dynamic reservoir. Almost the whole net is considered as a mechanism that projects the temporal series into a high dimensional space, storing temporal information. This reservoir is not adapted in order to reduce the complexity of the algorithm, and permits an online training with linear complexity with respect to the number of units of the network.

  As it has been said before, this vision emerged simultaneously as the ESN [12] approach and in the LSM one [13].

In this chapter the BPDC algorithm will be analyzed. First, the online version of APRL, and some considerations about its behaviour will motivate the changes that transforms APRL into BPDC. Then, the BPDC algorithm will be explained, and some considerations about the mentioned relation with the ESN approach will be given. Finally, a brief introduction to plasticity methods for reservoir architecture selection will be presented.

3.1 From APRL to BPDC

In this section, some considerations about the APRL algorithm (the basis of BPDC) will be included. First, the online version of APRL will be explained. Then, an analysis of the weight dynamics will support the idea of the functional separation between the output weights and the reservoir ones.

3.1.1 APRL Online Algorithm

For the online version of the APRL algorithm, at timestep $k+1$, the batch weight update is divided into the update up to step k and the new online update $\Delta w_{ij}(k+1)$, so

$$\begin{aligned}
\Delta w_{ij}(k+1) &= \Delta w_{ij}^{\text{batch}}(k+1) - \Delta w_{ij}^{\text{batch}}(k) \\
&= \frac{\eta}{\Delta t}\left[\hat{C}_{k+1}^{-1}\sum_{r=0}^{k} f_r\gamma_i(r+1)\right]_j - \frac{\eta}{\Delta t}\left[\hat{C}_{k}^{-1}\sum_{r=0}^{k-1} f_r\gamma_i(r+1)\right]_j \\
&= \frac{\eta}{\Delta t}\left[\hat{C}_{k+1}^{-1} f_k\right]_j \gamma_i(k+1) + \frac{\eta}{\Delta t}\left[\left(\hat{C}_{k+1}^{-1} - \hat{C}_{k}^{-1}\right)\sum_{r=0}^{k-1} f_r\gamma_i(r+1)\right]_j .
\end{aligned}$$

This expression can be rewritten to see better the relation with the batch update,

$$\begin{aligned}
\Delta w_{ij}(k+1) &= \frac{\eta}{\Delta t}\left[\hat{C}_{k+1}^{-1} f_k\right]_j \gamma_i(k+1) + \frac{\eta}{\Delta t}\left[\left(\hat{C}_{k+1}^{-1}\hat{C}_k\hat{C}_{k}^{-1} - \hat{C}_{k}^{-1}\right)\sum_{r=0}^{k-1} f_r\gamma_i(r+1)\right]_j \\
&= \frac{\eta}{\Delta t}\left[\hat{C}_{k+1}^{-1} f_k\right]_j \gamma_i(k+1) + \frac{\eta}{\Delta t}\left[\left(\hat{C}_{k+1}^{-1}\hat{C}_k - I\right)\hat{C}_{k}^{-1}\sum_{r=0}^{k-1} f_r\gamma_i(r+1)\right]_j .
\end{aligned} \tag{3.1.1}$$

The second term corresponds to $\Delta w_{ij}^{\text{batch}}(k)$ times the factor $\left(\hat{C}_{k+1}^{-1}\hat{C}_k - I\right)$. It is easy to see that this factor decays to zero, since

$$\lim_{k\to\infty}\hat{C}_{k+1}^{-1}\hat{C}_k = \lim_{k\to\infty}\frac{k+1}{k}\hat{C}_{k+1}^{-1}\hat{C}_k = I,$$

where it has been used that $\lim_{k\to\infty}\frac{1}{k}C_k = C$, the autocorrelation matrix of the activations. So the update consists on a mixture of a one-step error term, and a momentum term. The fading of the second term implies that this algorithm cannot include information from a long time back.

Using the same matrix notation of the previous chapter, the online update matrix $\Delta W(K+1)$ is given by

$$\begin{aligned}
\Delta W(K+1) &= \Delta W^{\text{Batch}}(K+1) - \Delta W^{\text{Batch}}(K) \\
&= \frac{\eta}{\Delta t}B(K+1)\hat{C}_{K+1}^{-1} - \frac{\eta}{\Delta t}B(K)\hat{C}_{K}^{-1} \\
&= \frac{\eta}{\Delta t}\left(\left[B(K) + \gamma(K+1)f_K^T\right]\hat{C}_{K+1}^{-1} - B(K)\hat{C}_{K}^{-1}\right) \\
&= \frac{\eta}{\Delta t}\left(B(K)\hat{C}_{K+1}^{-1} + \gamma(K+1)f_K^T\hat{C}_{K+1}^{-1} - B(K)\hat{C}_{K}^{-1}\right),
\end{aligned}$$

where again the terms $B(K)\left(\hat{C}_{K+1}^{-1} - \hat{C}_K^{-1}\right)$ vanishes when $k \to \infty$.

Substituting the recursive expression of $\hat{C}_{K+1}^{-1} = \hat{C}_K^{-1} - \frac{[\hat{C}_K^{-1} f_K][\hat{C}_K^{-1} f_K]^T}{1+f_K^T \hat{C}_K^{-1} f_K}$ given by equation (2.2.4), we obtain:

$$\begin{aligned}
\Delta W(K+1) &= \frac{\eta}{\Delta t}\left(B(K)\hat{C}_K^{-1} - B(K)\frac{\left[\hat{C}_K^{-1} f_K\right]\left[\hat{C}_K^{-1} f_K\right]^T}{1+f_K^T\hat{C}_K^{-1} f_K} \right. \\
&\quad \left. +\gamma(K+1) f_K^T \hat{C}_K^{-1} - \gamma(K+1) f_K^T \frac{\left[\hat{C}_K^{-1} f_K\right]\left[\hat{C}_K^{-1} f_K\right]^T}{1+f_K^T\hat{C}_K^{-1} f_K} - B(K)\hat{C}_K^{-1} \right) \\
&= \frac{\eta}{\Delta t}\left(-\frac{\left[B(K)\hat{C}_K^{-1} f_K\right]\left[\hat{C}_K^{-1} f_K\right]^T}{1+f_K^T\hat{C}_K^{-1} f_K} + \frac{\gamma(K+1) f_K^T\hat{C}_K^{-1}\left[1+f_K^T\hat{C}_K^{-1} f_K\right]}{1+f_K^T\hat{C}_K^{-1} f_K} \right. \\
&\quad \left. -\frac{\left[\gamma(K+1) f_K^T\hat{C}_K^{-1} f_K\right]\left[\hat{C}_K^{-1} f_K\right]^T}{1+f_K^T\hat{C}_K^{-1} f_K} \right) \\
&= \frac{\eta}{\Delta t}\left(-\frac{\left[B(K)\hat{C}_K^{-1} f_K\right]\left[\hat{C}_K^{-1} f_K\right]^T}{1+f_K^T\hat{C}_K^{-1} f_K} + \frac{\gamma(K+1) f_K^T\hat{C}_K^{-1}}{1+f_K^T\hat{C}_K^{-1} f_K} \right. \\
&\quad \left. +\frac{\gamma(K+1) f_K^T\hat{C}_K^{-1} f_K^T\hat{C}_K^{-1} f_K}{1+f_K^T\hat{C}_K^{-1} f_K} - \frac{\left[\gamma(K+1) f_K^T\hat{C}_K^{-1} f_K\right]\left[\hat{C}_K^{-1} f_K\right]^T}{1+f_K^T\hat{C}_K^{-1} f_K} \right) \\
&= \frac{\eta}{\Delta t}\left(\frac{\gamma(K+1)\left[\hat{C}_K^{-1} f_K\right]^T}{1+f_K^T\hat{C}_K^{-1} f_K} - \frac{\left[B(K)\hat{C}_K^{-1} f_K\right]\left[\hat{C}_K^{-1} f_K\right]^T}{1+f_K^T\hat{C}_K^{-1} f_K} \right. \\
&\quad \left. +\frac{\gamma(K+1) f_K^T\hat{C}_K^{-1} f_K^T\hat{C}_K^{-1} f_K}{1+f_K^T\hat{C}_K^{-1} f_K} - \frac{\left[\gamma(K+1) f_K^T\hat{C}_K^{-1} f_K\right]\left[\hat{C}_K^{-1} f_K\right]^T}{1+f_K^T\hat{C}_K^{-1} f_K} \right) \\
&= \frac{\eta}{\Delta t}\frac{\left[\gamma(K+1) - B(K)\hat{C}_K^{-1} f_K^T\right]\left[\hat{C}_K^{-1} f_K\right]^T}{1+f_K^T\hat{C}_K^{-1} f_K},
\end{aligned} \tag{3.1.2}$$

where it has been used that $\hat{C}_K^{-1} = (\hat{C}_K^{-1})^T$ (it is symmetric by definition) and that the last two terms cancel because $f_K^T\hat{C}_K^{-1} f_K$ is a scalar.

3.1.2 Weight Dynamics

In this section we will demonstrate the following lemma of Schiller and Steil [19], which states that, in the single output case, the APRL weights connecting the reservoir are strongly coupled.

Lemma 1 *Assume a network with a single output $i = 1$. Then the update given by APRL satisfies the equation*

$$\forall k, \forall i > 1, \forall j > 1, \forall h : \Delta w_{ih}(k) = \frac{w_{i1}(0)}{w_{j1}(0)} \Delta w_{jh}(k) = s_{ij} \Delta w_{jh}(k), \tag{3.1.3}$$

where s_{ij} is defined as the rate between the initial weights to unit i and j, $s_{ij} = \frac{w_{i1}(0)}{w_{j1}(0)}$.

Proof In order to demonstrate the previous result, the following series of equalities will be proved for $i > 1, j > 1, k$:

- $\gamma_i(k) = s_{ij}\gamma_j(k)$,
- $B_{ih}(k) = s_{ij}B_{jh}(k)$,
- $\Delta w_{ih}(k) = s_{ij}\Delta w_{jh}(k)$,
- $w_{i1}(k) = s_{ij}w_{j1}(k)$.

Induction on k will be used for the demonstration. To begin with, in the case $k = 1$, the fourth term is just the definition of s_{ij} and the first three terms are in fact 0 and we trivially have:

$$\begin{aligned}
\gamma_i(1) &= -e_i(1) = 0 = -e_j(1) = \gamma_j(1) = s_{ij}\gamma_j(1). \\
B_{ih}(1) &= \gamma_i(1) f(x_h^T(0)) = 0 = \gamma_j(1) f(x_h^T(0)) = B_{jh}(1) = s_{ij}B_{jh}(1). \\
\Delta w_{ih}(1) &= \frac{\eta}{\Delta t}\sum_{l=1}^{N} B_{il}(1)\left[C_1^{-1}\right]_{lh} = 0 = \frac{\eta}{\Delta t}\sum_{l=1}^{N} B_{jl}(1)\left[C_1^{-1}\right]_{lh} = \Delta w_{jh}(1) \\
&= s_{ij}\Delta w_{jh}(1);
\end{aligned}$$

in the first equality we have used the expression of γ in terms of its components, that is,

$$\gamma_i(k) = -e_i(k) + (1 - \Delta t)e_i(k-1) + \Delta t \sum_{l=1}^{N} w_{il} f'(x_l(k)) e_l(k-1). \tag{3.1.4}$$

Now, assuming the equalities to hold for k, they will be demonstrated for $k+1$. To do so, we will use the induction hypothesis, the fact that $e_i(k) = 0$ for $i > 1$, the expression (3.1.4) for computing γ, and formula (3.1.2) for the update with matrix notation. We have:

$$\begin{aligned}
\frac{\gamma_i(k+1)}{\gamma_j(k+1)} &= \frac{-e_i(k+1)+(1-\Delta t)e_i(k)+\Delta t\sum_{l=1}^N w_{il}f'(x_l(k+1))e_l(k)}{-e_j(k+1)+(1-\Delta t)e_j(k)+\Delta t\sum_{l=1}^N w_{jl}f'(x_l(k+1))e_l(k)} \\
&= \frac{\Delta t w_{i1}f'(x_1(k+1))e_1(k)}{\Delta t w_{j1}f'(x_1(k+1))e_1(k)} = \frac{w_{i1}}{w_{j1}} = s_{ij}. \\
\frac{B_{ih}(k+1)}{B_{jh}(k+1)} &= \frac{B_{ih}(k)+\gamma_i(k+1)f(x_h^T(k))}{B_{jh}(k)+\gamma_j(k+1)f(x_h^T(k))} \\
&= \frac{s_{ij}\left(B_{jh}(k)+\gamma_j(k+1)f(x_h^T(k))\right)}{B_{jh}(k)+\gamma_j(k+1)f(x_h^T(k))} = s_{ij}. \\
\frac{\Delta w_{ih}(k+1)}{\Delta w_{jh}(k+1)} &= \frac{\left(\gamma_i(k+1)-\sum_{l=1}^N B_{il}(k)\left[\hat{C}_k^{-1}f_k\right]_l\right)\left[\hat{C}_k^{-1}f_k\right]_h^T}{\left(\gamma_j(k+1)-\sum_{l=1}^N B_{jl}(k)\left[\hat{C}_k^{-1}f_k\right]_l\right)\left[\hat{C}_k^{-1}f_k\right]_h^T} \\
&= \frac{s_{ij}\left(\left(\gamma_j(k+1)-\sum_{l=1}^N B_{jl}(k)\left[\hat{C}_k^{-1}f_k\right]_l\right)\right)}{\left(\gamma_j(k+1)-\sum_{l=1}^N B_{jl}(k)\left[\hat{C}_k^{-1}f_k\right]_l\right)} = s_{ij}. \\
\frac{w_{i1}(k+1)}{w_{j1}(k+1)} &= \frac{w_{i1}(k)+\Delta w_{i1}(k+1)}{w_{j1}(k)+\Delta w_{j1}(k+1)} \\
&= \frac{s_{ij}\left(w_{j1}(k)+\Delta w_{j1}(k+1)\right)}{w_{j1}(k)+\Delta w_{j1}(k+1)} = s_{ij}.
\end{aligned}$$

□

According to lemma 1, in each update of the APRL algorithm, the weight update matrix can be expressed as

$$\Delta W(k) = \begin{pmatrix} \Delta w_{11}(k) & \Delta w_{12}(k) & \cdots & \Delta w_{1N}(k) \\ \Delta w_{21}(k) & \Delta w_{22}(k) & \cdots & \Delta w_{2N}(k) \\ s_{32}\Delta w_{21}(k) & s_{32}\Delta w_{22}(k) & \cdots & s_{32}\Delta w_{2N}(k) \\ \vdots & \vdots & \ddots & \vdots \\ s_{N2}\Delta w_{21}(k) & s_{N2}\Delta w_{22}(k) & \cdots & s_{N2}\Delta w_{2N}(k) \end{pmatrix},$$

and the weight at time K is given by

$$w_{ij}(K) = w_{ij}(0) + s_{i2}\sum_{k=1}^{K}\Delta w_{2j}(k), \quad j > 2.$$

So, in the single output case, between the N^2 weights, which are the free parameters that define the model, only $2N$ are really independent. On one side, the N weights that connect the reservoir to the single output. On the other side, all the $N(N-1)$ reservoir weights are determined by the value of the N weights from the whole network to one of the reservoir units (for example, in the equation above, they are determined by the weights w_{2i} from the network to unit 2), and the initial weight matrix. This result suggests a functional separation between the output weights and the reservoir weights of a RNN.

3.2 BPDC

The BPDC algorithm will be given here, starting from the previous results for APRL. We shall also discuss its cost.

3.2.1 BPDC Algorithm

The consideration of formula (3.1.1) suggests to delete the momentum term and also not to try to accumulate the full matrix C_k, using instead only instantaneous information. Doing this in (3.1.1) and recalling the definition $\hat{C}(k) = \epsilon I + f_k f_k^T$, the update is then given by

$$\Delta w_{ij}(k+1) = \frac{\eta}{\Delta t}\left[\hat{C}(k)^{-1} f_k\right]_j \gamma_i(k+1).$$

For the computation of $\hat{C}(k)^{-1}$, the small rank adjustment matrix inversion lemma is used (see appendix A.2.3), so

$$\begin{aligned}
\hat{C}(k)^{-1} f_k &= \left[\frac{1}{\epsilon} I - \frac{[\frac{1}{\epsilon} I f_k][\frac{1}{\epsilon} I f_k]^T}{1 + f_k^T \frac{1}{\epsilon} I f_k}\right] f_k \\
&= \left[\frac{1}{\epsilon} I - \frac{1}{\epsilon^2}\frac{f_k f_k^T}{1 + \frac{1}{\epsilon} f_k^T f_k}\right] f_k \\
&= \frac{1}{\epsilon} f_k - \frac{1}{\epsilon^2}\frac{f_k f_k^T f_k}{1 + \frac{1}{\epsilon} f_k^T f_k} \\
&= \frac{1}{\epsilon} f_k - \frac{1}{\epsilon^2}\frac{f_k \|f_k\|^2}{1 + \frac{1}{\epsilon}\|f_k\|^2} \\
&= \left[\frac{1}{\epsilon} - \frac{1}{\epsilon}\frac{\|f_k\|^2}{\epsilon + \|f_k\|^2}\right] f_k \\
&= \frac{1}{\epsilon}\left[\frac{\epsilon + \|f_k\|^2 - \|f_k\|^2}{\epsilon + \|f_k\|^2}\right] f_k \\
&= \left[\frac{1}{\epsilon}\frac{\epsilon}{\epsilon + \|f_k\|^2}\right] f_k \\
&= \frac{1}{\epsilon + \|f_k\|^2} f_k.
\end{aligned}$$

Thus the online update of the new algorithm will be expressed as

$$\Delta w_{ij}(k+1) = \frac{\eta}{\Delta t}\frac{f(x_j(k))}{\sum_{s=1}^N f(x_s(k))^2 + \epsilon}\gamma_i(k+1), \qquad (3.2.1)$$

where we recall that

$$\gamma_i(k+1) = \sum_{s \in O}\left((1-\Delta t)\delta_{is} + \Delta t w_{is} f'(x_s(k))\right) e_s(k) - e_i(k+1).$$

3.2.2 Cost of the Algorithm

The cost of the algorithm will be calculated in the most common case of a problem with only one output. For notation, it will be considered that $S = \{1\}$, that is, the first unit is the output of the model. Applying equation (3.2.1) to this particular problem, the update becomes

$$\begin{aligned}\Delta w_{ij}(k+1) &= \frac{\eta}{\Delta t}\frac{f(x_j(k))}{\|f(x(k))\|^2+\epsilon} \\ &\quad\times\begin{cases}((1-\Delta t)+\Delta t w_{11} f'(x_1(k)))\, e_1(k) - e_1(k+1), & i=1, \\ \Delta t w_{i1} f'(x_1(k)) e_1(k), & i>1.\end{cases}\end{aligned} \tag{3.2.2}$$

The complexity will be measured in terms of the number of multiplications to obtain the update. We discuss next the cost of the full algorithm (with update of all the weights) and the corresponding cost of the simplified version (updating only the outputs weights, as it will be explained in section 3.3).

Complete Update

For the whole algorithm the operations required are:

- The denominator of the equation (3.2.2) can be computed with N multiplications (to estimate the norm of the vector of activations).
- The product of the denominator by the numerator is one operation for each j, with a total of N.
- The upper term of the right part (when $i = 1$) will be calculated only once, so it is not considered.
- In the lower term, the dependence on i is given only by the term w_{i1}, which means that, precalculating $\Delta t f'(x_1(k))e_1(k)$, just one more multiplication is needed for each i, that is, a total of N operations.
- The remaining calculation is the multiplication of the left part by the right one, for each i and j, with a cost of N^2.

Putting this results together, the total cost of the online algorithm of BPDC is $N^2 + 3N$ for each data point k.

Update of the Output Weights

Changing only the output weights implies a big simplification. This is obtained fixing $i = 1$ in the update equation.

- The denominator, again, requires N multiplications.

- The product of the denominator by the right part will be calculated once, so it does not change the cost.
- The numerator will be multiplied by the previous quantity for each j with a cost N.

The complexity of the simplified BPDC is $2N$ for each data point.

3.3 Back to ESN

The relationship within the reservoir weights stated by lemma 1 represents a clear division between those that connect with the output, and those between the units of the reservoir.

On the other side, in formula (3.2.2) it is clear that the instantaneous error $e_i(k+1)$ only is propagated to the output weights, as stated in [11]. Moreover in [20] an empirical comparative of the magnitude of the update for APRL was presented. In this work, it was shown that the output weights change much faster than the other ones.

These ideas motivate the separation of the model training into two parts: the construction of a reservoir that stores as much temporal information as possible (with respect to the problem need) and the creation of a readout function to transform all this information into the desired target. This is exactly the paradigm known as Reservoir Computing that emerged both as the ESN and the LSM approaches (as mentioned in section 1.5).

In the case of BPDC algorithm, this justifies the update of only the output weights. With a slightly simplification on equation (3.2.2) we have the LMS [21] filter, that it comes to be an online version of a simple linear regression. Using the classical batch linear regression (with the pseudoinverse of the internal states matrix) we recover again the ESN approach.

So with this last simplification, mainly the update only of a part of the weights, the theoretical path from the classical approaches to the RC paradigm is complete.

3.4 Reservoir Architecture Definition

Although this part of the RC paradigm, the training/construction of good reservoirs, is a very big research field, here only some notions about the main ideas will be given.

It is important to notice that the random initialization of the reservoir matrix of a RNN is almost an antonym of optimization. In fact, even the simplest selection of reservoirs, as could be the generation of N random ones, and the selection of the best (provided that an evaluation method is given) will produce an improvement, with respect to the case of generating only one matrix, with probability $\frac{N-1}{N}$ (that is, the proportion of the N matrices that are not optimal).

Moreover, a big variability in the results has been observed, as stated in [22], so the importance of reservoir definition should not be obviated.

In the present section, some of the existing algorithms for defining a reservoir will be briefly explained.

The simplest approach to the reservoir definition is to try to build good "general" models. This means that the reservoirs will be independent of the concrete problem to be solved. So no data will be used, neither the inputs nor the desired outputs, to optimize the reservoir, and only some general receipts which are expected to produce models with a richness enough for most of the problems will be used.

Some examples of this approach are:

- In the ESN paradigm, the reservoir has to provide [23]:
  - A rich set of dynamics, so it has to be big enough.
  - Independent dynamics, so the reservoir matrix should be sparse.
  - Random dynamics. This permits the reservoir to have different dynamics and so to store more information. The matrix will be therefore randomly initialized.
  - Independence of the initial state. As it was described in subsection 1.5.3, this is essentially the ESP.
- As it was explained in the section 1.5, the scale of the weights can be used to determine the network speed and its nonlinearity, in function of the behaviour required.
- Another approaches try to specify special topologies, such as bioinspired ones, small-world, expansions of good small networks to bigger ones... Or even simplified versions, such as FFNN with delays, diagonal weight matrices...

Other more complex approaches use information that depends on the problem. These techniques can be classified into two big groups: the unsupervised and the supervised training methods.

The first ones do not use any information about the desired target of the problem. An example of these approaches, the Intrinsic Plasticity adaptation, will be described in detail in subsequent sections.

The second approaches, the supervised one, use the desired output for improving the reservoir performance. As an example, the evolutionary methods developed in [24] can be cited. In this work, the main parameters of a RNN are estimated with an evolutionary strategy (ES). This parameters are the size of the RNN, the spectral radius of the reservoir weight matrix and the connectivity (the proportion of links over the total N^2 possibilities). Moreover, evolutionary algorithms (EA) were also used to find all the weights of the RNN for small networks. A simplification of this method is to fix the topology (i.e., set first which weights will be equal to zero) and to apply the EA only to the non zero ones. Another method of this type used in [25] is to search in the space of the connection topology, i.e., the evolutionary algorithm determines which weights will be zero. That is, a mask matrix will be applied to a fixed weight matrix, and the optimization will be done over the space of this mask matrices.

In this work, the supervised reservoir training methods will not be further considered, and we will focus on the plasticity methods.

3.4.1 Unsupervised Reservoir Training: Intrinsic Plasticity

Intrinsic plasticity (IP) is the capability of certain neurons to change their excitability in function of the stimulus that are receiving [26]. This phenomenon is determined only by the presynaptic activity, so the unit presents an adaptation based on local information.

This property has been simulated in several models of artificial neurons. In particular, two of them will be described next: Exponential Intrinsic Plasticity (EIP) and Gaussian Intrinsic Plasticity (GIP).

These two techniques are considered unsupervised because they only use the input of the unit to estimate the adaptation of the network, so it is not necessary to have any knowledge about the desired final output of the model. In fact, they only adapt the reservoir so every unit transmits as much information as it can, with independence of the task that the network will have to solve.

Exponential Intrinsic Plasticity

Triesch [27] introduced an algorithm to simulate this IP in a neuron model with a continuous activation function. This method tries to transform the distribution of the output of a neuron into an exponential distribution. This is because the exponential distribution is the one that maximizes the entropy (and so the information that a neuron can transmit) among those with a given mean. This can be viewed, in a biological sense, as the distribution that maximizes the information transmitted with a given metabolic cost.

The model that will be used in this case is a single unit with input synaptic current x, with density $p(x)$. This input is transformed with a nonlinearity g to get $y = g(x)$. In an analog model, this will be directly the output of the unit. If a firing mechanism is used, y will represent the firing rate of the neuron. This is why g will be considered non-negative. Assuming that is also monotonically increasing (as the usual Fermi activation) the density of y will be (using a simple change of variable):

$$q(y) = \frac{p(x)}{\frac{\partial y}{\partial x}}. \tag{3.4.1}$$

The objective now is to make $q(y)$ as close as possible to $f_{\text{exp}}(y) = \frac{1}{\mu} \exp \frac{-y}{\mu}$, with μ fixed. As the measure of the distance between the two distributions the Kullback Leibler Divergence (KL-Divergence) is used. So the goal is to minimize the expression

$$\begin{aligned}
D \equiv d_{\mathrm{KL}}(q \| f_{\exp}) &= \int q(y) \log \left(\frac{q(y)}{\frac{1}{\mu} \exp \frac{-y}{\mu}} \right) dy \\
&= \int q(y) \log q(y) dy - \int q(y) \left(-\frac{y}{\mu} - \log \frac{1}{\mu} \right) dy \\
&= \int q(y) \log q(y) dy + \frac{1}{\mu} \int q(y) y dy + \log \mu \int q(y) dy \\
&= -H(y) + \frac{1}{\mu} E[y] + \log \mu.
\end{aligned}$$

From this expression it can be deduced that the exponential distribution is the one which maximizes the entropy for a fixed mean. Consider $q(y)$ an arbitrary distribution, with a given mean $\mu = E[y]$. The KL-Divergence with respect to the exponential distribution is always positive. In fact, the KL-Divergence is a distance, so it is zero if and only if $q(y)$ is precisely $f_{\exp}(y)$. But this divergence is minimized if and only if $H(y)$ is maximized (because $\mu = E[y]$ and $\log \mu = logE[y]$ are fixed), so $q(y)$ has maximum entropy if and only if the divergence is 0, and the divergence is zero if and only if $q(y) = f_{\exp}(y)$. In other words, if the distribution $q(y)$ has maximum entropy, then it has to be precisely the exponential distribution for a mean μ. If not, then the KL-Divergence to the exponential distribution will be positive, and a distribution of an exponential variable z will have a KL-Divergence of 0 and so a larger entropy:

$$\begin{aligned}
d_{\mathrm{KL}}(q \| f_{\exp}) = -H(y) + \frac{1}{\mu}\mu + \log \mu &> 0 = -H(z) + \frac{1}{\mu}\mu + \log \mu = d_{\mathrm{KL}}(f_{\exp} \| f_{\exp}) \\
\Longrightarrow H(z) &> H(y).
\end{aligned}$$

In this case, we shall use as the activation function

$$y = g(x) = \frac{1}{1 + \exp(-(ax + b))}.$$

The adaptation of the unit will be given by the modification of the activation function parameters a and b for which a stochastic descent gradient rule will be derived to minimize the expression given for D with respect to them.

Now the derivatives of D with respect to the parameters a and b will be computed. They are the basis of the plasticity rule.

$$\begin{aligned}
\frac{\partial D}{\partial a} &= -\frac{\partial H(y)}{\partial a} + \frac{1}{\mu}\frac{\partial E\left[y\right]}{\partial a} \\
&= \frac{\partial}{\partial a} E\left[\log q(y) + \frac{1}{\mu} y\right] \\
&= \frac{\partial}{\partial a} E\left[\log \frac{p(x)}{\frac{\partial y}{\partial x}} + \frac{1}{\mu} y\right] \\
&= E\left[\frac{\partial}{\partial a}\log p(x) - \frac{\partial}{\partial a}\log\frac{\partial y}{\partial x} + \frac{1}{\mu}\frac{\partial y}{\partial a}\right] \\
&= E\left[-\frac{\partial}{\partial a}\log\frac{\partial y}{\partial x} + \frac{1}{\mu}\frac{\partial y}{\partial a}\right] \\
&= E\left[-\frac{\partial}{\partial a}\log a - \frac{\partial}{\partial a}\log y - \frac{\partial}{\partial a}\log(1-y) + \frac{1}{\mu}\frac{\partial y}{\partial a}\right] \\
&= E\left[-\frac{1}{a} - \frac{1}{y}xy(1-y) + \frac{1}{1-y}xy(1-y) + \frac{1}{\mu}xy(1-y)\right] \\
&= E\left[-\frac{1}{a} - x + xy + xy + \frac{1}{\mu}xy - \frac{1}{\mu}xy^2\right] \\
&= -\frac{1}{a} + E\left[-x + \left(2 + \frac{1}{\mu}\right)xy - \frac{1}{\mu}xy^2\right].
\end{aligned}$$

It has been used that $\frac{\partial y}{\partial x} = ay(1-y)$, and $\frac{\partial y}{\partial a} = xy(1-y)$. The corresponding derivative for the parameter b is the expression:

$$\begin{aligned}
\frac{\partial D}{\partial b} &= -\frac{\partial H(y)}{\partial b} + \frac{1}{\mu}\frac{\partial E\left[y\right]}{\partial b} \\
&= \frac{\partial}{\partial b} E\left[\log \frac{p(x)}{\frac{\partial y}{\partial x}} + \frac{1}{\mu} y\right] \\
&= E\left[-\frac{\partial}{\partial b}\log\frac{\partial y}{\partial x} + \frac{1}{\mu}\frac{\partial y}{\partial b}\right] \\
&= E\left[-\frac{\partial}{\partial b}\log a - \frac{\partial}{\partial b}\log y - \frac{\partial}{\partial b}\log(1-y) + \frac{1}{\mu}\frac{\partial y}{\partial b}\right] \\
&= E\left[-\frac{1}{y}y(1-y) + \frac{1}{1-y}y(1-y) + \frac{1}{\mu}y(1-y)\right] \\
&= E\left[-(1-y) + y + \frac{1}{\mu}y - \frac{1}{\mu}y^2\right] \\
&= E\left[-1 + 2y + \frac{1}{\mu}y - \frac{1}{\mu}y^2\right] \\
&= E\left[-1 + \left(2 + \frac{1}{\mu}\right) - \frac{1}{\mu}y^2\right],
\end{aligned}$$

where the equality $\frac{\partial y}{\partial b} = y(1-y)$ has been used.

In the online stochastic gradient descent we replace the averages above by their sample values and the learning rule (with learning rate η) updates the a and b parameters in the opposite direction of the gradient, that is:

$$
\begin{aligned}
\Delta a &= \eta \left(\frac{1}{a} + x - \left(2 + \frac{1}{\mu} \right) xy + \frac{1}{\mu} xy^2 \right), \\
\Delta b &= \eta \left(1 - \left(2 + \frac{1}{\mu} \right) y + \frac{1}{\mu} y^2 \right).
\end{aligned}
$$

In figure 3.4.1 an example of the transformation of the output distribution through this algorithm is showed.

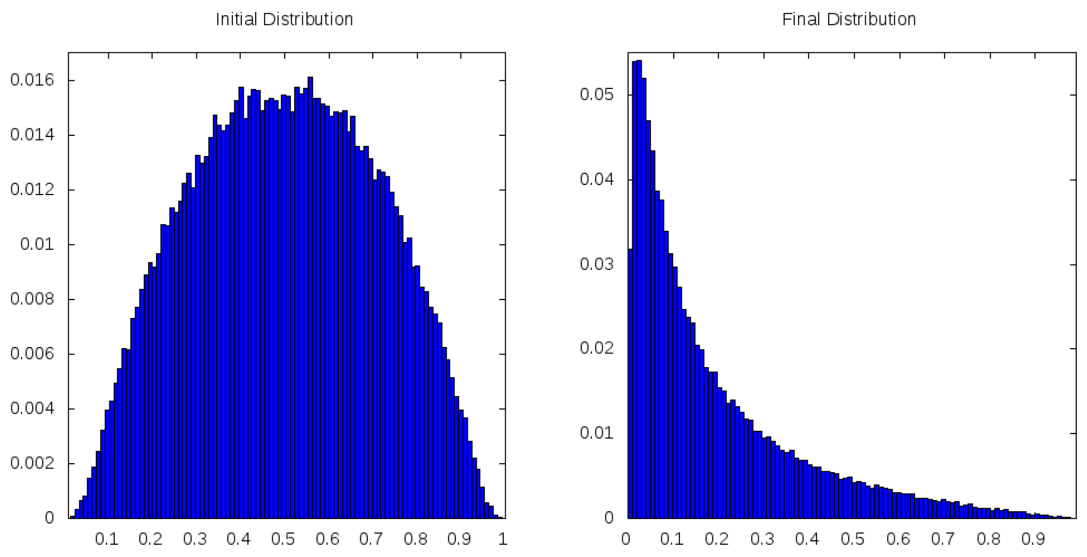


Figure 3.4.1: Transformation of the output distribution in a single Fermi unit with Gaussian noise as input. Exponential IP is applied during 1000 steps with $\eta = 0.01$ and $\mu = 0.2$.

Gaussian Intrinsic Plasticity

Schrawen and Steil [28] followed the same philosophy as Triesch (i.e. to adapt the distribution of the output of the units in order to maximize the entropy). But this time the target distribution is not the exponential one (that maximizes, for a fixed mean, and in a non-negative domain, the entropy) but the Gaussian distribution. This distribution is defined over all the real axis, and transmits the maximum information for a fixed mean and a given deviation. This new algorithm can be applied, therefore, with activation functions such as the hyperbolic tangent, with both negative and positive values.

The model is similar as the one for exponential IP, but this time the nonlinearity g will be defined over all $\mathbb{R}$. Again, the equality (3.4.1) will be used.

The KL-Divergence will measure the distance between the two distribution. Now, the goal is to approximate $q(y)$ to $f_{\text{gaus}}(y) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right)$, for μ and σ given, so the expression to minimize is

$$\begin{aligned}
D \equiv d_{\mathrm{KL}}(q \| f_{\mathrm{gaus}}) &= \int q(y) \log \left(\frac{q(y)}{\frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-\mu)^2}{2\sigma^2}\right)} \right) dy \\
&= \int q(y) \log q(y) dy - \int q(y) \left(\log \frac{1}{\sigma\sqrt{2\pi}} - \frac{y^2}{2\sigma^2} - \frac{\mu^2}{2\sigma^2} + \frac{\mu y}{\sigma^2} \right) dy \\
&= -H(Y) + E\left[\frac{y^2}{2\sigma^2} - \frac{\mu y}{\sigma^2} \right] + K,
\end{aligned}$$

where K is an expression that does not depend on y (thus it will not contribute to the optimization of the parameters).

In this case, the stochastic descent gradient rule will be obtained for the activation function g, defined as

$$y = g(x) = \tanh(ax+b) = \frac{\exp(ax+b) - \exp(-(ax+b))}{\exp(ax+b) + \exp(-(ax+b))},$$

$$\begin{aligned}
\frac{\partial D}{\partial a} &= -\frac{\partial H(y)}{\partial a} + \frac{\partial}{\partial a} E\left[K + \frac{y^2}{2\sigma^2} - \frac{\mu y}{\sigma^2} \right] \\
&= \frac{\partial}{\partial a} E\left[\log q(y) + K + \frac{y^2}{2\sigma^2} - \frac{\mu y}{\sigma^2} \right] \\
&= \frac{\partial}{\partial a} E\left[\log \frac{p(x)}{\frac{\partial y}{\partial x}} + K + \frac{y^2}{2\sigma^2} - \frac{\mu y}{\sigma^2} \right] \\
&= E\left[\frac{\partial}{\partial a} \log p(x) - \frac{\partial}{\partial a} \log \frac{\partial y}{\partial x} + \frac{\partial}{\partial a} \left(K + \frac{y^2}{2\sigma^2} - \frac{\mu y}{\sigma^2} \right) \right] \\
&= E\left[-\frac{\partial}{\partial a} \log \frac{\partial y}{\partial x} + \frac{\partial}{\partial a} \frac{y^2}{2\sigma^2} - \frac{\partial}{\partial a} \frac{\mu y}{\sigma^2} \right] \\
&= E\left[-\frac{\partial}{\partial a} \log a - \frac{\partial}{\partial a} \log(1-y^2) + \frac{\partial}{\partial a} \frac{y^2}{2\sigma^2} - \frac{\partial}{\partial a} \frac{\mu y}{\sigma^2} \right] \\
&= E\left[-\frac{1}{a} + \frac{1}{1-y^2} 2yx(1-y^2) + \frac{1}{2\sigma^2} 2yx(1-y^2) - \frac{\mu}{\sigma^2} x(1-y^2) \right] \\
&= -\frac{1}{a} + E\left[x \left(-\frac{\mu}{\sigma^2} + \frac{y}{\sigma^2} (2\sigma^2 + 1 - y^2 + \mu y) \right) \right],
\end{aligned}$$

where the equalities $\frac{\partial y}{\partial x} = a(1-y^2)$, and $\frac{\partial y}{\partial a} = x(1-y^2)$ have been used. The derivative with respect to the parameter b is:

$$
\begin{aligned}
\frac{\partial D}{\partial b} &= -\frac{\partial H(y)}{\partial b} + \frac{\partial}{\partial b} E\left[K + \frac{y^2}{2\sigma^2} - \frac{\mu y}{\sigma^2}\right] \\
&= \frac{\partial}{\partial b} E\left[\log \frac{p(x)}{\frac{\partial y}{\partial x}} + K + \frac{y^2}{2\sigma^2} - \frac{\mu y}{\sigma^2}\right] \\
&= E\left[-\frac{\partial}{\partial b} \log \frac{\partial y}{\partial x} + \frac{\partial}{\partial b}\frac{y^2}{2\sigma^2} - \frac{\partial}{\partial b}\frac{\mu y}{\sigma^2}\right] \\
&= E\left[-\frac{\partial}{\partial b} \log a - \frac{\partial}{\partial b}\log(1-y^2) + \frac{\partial}{\partial b}\frac{y^2}{2\sigma^2} - \frac{\partial}{\partial b}\frac{\mu y}{\sigma^2}\right] \\
&= E\left[\frac{1}{1-y^2} 2y(1-y^2) + \frac{1}{2\sigma^2} 2y(1-y^2) - \frac{\mu}{\sigma^2}(1-y^2)\right] \\
&= E\left[-\frac{\mu}{\sigma^2} + \frac{y}{\sigma^2}(2\sigma^2 + 1 - y^2 + \mu y)\right],
\end{aligned}
$$

where it has been used that $\frac{\partial y}{\partial b} = (1 - y^2)$.

So the online learning rule (with rate η) is:

$$
\begin{aligned}
\Delta a &= -\eta\left(-\frac{1}{a} + x\left(-\frac{\mu}{\sigma^2} + \frac{y}{\sigma^2}(2\sigma^2 + 1 - y^2 + \mu y)\right)\right), \\
\Delta b &= -\eta\left(-\frac{\mu}{\sigma^2} + \frac{y}{\sigma^2}(2\sigma^2 + 1 - y^2 + \mu y)\right).
\end{aligned}
$$

An example of the output distribution transformation can be viewed in figure 3.4.2 .

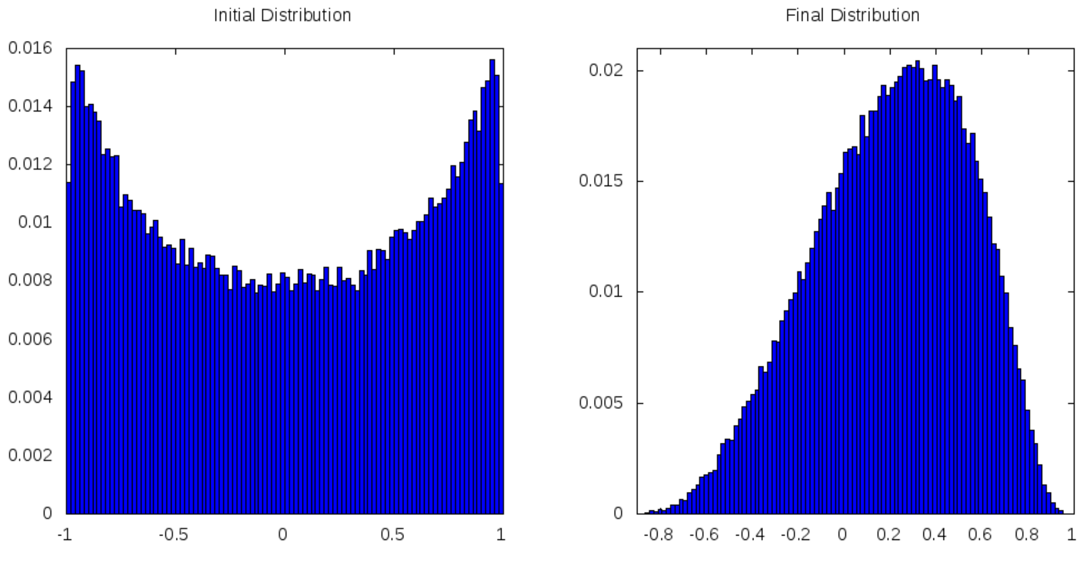


Figure 3.4.2: Transformation of the output distribution in a single $\tanh$ unit with gaussian noise as input. Gaussian IP is applied during 1000 steps with $\eta = 0.01$, $\mu = 0.2$ and $\sigma = 0.4$.

Chapter 4

Simulations

In this chapter, a set of experiments will be described. First, some simulations will help to understand how the ESN (and the RC approach in general) works. Then, some state of the art problems will be tackled to show the effectiveness of ESN. Finally, the application to a real world problem, the prediction of wind power, will be included.

4.1 Simulations Setup

A general description of the simulation environment will be given next.

4.1.1 Model

In this section, the RNN used during the simulations will be described.

- The time constant will be equal to 1, i.e., $\Delta t = 1$, so each update is given by equation (1.3.2), $x(k+1) = Wf(x(k))$. This implies that the units themselves have no memory.
- The activation function used is the $\tanh$ function, with parameters a and b, so

  $$f(x; a, b) = \frac{\exp(ax+b) - \exp(-(ax+b))}{\exp(ax+b) + \exp(-(ax+b))}.$$

  The parameters a and b are included to generalize the model, but in this simulations no IP adaptation is used, so they will be fixed, $a \equiv 1$ and $b \equiv 0$.
- In all the simulations below the RNN will have only one output.
- At the beginning of the simulations, a synthetic input with constant value to 1 will be generated. This input will act as the bias of the model.
- In the case of the ESN model, the weights will not be updated, and only a linear regression will take place at the end of the algorithm. In the BPDC model, the weights will be updated at each timestep with equation (3.2.2):

$$\begin{aligned}\Delta w_{ij}(k+1) &= \frac{\eta}{\Delta t}\frac{f(x_j(k))}{\|f(x(k))\|^2+\epsilon} \\ &\times\begin{cases}((1-\Delta t)+\Delta t w_{11}f'(x_1(k)))\,e_1(k)-e_1(k+1), & i=1,\\ \Delta t w_{i1}f'(x_1(k))e_1(k), & i>1,\end{cases}\end{aligned}$$

where $\epsilon = 0.02$ for all the experiments.

- In both BPDC and ESN, during the iteration of the model over the training test, gaussian noise can be injected to the states. This is done easily. In the case of ESN, once the matrix with the outputs of the units O has been computed, some gaussian noise (with a deviation σ that depends on each experiment, but generally it will be of $\sigma = 0.00001$) will be added. Then, the perturbed matrix will be used to compute the output weights. For BPDC, the gaussian noise is added at each step, before updating the weights of the model.
- Another auxiliary model, a MLP, will be used for obtaining a basis error. It will be applied only to those problem will delay windows, because the MLP itself can not retain any temporal information, so its application to purely generative (without inputs) tasks makes no sense.

4.1.2 Input/Output

Although each experiment will be explained in its corresponding section, a general idea of the input and output of the system will be given here.

- As it has been described, the algorithm will use a training sequence composed of the input/output pairs evolution through the time, i.e.

$$((u(1), y(1)), \ldots, (u(K), y(K))).$$

- Once the model is trained, testing is done by iterating the dynamic of the RNN (i.e. updating its state with equation (1.3.2), $x(k+1) = Wf(x(k))$, using if needed the inputs of the test set) starting at the point in which the training ended. So the testing is viewed as a continuation of the training sequence, but without using the desired outputs or updating the weights (for BPDC model).
- In those experiments where some previous outputs will be used as inputs (using a temporal window), during the test the RNN is run in a generative way. This means that, instead of using the true outputs as inputs, the outputs produced by the RNN in previous timesteps are inserted again as inputs, so no exterior information is put into the model.

4.1.3 Training and Testing Procedures

In this work, two main training algorithms will be used. The first is the ESN approach (described in section 1.5). This is one of the basics algorithms of the RC paradigm, and it is the end in the evolution path of RNN algorithms that we have considered. It is also very illustrative of how the RC paradigm works, and it presents surprisingly good results. Basically, the method consists on:

- Compute internal states using *teacher forcing*, iterating the RNN dynamic for each timestep. So for each step of the training sequence the state of the RNN is computed with $x(k+1) = Wf(x(k))$, forcing the state of the output unit to the desired value at each timestep, $x_1(k) = y_1(k)$.
- The internal states are collected into a matrix X, so each row in X contains the state of the RNN at one timestep.
- Clean up a percentage $p\%$ of the the first data values (this is the washout, it is done because of the dependency of this data on the initial state). More precisely, this means dropping the first $p\%$ rows of the matrix X. In our case, this will be replaced for doing an extra epoch at the beginning.
- Compute the output weights through a simple linear regression, using the pseudoinverse of the matrix $O = f(X)$ and the desired output vector Y.

The second algorithm is BPDC (see section 3.2), where we follow three different variations, although the training procedure will be the same for all of them. In the first one, every weight will be updated. We will refer to this variation as BPDC-C (complete BPDC). In the second approach, only the output weights will be changed (this is the simplified BPDC, BPDC-S). The third variation is a simplification of the second one. The learning rule is modified so the LMS filter rule (see [21]) could be applied, we will call this combination of LMS and BPDC as BPDC-L. Summarizing, training is done as follows:

- For each timestep.
  - Iterate the network dynamic with expression $x(k+1) = Wf(x(k))$, using the desired output of the RNN.
  - Update the weights according to the corresponding BPDC rule if the washout phase has been completed, either all the weights, or only the output ones.

With respect to the washout of the system, we have decided to do it using extra epochs over all the data (see section 4.5 for more details).

In both algorithms the testing phase is analogous. The trained RNN is iterated using either inputs of the test patterns or, if a generative mode is desired, feeding back the outputs into the inputs at the next timestep:

- For each timestep.
  - Feed back the previous outputs as the next inputs if a generative model is sought.
  - Iterate the network dynamic.

4.1.4 Error Measurement

To quantify the error produced by a model, the normalized square error will be used. This error is defined as:

$$\text{NSE} = \sum_{k=1}^{K} \sum_{s \in O} \frac{[x_s(k) - y_s(k)]^2}{\sigma^2}.$$

In our case only one output will be considered, so the NSE can be reduced to

$$\text{NSE} = \sum_{k=1}^{K} \frac{[x_1(k) - y_1(k)]^2}{\sigma^2},$$

where K is the length of the test sequence and σ is the deviation of the teacher signal $y_1(k)$ over the timesteps $k = 1, \ldots, K$. Notice that this error is normalized, so a trivial predictor that uses the mean of the target as the predicted values will have an error $\text{NSE}_{\text{TP}} = 1 = 100\%$.

The error measures included in this work are the mean over 20 repetitions of the simulations (to take account of the variability due to the random initialization). The standard deviation of the error is also given, to evaluate this variability.

4.1.5 List of Experiments

Table 4.1.1 summarizes the settings of each of the experiment included in this chapter.

Name	**Ref**	**Model**	**Parameters**
Pure Sinewave Generator	4.2.1	ESN/BPDC	$N = 20, ep = 1, \lambda = 0.8, \eta = 0.01$
Sinewave Generator with Delay Input	4.2.2	ESN/BPDC/MLP	$N = 20, ep = 1, \lambda = 0.8, \eta = 0.05$
Modulated Pure Sinewave Generator	4.2.3	ESN/BPDC	$N = 100, ep = 1, \lambda = 0.8, \eta = 0.01$
Mackey-Glass	4.3	ESN/BPDC/MLP	$N = 100, ep = 1, \lambda = 0.8, \eta = 0.01$
Pure Wind Power Forecasting	4.4	ESN/BPDC/MLP	$N = 30, ep = 1, \lambda = 0.8, \eta = 0.01$

Table 4.1.1: Settings of the simulations. N is the number of units (considering also the output one), ep is the number of epochs used to do the washout, λ is the spectral radius of W^{Res} and η is the learning rate for the BPDC algorithms.

4.2 Sinewave Generator

The first experiment consists simply on learning to generate a sinusoidal wave. This is quite an easy problem, although obtaining a stable autonomous generator is not trivial, because the RNN must have a dynamic strong enough to keep the desired trajectory without fading to zero, but if the network becomes too unstable, the system will diverge quickly. But it is illustrative of how a RNN works under the RC paradigm. This set of simulations will illustrate the capability of the ESN algorithm. The BPDC variations will also be included in order to contrast their behaviour. The parameters, results and observations will be described in the corresponding sections.

4.2.1 Pure Generator

In this experiment, the task is to transform the RNN into a sinewave generator. The desired output is given by

$$y(k) = \frac{1}{2} \sin\left(\frac{k}{4}\right).$$

The size of the train set is 300 (the washout will be done with a complete epoch), and 50 steps of test are done (this means approximately 12 cycles for training and 2 for testing). Nevertheless, the error in a much bigger test sequence will be shown, to see how the model degenerates with time. The total number of units is 20, and the spectral radius 0.80. To set the spectral radius, every connection between the units of the reservoir (in our case, every w_{ij} with $i > 1, j > 1$ and $i \leq N - M, j \leq N - M$) will be scaled dividing by the spectral radius of W^{Res}. Then, they will be multiplied by the desired value 0.80. As it will be further discussed in section 5.2, this spectral radius close to 1 increases the computational power of the RNN, and it permits an autonomous dynamic. If the spectral radius were bigger, then the trajectory could diverge. The learning rate for the online algorithms (the BPDC methods) is $\eta = 0.01$. This value has been chosen so that the network dynamic does not diverge in any simulation. As it will be explained below, bigger values of η can provide better results.

No inputs are used; conceptually: during training the desired output enters into the RNN through the feedback connections (the links from the output unit to the other units), thanks to *teacher forcing* (the output unit is supposed to produce exactly the desired output). So, after some timesteps the sinewave should be spread over the internal units, and the reconstruction of the output should be possible. This is illustrated in figure 4.2.1, where the internal states during training of the first 4 units in ESN are shown (unit 1 is not included because is the output unit).

The error obtained for this task, in the case of the ESN model, is very small, as can be observed in table 4.2.1. In figure 4.2.2 the last two cycles of the test of this model is represented. As it can be observed, the RNN conserves the desired trajectory for a very long time. In the case of the models derived of BPDC, the error is bigger. Observing the graphics (see for example figure 4.2.3, corresponding to the BPDC-S model), with the used learning rate the models are not able to capture the dynamic. If a bigger learning rate is used, the stability of the model depends on the random initialization, so in the set of 20 repetitions it is easy that one diverges completely, but in some simulations the model success to learn the desired trajectory (see figure 4.2.4, where the BPDC-L models very well the sinewave dynamic). These results will be discussed in section 4.5.

Cycles	Steps	ESN	ESN*	BPDC-L	BPDC-L*	BPDC-S	BPDC-S*	BPDC-C	BPDC-C*
4	100	$3.09e-02 \pm 3.06e-03$	$3.08e-02 \pm 3.38e-03$	$8.80e-01 \pm 2.37e-01$	$9.03e-01 \pm 1.75e-01$	$8.58e-01 \pm 8.89e-02$	$8.77e-01 \pm 3.84e-02$	$9.06e-01 \pm 6.88e-06$	$9.06e-01 \pm 3.60e-06$
8	200	$4.73e-02 \pm 5.35e-03$	$4.75e-02 \pm 6.98e-03$	$9.26e-01 \pm 1.62e-01$	$9.33e-01 \pm 9.27e-02$	$8.75e-01 \pm 6.13e-02$	$8.88e-01 \pm 1.93e-02$	$9.03e-01 \pm 3.29e-06$	$9.03e-01 \pm 1.71e-06$
12	300	$6.58e-02 \pm 7.99e-03$	$6.63e-02 \pm 1.09e-02$	$9.44e-01 \pm 1.10e-01$	$9.49e-01 \pm 6.41e-02$	$8.81e-01 \pm 4.68e-02$	$8.92e-01 \pm 1.29e-02$	$9.02e-01 \pm 2.09e-06$	$9.02e-01 \pm 1.08e-06$
16	400	$8.51e-02 \pm 1.07e-02$	$8.58e-02 \pm 1.48e-02$	$9.57e-01 \pm 6.75e-02$	$9.57e-01 \pm 5.53e-02$	$8.85e-01 \pm 3.76e-02$	$8.94e-01 \pm 9.71e-03$	$9.01e-01 \pm 1.49e-06$	$9.01e-01 \pm 7.69e-07$
20	500	$1.05e-01 \pm 1.34e-02$	$1.06e-01 \pm 1.86e-02$	$9.67e-01 \pm 5.93e-02$	$9.54e-01 \pm 6.47e-02$	$8.88e-01 \pm 3.10e-02$	$8.95e-01 \pm 7.77e-03$	$9.01e-01 \pm 1.19e-06$	$9.01e-01 \pm 6.15e-07$
24	600	$1.25e-01 \pm 1.61e-02$	$1.26e-01 \pm 2.24e-02$	$9.76e-01 \pm 6.36e-02$	$9.57e-01 \pm 6.83e-02$	$8.90e-01 \pm 2.62e-02$	$8.96e-01 \pm 6.51e-03$	$9.01e-01 \pm 8.95e-07$	$9.01e-01 \pm 4.55e-07$
28	700	$1.45e-01 \pm 1.88e-02$	$1.46e-01 \pm 2.62e-02$	$9.81e-01 \pm 6.66e-02$	$9.58e-01 \pm 6.28e-02$	$8.91e-01 \pm 2.24e-02$	$8.96e-01 \pm 5.58e-03$	$9.01e-01 \pm 7.67e-07$	$9.01e-01 \pm 3.90e-07$

Table 4.2.1: Evolution of the normalized square error for Sinewave with respect to the length of the test sequence. The models marked with * are perturbed with white noise. The mean error is presented with its deviation. Both terms are computed over 20 runs.

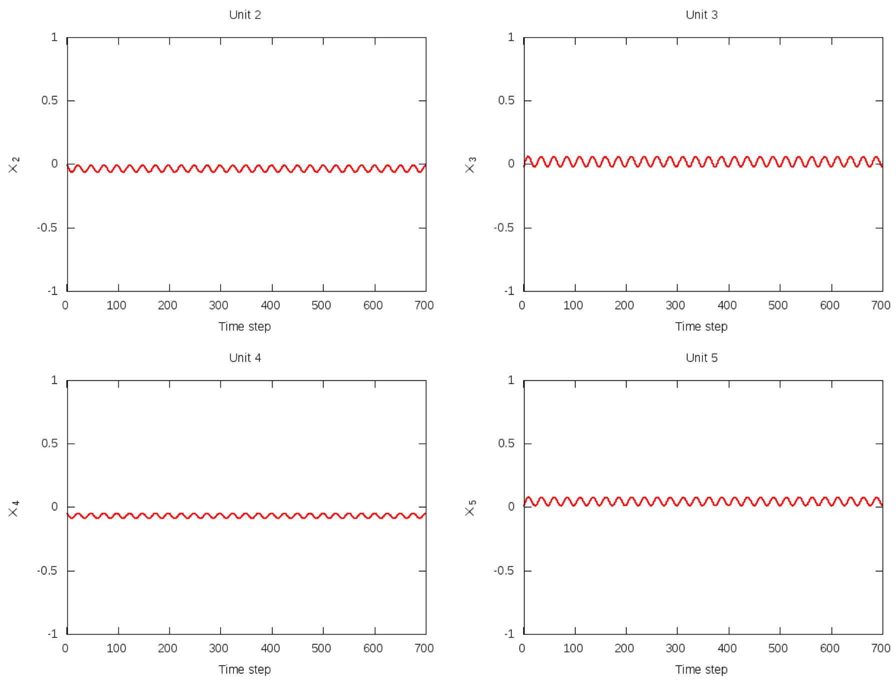


Figure 4.2.1: Activation of the first 4 internal units (units 1 to 5) during training, for the pure sinewave generator with ESN.

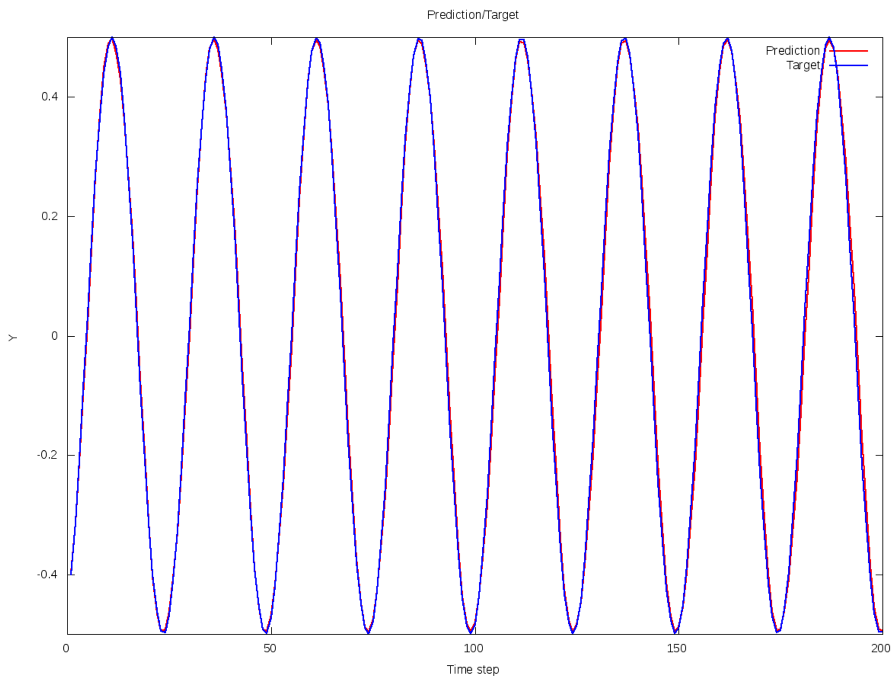


Figure 4.2.2: Prediction of the RNN versus the target (the sinewave) for the test of the pure sinewave generator with ESN. The model captures perfectly the required dynamic for the complete shown sequence.

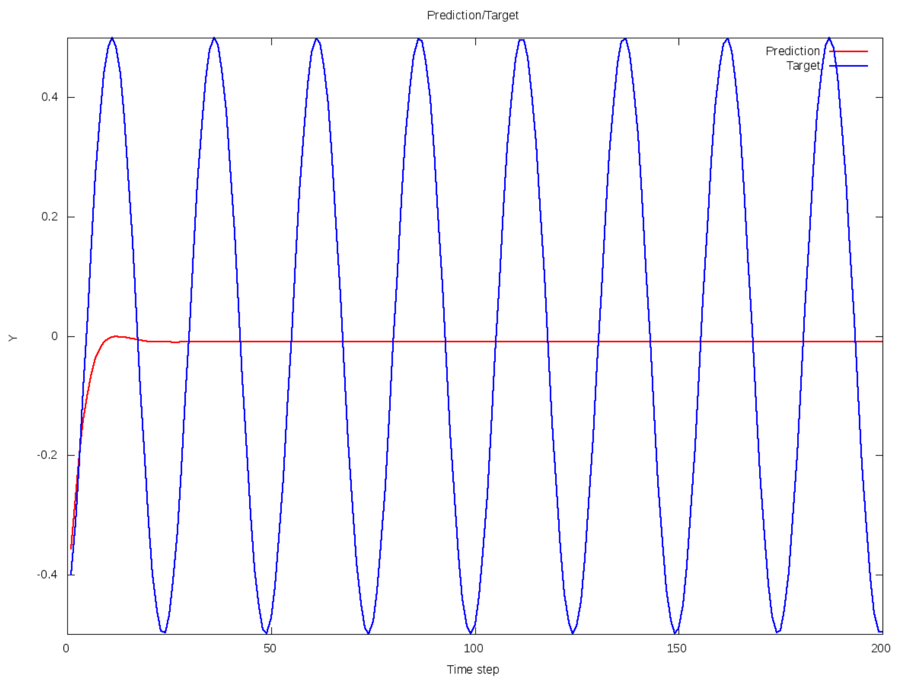


Figure 4.2.3: Prediction of the RNN versus the target (the sinewave) for the test of the pure sinewave generator with BPDC-S. The model fades too quickly.

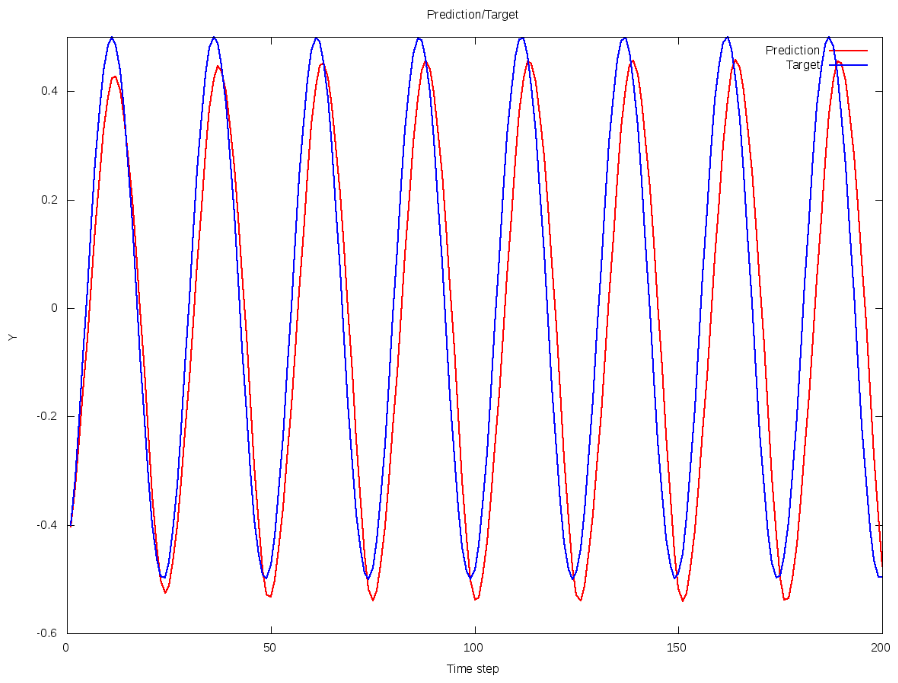


Figure 4.2.4: Prediction of the RNN versus the target (the sinewave) for the test of the pure sinewave generator with BPDC-L. The models captures the dynamics, but with more error than in the ESN example. A learning rate of $\eta = 0.3$ (30 times the one used for the mean error) has been used, which can derive in stability problems.

4.2.2 Generator with Delay Input

This experiment consists on the generation of a sinewave but using a delay windows with the last 5 steps as inputs. This is a little windows compared to the size of a sine cycle (25 steps), but it is enough to capture the tendency of the dynamical system, making much easier the task. So the input will be the delays of the last 5 outputs, and they will be used to predict the next step. For testing the model, it will be run in an autogenerative way, using its own predictions as the inputs of the next timesteps.

The configuration of the network is the same than in the previous experiment, a small RNN of only 20 units and a spectral radius of 0.80. The learning rate for the online algorithms is $\eta = 0.05$ (the delay window provides more stability to the RNN, so a bigger learning rate could be used).

As we have mentioned, this is a big simplification with respect to the pure sinewave generator of the previous simulation, and so the error will be much lower, as it can be appreciated in the error table 4.2.2. Nevertheless, the error of the BPDC models is much bigger than the one of ESN. As it can be viewed in figure 4.2.6, the BPDC-L model starts losing the dynamics in a few cycles, while the ESN model (figure 4.2.5) conserves it almost perfectly. Observing the weights in this simple example, it can be seen in figure 4.2.7 that the model trained with BPDC-L tends to the one trained with ESN. It has a dynamic much weaker, but its error is better than the one of the other variations. Moreover, the ESN model is centered mainly in the input units (that are the last 6 with the bias), so the corresponding weights are much bigger than the other ones. The importance of the inputs also explains the great performance of the simple MLP. Again, an increase of the learning rate will provide best results for the BPDC algorithms, but with the risk of divergence problems.

Cycles	**Steps**	**ESN**	**ESN***	**BPDC-L**	**BPDC-L***	**BPDC-S**	**BPDC-S***	**BPDC-C**	**BPDC-C***	**MLP**
4	100	$6.81e-07 \pm 4.68e-07$	$7.06e-07 \pm 5.24e-07$	$1.72e-01 \pm 4.62e-02$	$1.80e-01 \pm 3.91e-02$	$4.56e-01 \pm 9.80e-02$	$3.82e-01 \pm 1.37e-01$	$3.95e-01 \pm 1.01e-01$	$3.88e-01 \pm 1.28e-01$	$2.75e-04 \pm 2.57e-04$
8	200	$1.02e-06 \pm 7.27e-07$	$1.19e-06 \pm 8.63e-07$	$3.33e-01 \pm 9.73e-02$	$3.46e-01 \pm 8.37e-02$	$8.73e-01 \pm 1.18e-01$	$7.48e-01 \pm 2.47e-01$	$7.88e-01 \pm 1.80e-01$	$7.58e-01 \pm 2.25e-01$	$5.32e-04 \pm 5.88e-04$
12	300	$1.36e-06 \pm 1.02e-06$	$1.69e-06 \pm 1.20e-06$	$4.78e-01 \pm 1.44e-01$	$4.93e-01 \pm 1.26e-01$	$9.56e-01 \pm 7.86e-02$	$8.72e-01 \pm 2.37e-01$	$9.44e-01 \pm 1.76e-01$	$8.93e-01 \pm 2.23e-01$	$7.91e-04 \pm 9.26e-04$
16	400	$1.73e-06 \pm 1.34e-06$	$2.20e-06 \pm 1.54e-06$	$6.21e-01 \pm 1.83e-01$	$6.39e-01 \pm 1.66e-01$	$9.48e-01 \pm 5.04e-02$	$9.02e-01 \pm 2.12e-01$	$9.35e-01 \pm 1.64e-01$	$9.00e-01 \pm 1.83e-01$	$1.09e-03 \pm 1.32e-03$
20	500	$2.10e-06 \pm 1.67e-06$	$2.73e-06 \pm 1.89e-06$	$7.57e-01 \pm 2.18e-01$	$7.76e-01 \pm 2.04e-01$	$9.75e-01 \pm 5.76e-02$	$9.36e-01 \pm 1.96e-01$	$9.58e-01 \pm 1.57e-01$	$9.12e-01 \pm 1.36e-01$	$1.38e-03 \pm 1.68e-03$
24	600	$2.45e-06 \pm 1.98e-06$	$3.24e-06 \pm 2.22e-06$	$8.65e-01 \pm 2.40e-01$	$8.87e-01 \pm 2.33e-01$	$1.00e+00 \pm 6.99e-02$	$9.59e-01 \pm 1.83e-01$	$9.75e-01 \pm 1.40e-01$	$9.31e-01 \pm 1.06e-01$	$1.67e-03 \pm 2.06e-03$
28	700	$2.83e-06 \pm 2.31e-06$	$3.76e-06 \pm 2.57e-06$	$9.54e-01 \pm 2.44e-01$	$9.79e-01 \pm 2.50e-01$	$9.96e-01 \pm 5.95e-02$	$9.59e-01 \pm 1.67e-01$	$9.57e-01 \pm 1.14e-01$	$9.50e-01 \pm 8.15e-02$	$1.98e-03 \pm 2.46e-03$
32	800	$3.19e-06 \pm 2.62e-06$	$4.29e-06 \pm 2.92e-06$	$1.03e+00 \pm 2.39e-01$	$1.06e+00 \pm 2.61e-01$	$9.87e-01 \pm 3.92e-02$	$9.68e-01 \pm 1.55e-01$	$9.60e-01 \pm 9.96e-02$	$9.79e-01 \pm 7.37e-02$	$2.27e-03 \pm 2.83e-03$
36	900	$3.56e-06 \pm 2.94e-06$	$4.81e-06 \pm 3.25e-06$	$1.07e+00 \pm 2.27e-01$	$1.11e+00 \pm 2.65e-01$	$9.88e-01 \pm 2.02e-02$	$9.80e-01 \pm 1.42e-01$	$9.70e-01 \pm 8.80e-02$	$9.85e-01 \pm 6.79e-02$	$2.58e-03 \pm 3.22e-03$
40	1000	$3.94e-06 \pm 3.27e-06$	$5.33e-06 \pm 3.60e-06$	$1.10e+00 \pm 2.05e-01$	$1.14e+00 \pm 2.58e-01$	$9.85e-01 \pm 2.07e-02$	$9.84e-01 \pm 1.25e-01$	$9.79e-01 \pm 7.80e-02$	$9.90e-01 \pm 7.14e-02$	$2.89e-03 \pm 3.62e-03$

Table 4.2.2: Evolution of the normalized square error for Sinewave with Delay Input with respect to the length of the test sequence. The models marked with * are perturbed with white noise. The mean error is presented with its deviation. Both terms are computed over 20 runs.

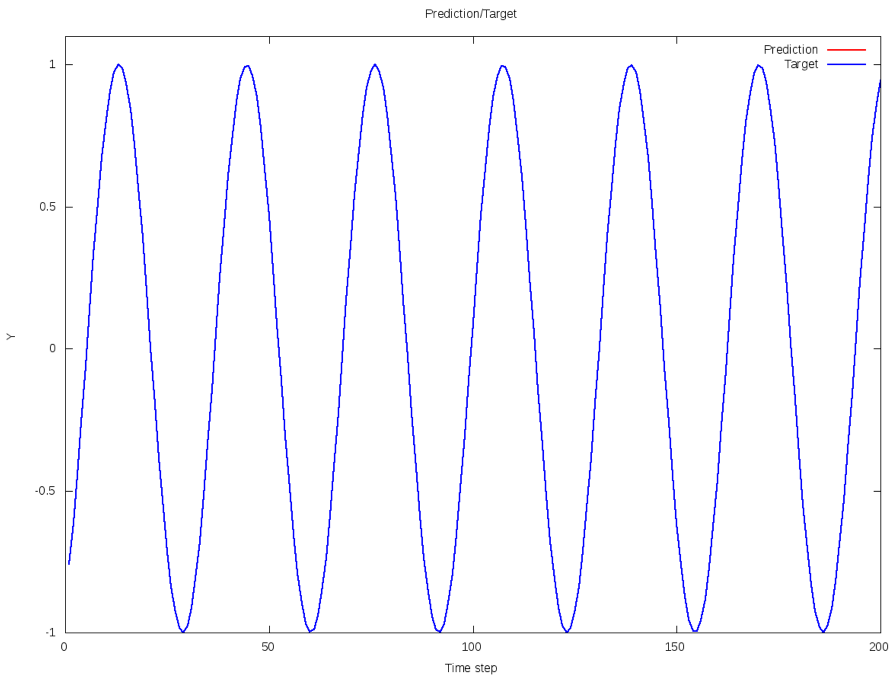


Figure 4.2.5: Prediction of the RNN versus the target (the sinewave) for the test of the sinewave generator with delay input with ESN. This task is easier than the one of the pure sinewave generator, so the model captures the required dynamic perfectly.

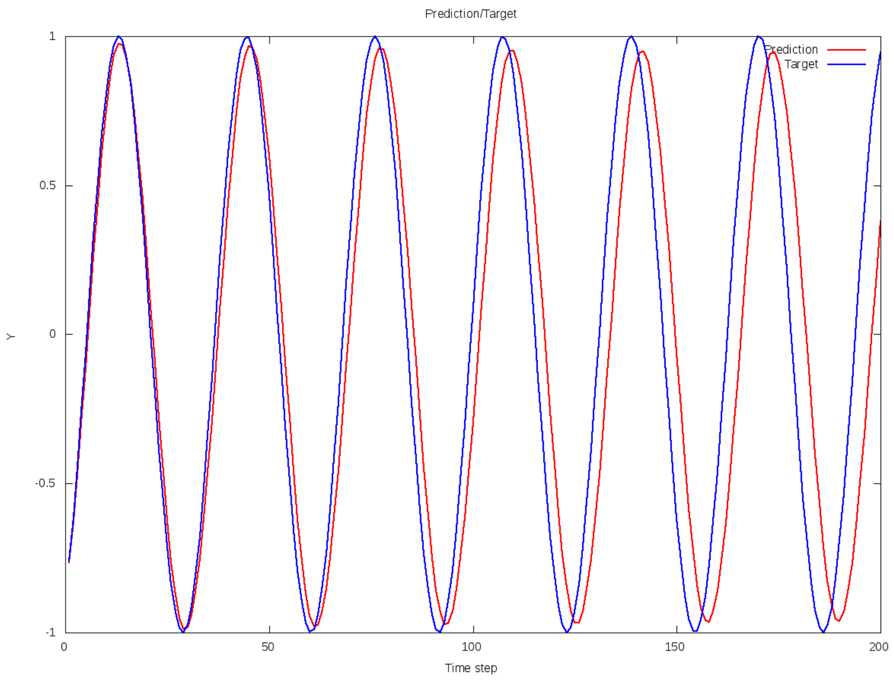


Figure 4.2.6: Prediction of the RNN versus the target (the sinewave) for the test of the sinewave generator with delay input with BPDC-L. The model is worse than the ESN one.

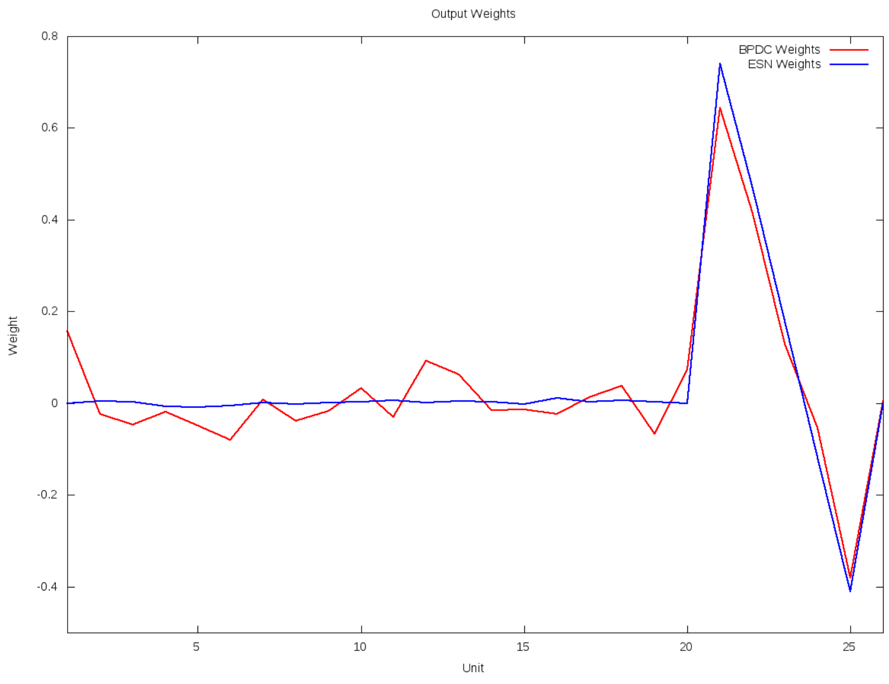


Figure 4.2.7: Weights of the ESN model versus the weights of the BPDC-L model. The first unit is the output, and the 6 last are the inputs (the 5 delays and the bias).

4.2.3 Modulated Pure Generator

The problem of this experiment is represented in figure 4.2.8. As it can be observed, the frequency of the output is modulated by the input, so the higher the input is, the faster the output is. This problem presents two difficulties. First, it is a pure generator (it has no delay windows), so the RNN has to conserve the dynamic in a generative way. Second, it uses an input that transforms the output.

Formally, the input of the problem at time k is given by

$$u(k) = \frac{\sin(0.01\pi i) + 1}{2},$$

and the corresponding output

$$y(k) = \frac{\sin(a(k)) + 1}{2},$$

with $a(k) = a(k-1) + 0.1 + 0.9u(k)$, and $a(0) = 0$.

The size of the training set is of 1200 steps. The original testing set is composed of the following 50 steps, although a much longer test is done for showing the degradation of the dynamic. A RNN of 100 units has been used, with a spectral radius of 0.80. The learning rate for the online algorithms was again $\eta = 0.01$.

As before, figure 4.2.9 represents the internal states, that show the frequency changes in function of the input. In figure 4.2.10 the test of the model is represented for ESN. This is the unique model that captures (initially) the dynamic of the problem. The rest of the algorithms have a trajectory too damped, so they only tend to the mean of the desired output, as it can be seen in figure 4.2.11 for the BPDC-L algorithm.

The evolution of the mean relative error in function of the number of testing steps is given in table 4.2.3. Notice that the BPDC models have less error than the ESN one. This is because, as explained above, the online algorithms do not produce models with an autonomous dynamic, so these RNNs fade to zero quickly. For this reason the error of these models is almost as big as the one of the trivial predictor (i.e. they have an error equal to the squared standard deviation). On the other hand, the ESN model is too unstable, so it conserves the desired dynamic for a few cycles, and then it starts to oscillate quickly, producing a bigger error.

Steps	**ESN**	**ESN***	**BPDC-L**	**BPDC-L***	**BPDC-S**	**BPDC-S***	**BPDC-C**	**BPDC-C***
100	$3.99e{+}00 \pm 3.69e+00$	$4.13e{+}00 \pm 2.69e+00$	$9.47e-01 \pm 1.05e-02$	$9.49e-01 \pm 1.04e-02$	$9.34e-01 \pm 8.86e-03$	$9.38e-01 \pm 1.18e-02$	$9.57e-01 \pm 3.78e-03$	$9.57e-01 \pm 4.12e-03$
200	$8.68e{+}00 \pm 4.93e+00$	$8.79e{+}00 \pm 2.91e+00$	$9.36e-01 \pm 7.38e-03$	$9.38e-01 \pm 5.84e-03$	$9.45e-01 \pm 5.74e-03$	$9.46e-01 \pm 7.84e-03$	$9.38e-01 \pm 1.66e-03$	$9.39e-01 \pm 1.67e-03$
300	$1.03e{+}01 \pm 5.36e+00$	$1.01e{+}01 \pm 2.98e+00$	$9.51e-01 \pm 7.06e-03$	$9.52e-01 \pm 5.61e-03$	$9.59e-01 \pm 4.82e-03$	$9.60e-01 \pm 7.35e-03$	$9.56e-01 \pm 2.17e-03$	$9.56e-01 \pm 2.29e-03$
400	$1.15e{+}01 \pm 5.55e+00$	$1.11e{+}01 \pm 3.09e+00$	$9.45e-01 \pm 6.80e-03$	$9.47e-01 \pm 4.64e-03$	$9.59e-01 \pm 5.26e-03$	$9.59e-01 \pm 7.43e-03$	$9.47e-01 \pm 1.54e-03$	$9.47e-01 \pm 1.55e-03$
500	$1.20e{+}01 \pm 5.60e+00$	$1.15e{+}01 \pm 3.13e+00$	$9.44e-01 \pm 6.85e-03$	$9.45e-01 \pm 4.92e-03$	$9.55e-01 \pm 4.59e-03$	$9.56e-01 \pm 7.15e-03$	$9.49e-01 \pm 1.96e-03$	$9.49e-01 \pm 2.04e-03$
600	$1.25e{+}01 \pm 5.67e+00$	$1.19e{+}01 \pm 3.20e+00$	$9.42e-01 \pm 6.90e-03$	$9.43e-01 \pm 4.74e-03$	$9.57e-01 \pm 5.06e-03$	$9.57e-01 \pm 7.32e-03$	$9.45e-01 \pm 1.58e-03$	$9.45e-01 \pm 1.58e-03$
700	$1.28e{+}01 \pm 5.70e+00$	$1.20e{+}01 \pm 3.22e+00$	$9.47e-01 \pm 7.01e-03$	$9.49e-01 \pm 4.96e-03$	$9.61e-01 \pm 4.67e-03$	$9.61e-01 \pm 7.17e-03$	$9.52e-01 \pm 1.82e-03$	$9.52e-01 \pm 1.85e-03$
800	$1.31e{+}01 \pm 5.74e+00$	$1.23e{+}01 \pm 3.27e+00$	$9.44e-01 \pm 6.84e-03$	$9.46e-01 \pm 4.63e-03$	$9.60e-01 \pm 4.99e-03$	$9.60e-01 \pm 7.29e-03$	$9.47e-01 \pm 1.55e-03$	$9.48e-01 \pm 1.54e-03$

Table 4.2.3: Evolution of the normalized square error for Modulated Sinewave with respect to the length of the test sequence. The models marked with * are perturbed with white noise. The mean error is presented with its deviation. Both terms are computed over 20 runs.

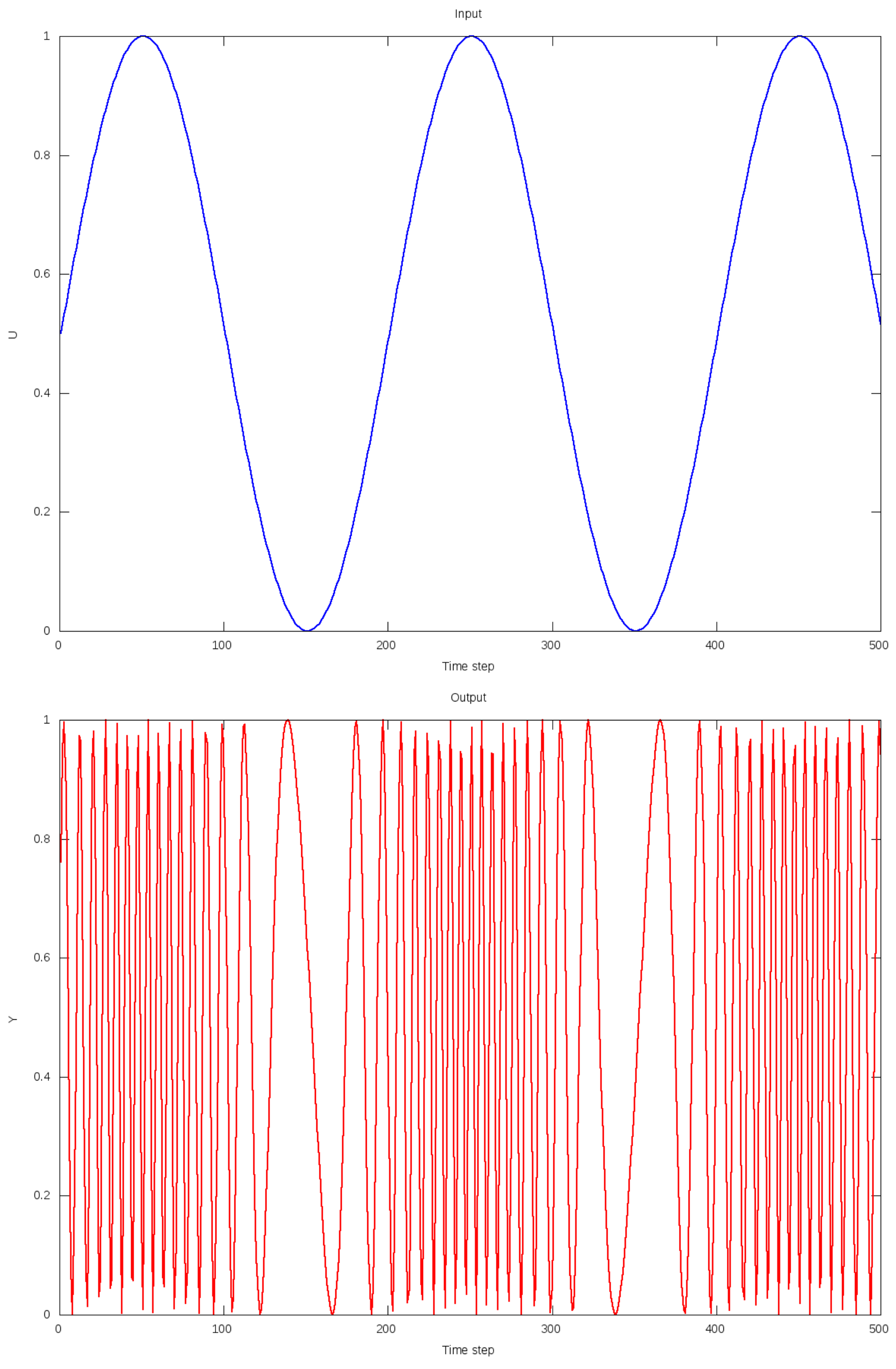


Figure 4.2.8: Input versus output of the modulated pure sinewave generator trained with ESN.

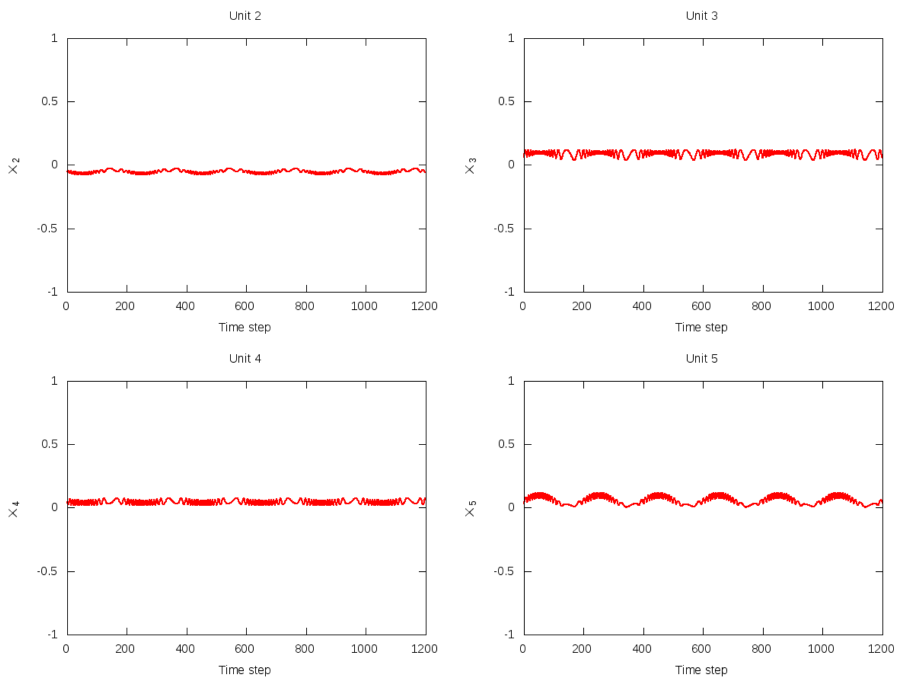


Figure 4.2.9: Activation of some of the first 4 internal (not output) units during training, for the modulated pure sinewave generator with ESN.

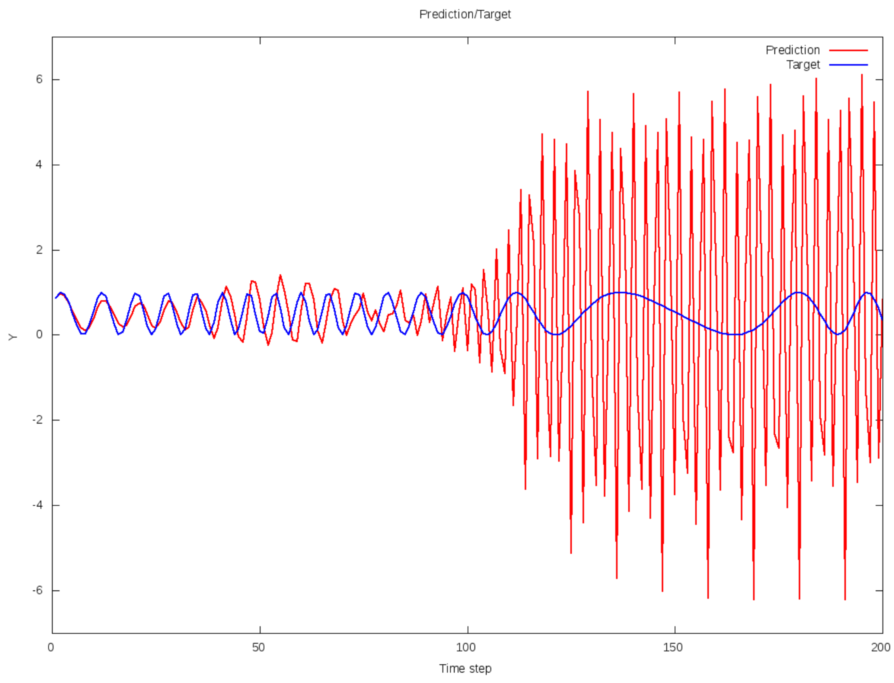


Figure 4.2.10: Prediction of the RNN versus the target for the test of the modulated pure sinewave generator with ESN. The model follows the dynamic of the problem in the initial states, but at the end of the sequence it loses the desired trajectory.

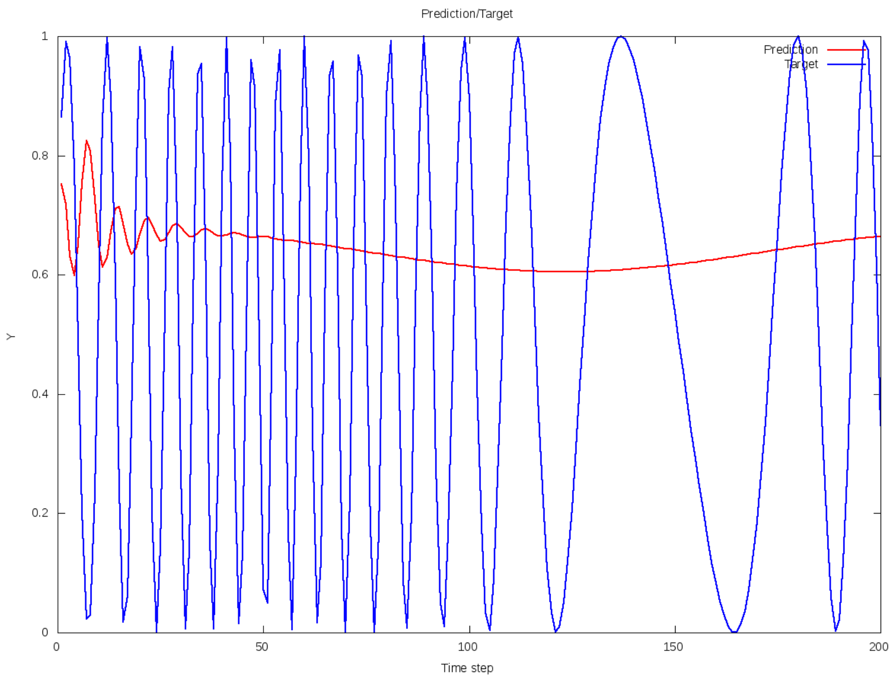


Figure 4.2.11: Prediction of the RNN versus the target for the test of the modulated pure sinewave generator with BPDC-L. The model soften the dynamic too much.

4.3 Mackey-Glass

The Mackey-Glass attractor is a classical problem in the field of temporal series prediction [29]. It consists on modeling the semi-chaotic trajectory which is a solution of the differential equation

$$y'(t) = -0.1y(t) + \frac{0.2y(t-\tau)}{1+y(t-\tau)^{10}}.$$

In this case, the value $\tau = 17$ will be used. This trajectory presents the form shown in figure 4.3.1.

In this simulation, a training sequence of 3000 steps will be given. The error will be measured on the next 500 steps. This is quite a difficult problem, because the RNN has to present an autonomous complex trajectory. A delay windows of the last 10 steps is used. A RNN of 100 units with a spectral radius of 0.80 has been applied to this task. The learning rate was $\eta = 0.01$.

The results presented in table 4.3.1 show that the best model for this task is again ESN. As it could be seen in the discussion, section 4.5, this could be due to the magnitude of the weighs, because the trajectory of the RNN trained with the BPDC algorithms are again too soft, so they fade to zero and do not capture the complex details of the trajectory (only the main dynamic). This can be seen in figure 4.3.4. In figure 4.3.3 the results for the ESN model are shown. Figures 4.3.6 and 4.3.5 show the last steps of the test sequence. Although the error is almost the same in both models, the ESN one conserves the dynamic (but it is shifted with respect to the target) while the BPDC-C model is completely damped.

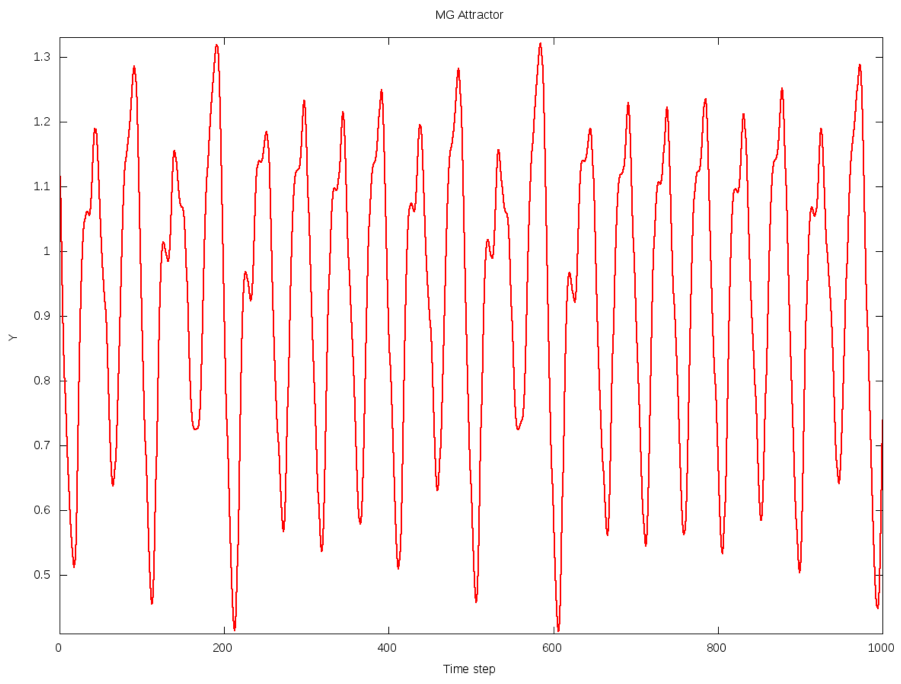


Figure 4.3.1: Mackey-Glass trajectory for $\tau = 17$.

Steps	ESN	ESN*	BPDC-L	BPDC-L*	BPDC-S	BPDC-S*	BPDC-C	BPDC-C*	MLP
100	$1.11e-01 \pm 2.31e-02$	$1.24e-01 \pm 2.64e-02$	$6.70e-01 \pm 8.06e-02$	$7.52e-01 \pm 1.78e-01$	$6.35e-01 \pm 1.68e-02$	$6.29e-01 \pm 2.37e-02$	$6.27e-01 \pm 2.29e-02$	$6.24e-01 \pm 1.52e-02$	$5.94e-01 \pm 1.89e-01$
200	$2.03e-01 \pm 3.40e-02$	$2.22e-01 \pm 5.03e-02$	$7.50e-01 \pm 5.56e-02$	$9.44e-01 \pm 2.76e-01$	$7.44e-01 \pm 6.97e-03$	$7.40e-01 \pm 1.17e-02$	$7.36e-01 \pm 1.11e-02$	$7.35e-01 \pm 7.84e-03$	$7.72e-01 \pm 1.68e-01$
300	$3.24e-01 \pm 5.91e-02$	$3.41e-01 \pm 5.00e-02$	$8.13e-01 \pm 3.72e-02$	$1.22e+00 \pm 4.85e-01$	$7.99e-01 \pm 4.57e-03$	$7.97e-01 \pm 7.53e-03$	$7.94e-01 \pm 7.16e-03$	$7.94e-01 \pm 4.95e-03$	$8.72e-01 \pm 1.09e-01$
400	$5.24e-01 \pm 1.19e-01$	$5.70e-01 \pm 1.34e-01$	$8.21e-01 \pm 2.89e-02$	$1.52e+00 \pm 8.24e-01$	$8.07e-01 \pm 3.36e-03$	$8.05e-01 \pm 5.61e-03$	$8.04e-01 \pm 5.35e-03$	$8.03e-01 \pm 3.66e-03$	$8.73e-01 \pm 1.08e-01$
500	$6.16e-01 \pm 1.59e-01$	$2.06e+26 \pm 8.96e+26$	$8.31e-01 \pm 2.46e-02$	$3.09e+00 \pm 5.10e+00$	$8.16e-01 \pm 2.72e-03$	$8.15e-01 \pm 4.52e-03$	$8.14e-01 \pm 4.30e-03$	$8.13e-01 \pm 2.94e-03$	$8.97e-01 \pm 1.29e-01$

Table 4.3.1: Evolution of the normalized square error for Mackey-Glass with respect to the length of the test sequence. The models marked with * are perturbed with white noise. The mean error is presented with its deviation. Both terms are computed over 20 runs.

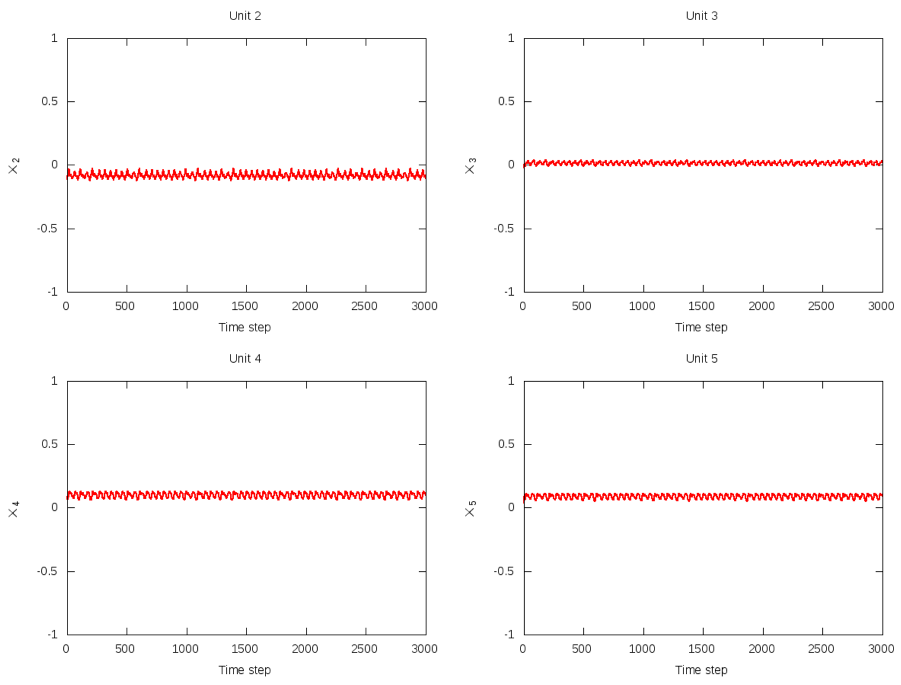


Figure 4.3.2: Activation of some of the first 4 internal (not output) units during training, for the Mackey-Glass problem with ESN.

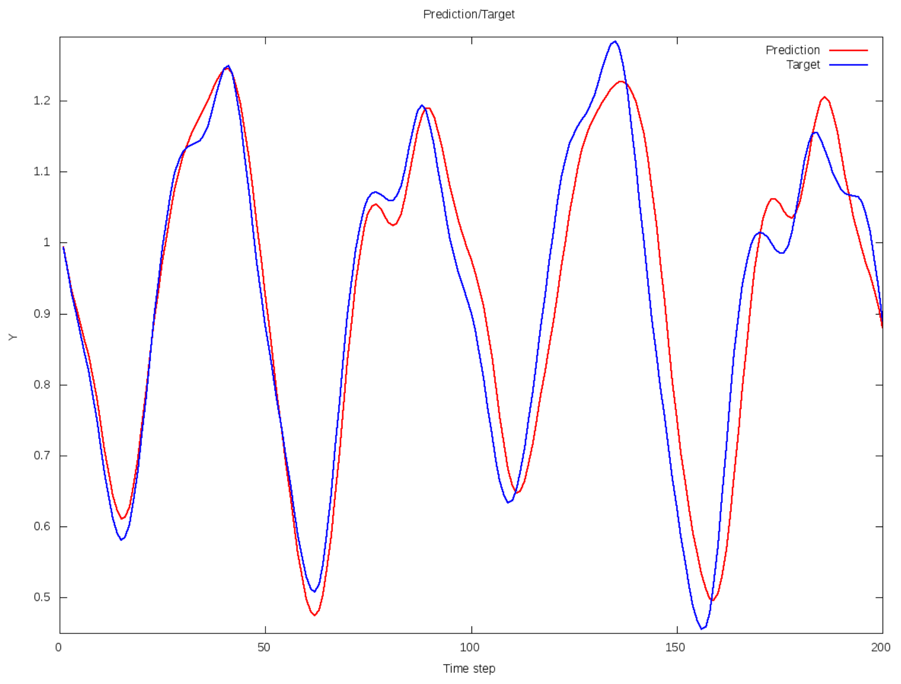


Figure 4.3.3: Prediction of the RNN versus the target for the test of the Mackey-Glass attractor with ESN. The model captures the required dynamic, but it starts to lose the trajectory when it has been running autonomously for a long time.

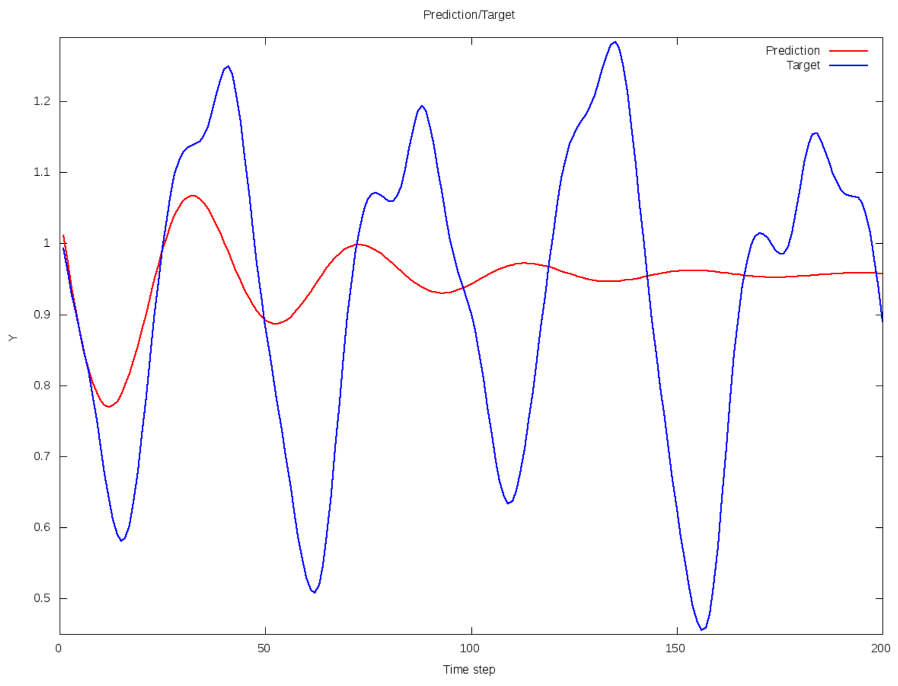


Figure 4.3.4: Prediction of the RNN versus the target for the test of the Mackey-Glass attractor with BPDC-C. The model captures only the main dynamic, softening the trajectory and fading to zero.

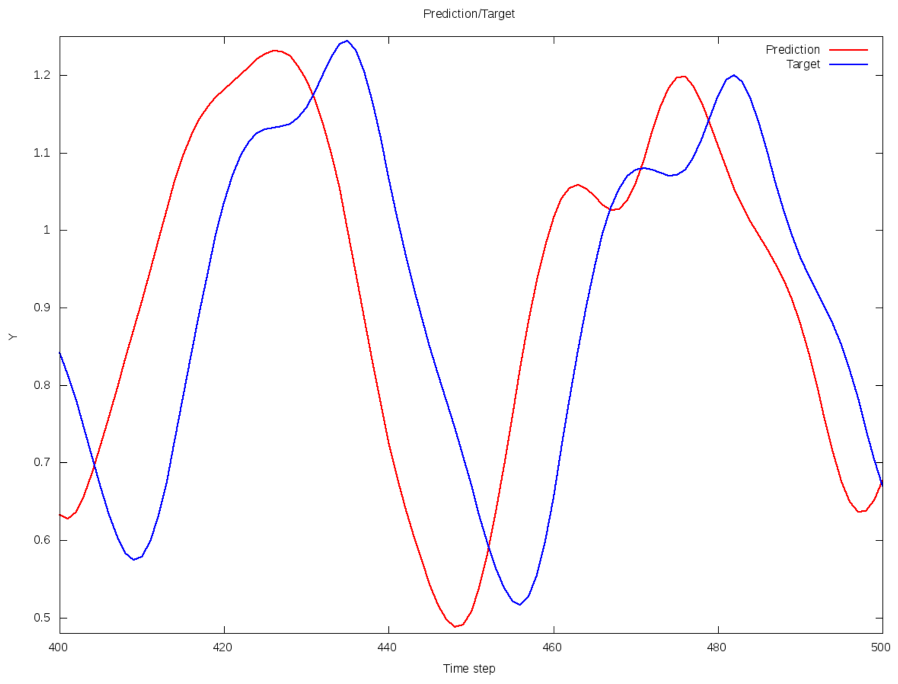


Figure 4.3.5: Last steps of the test sequence for the Mackey-Glass attractor with ESN. The model conserves the dynamic, but it is shifted with respect to the target.

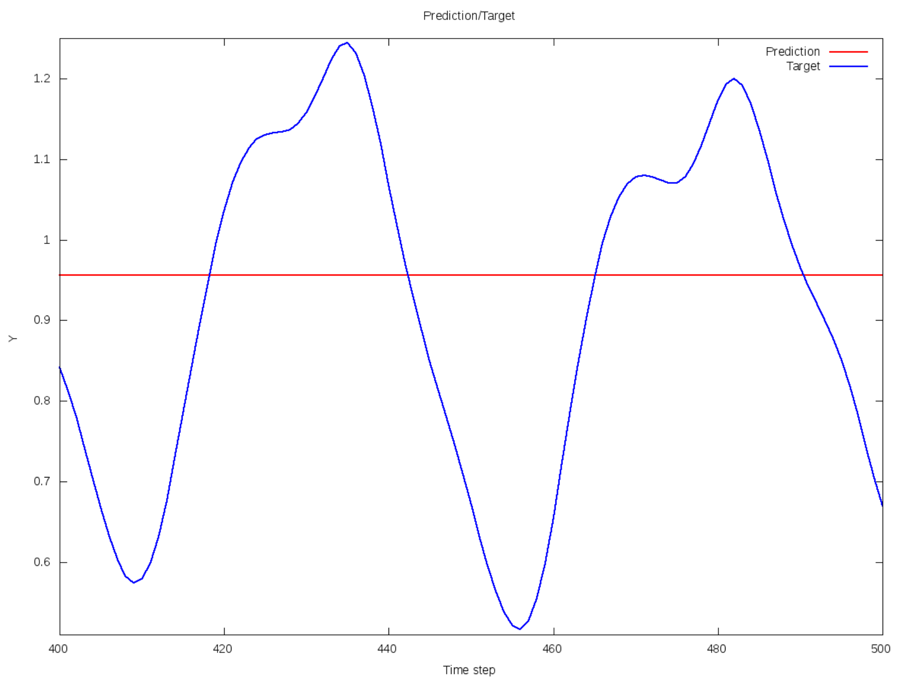


Figure 4.3.6: Last steps of the test sequence for the Mackey-Glass attractor with BPDC-C. The model has converged to the mean of the target.

4.4 Application to Wind Power Forecasting

In this section, a RNN will be applied to the problem of wind power forecasting. The time series of power production of a wind power station presents certain periodic behaviour, so some hidden patterns can be discovered by the model (see figure 4.4.1 to see this trajectory).

The data used in this simulations correspond to the power production of all the country (so it is a somehow regular series) for 7 months. The first 6 months will be used to train the models, and the last one to test it. Predictions in a autogenerative way will be emitted for 48 hours starting at the end of the training set.

This simulation consists on the forecasting of the wind power itself. For this task, a delay window of the last 24 productions will be used to predict the next hour production. So the system receives 24 inputs with the last 24 productions (i.e. the information of the last day). When testing the model, the RNN will be iterated in an autogenerative way for 48 steps, so the complete prediction of the next two days will be used to measure the error. This procedure will be repeated for 1 month, following the steps:

- For each day of the testing month (composed by 30 days).
  - Train the model with the previous 6 months data (180 days). One pattern is obtained for each hour, so 4320 points will be used to train the RNN.
  - Test the model iterating the RNN for the following 2 days (48 steps).

A RNN of 100 units has been used, with a spectral radius of 0.80. The learning rate for the online algorithms was $\eta = 0.01$. The models has been trained with a noise of deviation $\sigma = 0.01$, that permits the ESN model to converge and stabilize over the required two days of test sequence. Simulations without noise showed that the ESN model diverges very quickly from the desired trajectory, producing a much bigger error.

In table $4.4.1$ the comparative of the 4 main algorithms is given. Here, the error used is the mean absolute error normalized using the total power of the considered plants. As it can be observed, the BPDC-C algorithm and the BPDC-L one present the best results. The ESN improves slightly the results of the MLP, that are at the same level as the ones of BPDC-L.

Day	ESN*	BPDC-L*	BPDC-S*	BPDC-C*	MLP
1	$5.58e-02 \pm 3.89e-04$	$5.84e-02 \pm 1.63e-03$	$5.62e-02 \pm 6.98e-04$	$5.59e-02 \pm 9.75e-04$	$5.76e-02 \pm 5.53e-04$
2	$9.44e-02 \pm 4.71e-04$	$9.74e-02 \pm 1.90e-03$	$9.48e-02 \pm 5.71e-04$	$9.43e-02 \pm 6.37e-04$	$9.50e-02 \pm 6.42e-04$
3	$4.10e-02 \pm 5.41e-04$	$4.66e-02 \pm 7.53e-04$	$4.16e-02 \pm 7.83e-04$	$4.22e-02 \pm 6.93e-04$	$4.16e-02 \pm 5.37e-04$
4	$6.36e-02 \pm 7.87e-04$	$6.60e-02 \pm 1.75e-03$	$6.45e-02 \pm 1.50e-03$	$6.48e-02 \pm 1.92e-03$	$6.38e-02 \pm 2.63e-04$
5	$7.54e-02 \pm 1.07e-03$	$8.33e-02 \pm 3.88e-03$	$7.94e-02 \pm 1.39e-03$	$7.94e-02 \pm 8.50e-04$	$7.55e-02 \pm 5.23e-04$
6	$1.01e-01 \pm 1.05e-03$	$8.94e-02 \pm 3.22e-03$	$9.14e-02 \pm 1.36e-03$	$9.21e-02 \pm 2.86e-03$	$9.96e-02 \pm 4.30e-04$
7	$1.21e-01 \pm 1.17e-03$	$1.17e-01 \pm 2.23e-03$	$1.13e-01 \pm 2.25e-03$	$1.12e-01 \pm 1.01e-03$	$1.21e-01 \pm 4.11e-04$
8	$6.19e-02 \pm 9.95e-04$	$6.92e-02 \pm 5.13e-03$	$6.34e-02 \pm 2.59e-03$	$6.27e-02 \pm 1.68e-03$	$6.28e-02 \pm 4.24e-04$
9	$1.32e-01 \pm 5.53e-04$	$1.29e-01 \pm 1.40e-03$	$1.28e-01 \pm 6.65e-04$	$1.29e-01 \pm 9.14e-04$	$1.32e-01 \pm 4.77e-04$
10	$6.15e-02 \pm 1.15e-03$	$7.14e-02 \pm 5.33e-03$	$6.10e-02 \pm 2.42e-03$	$5.97e-02 \pm 2.81e-03$	$6.42e-02 \pm 8.00e-04$
11	$6.40e-02 \pm 4.51e-04$	$5.93e-02 \pm 1.73e-03$	$6.07e-02 \pm 1.40e-03$	$6.07e-02 \pm 7.80e-04$	$6.46e-02 \pm 6.20e-04$
12	$5.60e-02 \pm 4.72e-04$	$6.01e-02 \pm 2.97e-03$	$5.71e-02 \pm 1.55e-03$	$5.63e-02 \pm 1.22e-03$	$5.70e-02 \pm 5.86e-04$
13	$7.38e-02 \pm 6.93e-04$	$8.10e-02 \pm 1.84e-03$	$7.63e-02 \pm 7.79e-04$	$7.64e-02 \pm 8.15e-04$	$7.47e-02 \pm 3.01e-04$
14	$6.35e-02 \pm 1.88e-03$	$6.51e-02 \pm 4.35e-03$	$6.15e-02 \pm 4.52e-03$	$6.12e-02 \pm 3.20e-03$	$6.32e-02 \pm 5.38e-04$
15	$1.24e-01 \pm 4.95e-04$	$1.07e-01 \pm 2.75e-03$	$1.15e-01 \pm 7.34e-04$	$1.15e-01 \pm 9.40e-04$	$1.24e-01 \pm 5.18e-04$
16	$5.18e-02 \pm 1.42e-03$	$5.60e-02 \pm 4.31e-03$	$5.26e-02 \pm 1.60e-03$	$5.23e-02 \pm 1.89e-03$	$5.28e-02 \pm 4.54e-04$
17	$7.61e-02 \pm 1.13e-03$	$6.92e-02 \pm 3.33e-03$	$7.19e-02 \pm 1.97e-03$	$7.26e-02 \pm 1.47e-03$	$7.58e-02 \pm 4.63e-04$
18	$9.71e-02 \pm 1.34e-03$	$8.49e-02 \pm 2.51e-03$	$8.97e-02 \pm 1.48e-03$	$9.05e-02 \pm 2.20e-03$	$9.60e-02 \pm 4.09e-04$
19	$1.01e-01 \pm 1.76e-03$	$9.96e-02 \pm 3.52e-03$	$9.60e-02 \pm 3.64e-03$	$9.47e-02 \pm 2.10e-03$	$1.04e-01 \pm 6.93e-04$
20	$4.92e-02 \pm 1.76e-03$	$6.15e-02 \pm 7.85e-03$	$5.23e-02 \pm 4.08e-03$	$5.30e-02 \pm 3.03e-03$	$5.17e-02 \pm 8.56e-04$
21	$3.77e-02 \pm 2.58e-03$	$5.34e-02 \pm 9.28e-03$	$4.03e-02 \pm 4.27e-03$	$3.91e-02 \pm 2.64e-03$	$3.90e-02 \pm 7.75e-04$
22	$6.45e-02 \pm 1.41e-03$	$7.16e-02 \pm 4.91e-03$	$6.49e-02 \pm 1.30e-03$	$6.52e-02 \pm 1.33e-03$	$6.60e-02 \pm 6.26e-04$
23	$6.89e-02 \pm 4.02e-04$	$7.65e-02 \pm 3.16e-03$	$7.08e-02 \pm 4.87e-04$	$7.09e-02 \pm 6.08e-04$	$6.99e-02 \pm 4.47e-04$
24	$5.01e-02 \pm 7.74e-04$	$5.26e-02 \pm 3.35e-03$	$4.94e-02 \pm 2.61e-03$	$4.87e-02 \pm 1.11e-03$	$5.04e-02 \pm 5.62e-04$
25	$4.69e-02 \pm 1.25e-03$	$5.28e-02 \pm 4.79e-03$	$4.90e-02 \pm 1.61e-03$	$4.89e-02 \pm 2.11e-03$	$4.67e-02 \pm 4.52e-04$
26	$5.03e-02 \pm 1.15e-03$	$4.51e-02 \pm 2.63e-03$	$4.72e-02 \pm 1.93e-03$	$4.76e-02 \pm 1.36e-03$	$5.04e-02 \pm 2.19e-04$
27	$9.82e-02 \pm 6.21e-04$	$9.66e-02 \pm 1.71e-03$	$9.11e-02 \pm 1.05e-03$	$9.15e-02 \pm 1.07e-03$	$9.94e-02 \pm 5.97e-04$
28	$7.96e-02 \pm 2.37e-03$	$8.00e-02 \pm 7.32e-03$	$7.41e-02 \pm 4.22e-03$	$7.29e-02 \pm 2.09e-03$	$8.12e-02 \pm 3.72e-04$
29	$5.83e-02 \pm 1.64e-03$	$5.42e-02 \pm 1.85e-03$	$5.36e-02 \pm 1.13e-03$	$5.48e-02 \pm 3.46e-03$	$5.73e-02 \pm 2.94e-04$
30	$1.03e-01 \pm 4.12e-04$	$9.84e-02 \pm 2.24e-03$	$9.91e-02 \pm 8.00e-04$	$9.94e-02 \pm 7.20e-04$	$1.04e-01 \pm 5.50e-04$
Mean	$7.41e-02 \pm 2.52e-02$	$7.51e-02 \pm 2.15e-02$	$7.22e-02 \pm 2.27e-02$	$7.21e-02 \pm 2.27e-02$	$7.47e-02 \pm 2.50e-02$

Table 4.4.1: Mean absolute error (normalized over the total power of the considered plants) for each of the test days. The mean error is presented with its deviation. Both terms are computed over 20 runs. The error over all the month and its deviation is presented in the last row.

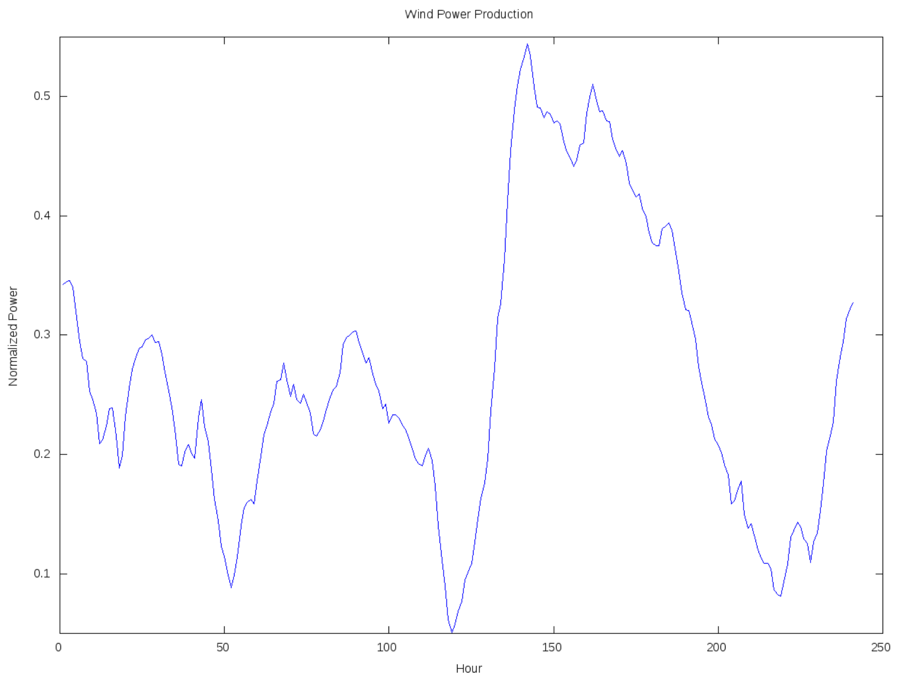


Figure 4.4.1: Wind power for 10 days (240 hours). A certain periodicity can be observed in the trajectory.

4.5 Discussion and Remarks

Some important remarks related with the previous simulations will be given in this section. First, the BPDC algorithm does not require, in principle, the use of *teacher forcing*. Nevertheless, the convergence of the weights using the real output of the RNN can be much slower. Moreover, a comparative between the ESN approach and the BPDC one can be done easily if the RNN presents the same dynamic during training. This is achieved using the desired output instead of the real one even for BPDC.

The addition of noise has demonstrated to be an important regularization technique. In this work we have disturbed the internal states of the RNN through gaussian white noise. Although further research is needed, the first results seem to suggest that also in this context the noise regularizes the dynamic. In this sense, the deviation of the injected perturbation determines the magnitude of the weight absolute values versus the complexity of the model. Complex dynamics can be achieved if no noise is added, but the resulting trajectory can be too unstable. On the other hand, if the noise is too big the RNN dynamic will fade to zero. In fact, in the simple case in which the correlation matrix is invertible, if the noise is big enough then the correlation matrix changes completely, and it becomes approximately a multiple (with a big constant σ that is the deviation of the noise) of the identity, $O \approx \sigma I$. So any information about the variables gets lost, and the regression makes no sense. Moreover, since the inverse of the correlation matrix is approximated by the inverse of this multiple of the identity, the weights will be equal to the desired outputs divided by the noise deviation, and they will tend to zero:

$$w_s = O^{+} Y_s \approx \frac{1}{\sigma} Y_s \to 0.$$

As this technique controls the magnitude of the weights, it could also be used as a way of avoiding overfitting.

The washout of the data is generally done ignoring the first $p\%$ of the training sequence steps. Nevertheless, we have opted for iterating the RNN over a complete cycle of the training set before starting the training itself. This means that for the ESN algorithm 2 iterations have been done. At the end of the second one, the internal states matrix of this last cycle are used to estimate the output weights. In the case of BPDC, one complete cycle have been done, iterating the RNN and using *teacher forcing*, before the online update of the weights starts. We have observed in all the experiments that in only one cycle the RNN stabilizes completely, so the following iterations will produce the same patterns (with the little variations due to the addition of noise).

The comparative between the output weights obtained by the BPDC-L algorithm versus the pure ESN algorithm shows some interesting results. The ESN algorithm use the pseudoinverse, in this case implemented using the SVD decomposition of the matrix. This is the mathematical generalized definition of the pseudoinverse (see, for example, [30]), and it finds the solution of minimal norm even when the matrix is not full rank. On the other hand, a further study of the behaviour of the LMS filter when the covariance matrix is singular could clarify why the BPDC-L algorithm seems to find a minimal in the error surface with smaller weights. Anyway, if the reservoir is considered to be the whole RNN (including the output unit), then these weights can modify the effective spectral radius. This could justify why the ESN presents a more active trajectory, while the RNN trained with BPDC fades to zero quickly.

The above consideration is also appreciated in the experiment of the wind power production. Here, the ESN RNN (if no noise is used to regularize it) diverges in a few steps, while BPDC produced a much more stable RNN. Nevertheless, these are still only hypotheses, and further work is needed to understand this complex behaviour.

Chapter 5

Discussion, Conclusions and Further Work

5.1 Discussion

In this work, a state of the art in the field of RNNs has been discussed. In particular, the paradigm of Reservoir Computing has been studied. For the complete understanding of this approach, the evolution of the learning algorithms of RNN has been studied. First, classical algorithm based on the computation of the exact error gradient emerged. These techniques were inspired in classical approaches such as the algorithms for the MLPs. They suffered from some important disadvantages, as their computational cost. Then, a new point of view given by Atiya and Parlos permitted the unification of most of the previous algorithms, and derived into the new APRL algorithm. Some simplifications on the online learning rule of APRL result in the BPDC algorithm. The analysis of the dynamic of the weights of APRL and the update equation of BPDC suggested the adaptation only of a small part of the RNN. This is a possible way to arrive at the Reservoir Computing approach.

The ESN technique, based on the training only of the output weights of a RNN, can be considered, with the LSM approach, the basis of the RC paradigm. In this work, a complete description of this algorithm has been given. The simulations showed that this model, despite its simplicity, works very well. This is surprising because of the random nature of the initial state or the weight matrix initialization, and specially for the lack of reservoir weights adjustment. Some mathematical considerations, like the ESP, support that under certain conditions this models depends only on the inputs and desired outputs, so the initial state has (theoretically) no influence. But the reservoir initialization is still a crucial point for this approach. In fact, currently the RC paradigm assumes only a functional separation between two parts of the RNN. One (big) part, the reservoir, is used as a tool that projects the temporal information into a multidimensional space, and the other (smaller) part combines this information to rebuild the desired output. So a specialized training of the reservoir is also admitted.

Another remarkable point is the difficulty of tuning the parameters in this kind of models. This was one of the disadvantages of the classical approaches, and in ESN is partially solved because the only important parameter to be set is the spectral radius of the reservoir. Nevertheless, the algorithm of BPDC and its variations have a bigger complexity because of their online nature. That is, it is necessary to choose a correct learning rate, and the number of epochs needed to train the model. So the parameters require a certain experience with this algorithm, or some kind of metatraining, to achieve the best results.

In the experimental part of this work a "continuous" transformation from the classical approaches to the ESN has been presented. The starting point is the complete BPDC algorithm. This technique is based on the Atiya-Parlos reformulation (so it is not really one of the first approaches to RNN training, as it could be BPTT or RTRL) and it updated all the weights of the RNN. Then, the mentioned coupled update of the reservoir weights suggest to adapt only the output weights. This is the BPDC without reservoir simplification. If we focus our attention in the BPDC output weights update, then only a term makes it different from the LMS filter. Deleting this term drives to what we have called the BPDC-L algorithm. Assuming that the ESP of the RNN is satisfied, that a big enough washout is done and that the net is teacher driven during training, then the reservoir starts to produce the same results at each epoch. So the internal states of the RNN arise as a set of fixed patterns, and the BPDC-L algorithm can be seen in fact as the online version of a simple linear regression. Changing this online regression by a batch one (this is, obtaining the output weights using the pseudoinverse of the internal states matrix) constitutes the ESN algorithm.

5.2 Further Work

In this work, the transition from the classical algorithm to ESN has been described conceptually. The experimental comparative between the algorithms from complete BPDC to ESN has not been too successful as it has been described in section 4.5. The first extension to this research is to improve the simulations, trying to explain the path BPDC $\rightarrow$ ESN. The introduction of more scenarios, or other state of the art time series, could help in this task. Moreover, a further analysis about the theoretical relationship between these algorithms is also needed. Maybe the step from online algorithm to batch ones produces any "discontinuity" in the mentioned path, what will partially explain the results.

It is important to realize that the best results in the experiments have been achieved using a spectral radius near to 1. In this region, the network has an almost chaotic behaviour. It has been claimed in [18] that in this region, the edge of chaos, the network has more computational power. A more exhaustive study of the behaviour of the RNN at the edge of chaos can clarify some results of the simulations of this work. On the other hand, the relationship between the regularization (through the addition of noise) and the spectral radius of the trained network can be another interesting extension of this research.

As it has been explained before, an analysis of the different architecture selection techniques can be done. Although some insights on the reservoir adaptation have been tackled in this work with the IP techniques, this is still an expanding field with continuous contributions from the biological branch of the neuroscience. In particular, an interesting future work could be the use of some evolutive approaches, like the construction of bigger nets and the posterior pruning of the least important units (considering its correlation with the output or more complex measures).

The experimental work can be extended with experiments using the IP adaptation rule. The study of this results and the regularity of the internal states matrix can clarify if this technique in fact increases the information transmitted by the net. Moreover, the parameters obtained with the IP rule for the activation functions can also indicate which units are important and which ones only act as binary inputs. This could be the basis of another pruning method.

Finally, this work has required the implementation of these complex algorithms. There is not too much open code available, and most of it is implemented in Matlab. The publication of an open source toolbox in C is another future extension of this work.

Appendix A

Auxiliar Theorems and Lemmas

A.1 Takens Theorem

As mentioned in the introduction, Takens Theorem [1] can be used as a first approach to transform predictions for a dynamical system into a classical regression problem. In this section, the theorem will be cited and a brief description of its implications will be given.

A.1.1 Theorem

Theorem 1 (Takens) *Let M be a compact manifold of dimension m. For pairs (ϕ, y), with $\phi \in$ Diff$^2(M), y \in C^2(M, \mathbb{R})$, it is a generic property that the map $\Phi_{2m+1,(\phi,y)} : M \to \mathbb{R}^{2m+1}$, defined by*

$$\Phi_{2m+1,(\phi,y)}(x) = (y(x), y(\phi(x)), \dots, y(\phi^{2m}(x))),$$

is an embedding.

Here the term "generic" means that this property is verified in a dense and open subset $U \subset$ Diff$^2(M) \times C^2(M, \mathbb{R})$.

A.1.2 Discussion

The functions $y \in C^2(M, \mathbb{R})$ are known as measurement functions, and the maps $\Phi_{(\phi,y)}$ as reconstruction maps.

The measurement function provides observations of the current state of the hidden dynamical system, given by the orbit of ϕ

$$(x_0, x_1, x_2, \dots) = (x_0, \phi(x_0), \phi^2(x_0), \dots).$$

For the prediction of a temporal series, a hidden state x_i with dynamic $x_{i+1} = \phi(x_i)$ can be assumed, so the desired output is given by the readout $y_i = y(x_i)$. The goal is to predict the next value of the series, y_{i+k}, given the previous observations $(\dots, y_{i+k-2}, y_{i+k-1})$. Using Takens Theorem, the reconstruction map $\Phi_{k,(\phi,y)}(x), k > 2m$[1] permits to transform the hidden space M to the observation space $\mathbb{R}^k$, so the prediction can be done as indicated in next scheme:

$$\begin{tikzcd}
x_i \arrow[r, "\phi"] \arrow[d, shift right, "\Phi_{k,(\phi,y)}"'] & x_{i+1} \arrow[d, shift right, "\Phi_{k,(\phi,y)}"'] \\
(y_i, \dots, y_{i+k-1}) \arrow[r, "\Phi_{k,(\phi,y)} \circ \phi \circ \Phi^{-1}_{k,(\phi,y)}"] \arrow[u, shift right, "\Phi^{-1}_{k,(\phi,y)}"'] & (y_{i+1}, \dots, y_{i+k}) \arrow[u, shift right, "\Phi^{-1}_{k,(\phi,y)}"']
\end{tikzcd},$$

where the invertibility of the embeddings has been used.

In summary, if the number of delays used as input k is big enough, then a deterministic function that predicts the next observation will exist and could be approximated by a certain model.

A.2 Small Rank Adjustment Matrix Inversion Lemma

A.2.1 Lemma

The following matrix inversion lemma has been used to get the results for the BPDC online algorithm.

Lemma 2 (Small Rank Adjustment Matrix Inversion) *Let A be square $N \times N$ matrix. Let U, C and V be matrices of dimensions $N \times K$, $K \times K$ and $K \times N$ respectively. Then*

$$(A + UCV)^{-1} = A^{-1} + A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}.$$

A.2.2 Application to APRL

For compute, in a recursive way, the inverse of $\hat{C}_K = \hat{C}_{K-1} + f_{K-1}f_{K-1}^T$ in terms of $\hat{C}_{K-1}^{-1}$. Using the lemma 2 with $A = \hat{C}_{K-1}$, $A^{-1} = \hat{C}_{K-1}^{-1}$, $U = f_{K-1}$, $C = 1$, $V = f_{K-1}^T$:

$$\begin{aligned}
\hat{C}_K^{-1} &= \hat{C}_{K-1}^{-1} - \frac{\left[\hat{C}_{K-1}^{-1} f_{K-1}\right]\left[f_{K-1}^T \hat{C}_{K-1}^{-1}\right]}{1 + f_{K-1}^T \hat{C}_{K-1}^{-1} f_{K-1}} \\
&= \hat{C}_{K-1}^{-1} - \frac{\left[\hat{C}_{K-1}^{-1} f_{K-1}\right]\left[\hat{C}_{K-1}^{-1} f_{K-1}\right]^T}{1 + f_{K-1}^T \hat{C}_{K-1}^{-1} f_{K-1}}.
\end{aligned}$$

It has been used that $1 + \frac{1}{\epsilon} f_{K-1}^T \hat{C}_{K-1}^{-1} f_{K-1}$ is a 1×1 matrix (that is, a scalar), so its inverse is just $\frac{1}{1 + \frac{1}{\epsilon} f_{K-1}^T \hat{C}_{K-1}^{-1} f_{K-1}}$.

[1]Note that if $k > 2m + 1$ the embedding will be given by Takens Theorem and the canonical embedding $\mathbb{R}^{2m+1} \xrightarrow{i} \mathbb{R}^k$.

A.2.3 Application to BPDC

It has been applied to prove the equality

$$(\epsilon I + f_k f_k^T)^{-1} = \frac{1}{\epsilon} I - \frac{[\frac{1}{\epsilon} I f_k][\frac{1}{\epsilon} I f_k]^T}{1 + f_k^T \frac{1}{\epsilon} I f_k},$$

where I is the identity of dimension $N \times N$, and f_k is a vector $N \times 1$. Substituting in the lemma 2 the expressions $A = \epsilon I$, $A^{-1} = \frac{1}{\epsilon} I$, $U = f_k$, $C = 1$, $V = f_k^T$:

$$\begin{aligned}
(\epsilon I + f_k f_k^T)^{-1} &= A^{-1} + A^{-1} U (C^{-1} + V A^{-1} U)^{-1} V A^{-1} \\
&= \frac{1}{\epsilon} I - \frac{1}{\epsilon} I f_k \left(1 + \frac{1}{\epsilon} f_k^T I f_k \right)^{-1} f_k^T \frac{1}{\epsilon} I \\
&= \frac{1}{\epsilon} I - \frac{1}{\epsilon} I f_k \frac{1}{1 + \frac{1}{\epsilon} f_k^T I f_k} f_k^T \frac{1}{\epsilon} I \\
&= \frac{1}{\epsilon} I - \frac{[\frac{1}{\epsilon} I f_k][\frac{1}{\epsilon} I f_k]^T}{1 + f_k^T \frac{1}{\epsilon} I f_k}.
\end{aligned}$$

Where it has been used that $(1 + \frac{1}{\epsilon} f_k^T I f_k)^{-1} = \frac{1}{1 + \frac{1}{\epsilon} f_k^T I f_k}$.

Appendix B

Notation Glossary

During all the work, a unified notation has been used, so all the algorithms can be easily compared and related.

In this appendix, a summary of this notation is included.

B.1 Abbreviations

APRL Atiya-Parlos Recurrent Learning.

BP BackPropagation.

BPDC BackPropagation Decorrelation.

BPTT BackPropagation Through Time.

EA Evolutionary Algorithm.

EIP Exponential Intrinsic Plasticity.

ES Evolutionary Strategy.

ESN Echo State Network.

ESP Echo States Property.

FFNN Feedforward Neural Network.

GIP Gaussian Intrinsic Plasticity.

IP Intrinsic Plasticity.

LSM Liquid State Machine.

MLP Multilayer Perceptron.

NN Neural Network.

RC Reservoir Computing.

RNN Recurrent Neural Network.

RTRL Real Time Recurrent Learning.

B.2 Notation

a_i^k Activation of the unit i at layer k in a MLP.

A_k Matrix defined in the APRL algorithm for compact notation.

α Spectral radius of the built reservoir weigh matrix.

$B(k)$ Matrix defined in the APRL algorithm for matrix notation.

C_K Approximation of the auto-correlation matrix (except for a constant) at time k.

$\hat{C}_K$ Regularized version of C_k.

$\hat{C}(k)$ Instantaneous version of $\hat{C}_k$.

$d_{\textbf{KL}}$ Kullback Leibler Divergence.

diag[v**]** Matrix with the vector v in its diagonal, it is equivalent to $I \otimes v$ where $\otimes$ stands for the Kronecker product and I is the identity matrix.

δ Vector defined for BPTT reformulation.

δ_{im} Kronkecker's delta, $\delta_{ii} = 1$ and $\delta_{im} = 0, i \neq m$.

$\delta_i(k)$ Derivative of the error w.r.t. the activation of the unit j at time k.

Δa Update of parameter a.

Δb Update of parameter b.

Δt Time constant for the RNN dynamic.

$\Delta \mathbf{x}$ Desired state update (in the opposite direction of the error gradient).

$\Delta w_{ij}^{\textbf{batch}}$ Update of the weight from unit j to unit i in APRL for the batch algorithm.

$\Delta w_{ij}(k)$ Update of the weight from unit j to unit i at time k for an online algorithm.

$\Delta W^{\textbf{batch}}$ Weight update in APRL for the batch algorithm with matrix notation.

$\Delta W(k)$ Weight update at time k for an online algorithm with matrix notation.

$\Delta\mathbf{w}$ Weight update in APRL.

$\Delta\mathbf{w}^{\mathbf{batch}}$ Weight update in APRL for the batch algorithm.

$e_s(k)$ Instantaneous error of unit s at time k.

$e(k)$ Vector of the instantaneous errors of the N units at time k.

E Squared error of the RNN over the training set.

$E[a]$ Expectation of a.

ϵ Regularization constant.

η Learning rate.

$f()$ Activation function of the RNN.

f_k Vector of activations at time k.

$f_a(a)$ Density function of a.

g Constraint of the optimization problem defined in APRL.

$g()$ Activation function of the RNN with IP notation.

$g_i(k)$ Constraint of the optimization problem defined in APRL for unit i at time k.

$g(k)$ Constraint vector of the optimization problem defined in APRL at time k.

$\mathbf{g}$ Vector collecting the constraints of the N units at the K timesteps.

γ Vector defined in the APRL algorithm for compact notation.

$\gamma_i(k)$ Element of γ associated to unit k at time k.

$\gamma(k)$ Subvector of γ associated to time k.

$H(a)$ Entropy of a.

I Index set of the RNN inputs.

k_0 Number of timesteps discarded at the beginning of the training sequence in ESN algorithm.

K Number of timesteps in the training set.

$\lambda_{\mathbf{max}}(A)$ Spectral radius of A, its largest absolute eigenvalue.

μ Desired mean in IP.

N Number of units of the RNN.

o_i^k Value of the unit i at layer k in a MLP.

O Index set of the RNN outputs.

O Matrix of the collected outputs of the units of a RNN, $O = f(X)$.

$p(x)$ Density function of the input x in IP.

$q(y)$ Density function of the output y in IP.

σ Desired deviation in IP.

$\sigma_{max}(A)$ Largest singular value of A.

u_r^k Value of the input r at layer k in a MLP.

$u_r(k)$ Value of the input r at time k.

$u(k)$ Input vector at time k.

w_s Vector of weights that connects output s.

w_{ij} Weight of the connection from unit j to unit i.

w_{ij}^k Weight of the connection from unit j to unit i in layer k in a MLP.

$\mathbf{w}$ Vectorial form of the weight matrix of the RNN.

W Weight matrix of the RNN.

$W^{\textbf{Res}}$ Reservoir weight matrix of the RNN.

x_i^k Value of the unit i at layer k in a MLP.

$x_i(k)$ Value of the unit i at time k.

$x(k)$ State vector at time k.

$\mathbf{x}$ Vector collecting the states of the N units at the K timesteps.

X Matrix of the collected states of the units of a RNN.

$y_i(k)$ Value of the desired output i at time k.

$y(k)$ Desired output vector at time k.

Y_s Vector of the collected desired s-th output.

τ Parameter of the Mackey-Glass system.

$[v]_s$ Component s of a vector.

Z Matrix defined for RTRL reformulation.

Bibliography

[1] Floris Takens. Detecting strange attractors in turbulence. *Dynamical Systems and Turbulence, Warwick 1980*, pages 366–381, 1981.

[2] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Neurocomputing: foundations of research*, pages 457–464, 1988.

[3] D.R. Hush and B.G. Horne. Progress in supervised neural networks. *Signal Processing Magazine, IEEE*, 10(2):8–39, 1993.

[4] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2:303–314, 1989.

[5] Hava T. Siegelmann and Eduardo D. Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4:77–80, 1991.

[6] Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, August 2009.

[7] Kenji Doya. Bifurcations in the learning of recurrent neural networks. *IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS*, pages 2777–2780, 1992.

[8] Paul J. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, October 1990.

[9] Ronald J. Williams and David Zipser. Experimental analysis of the real-time recurrent learning algorithm. *Connection Science*, 1:87–111, 1989.

[10] Amir F. Atiya and Alexander G. Parlos. New results on recurrent network training: Unifying the algorithms and accelerating convergence. *IEEE Trans. Neural Networks*, 11:697–709, 2000.

[11] Jochen J. Steil. Backpropagation-decorrelation: online recurrent learning with o(n) complexity. *IEEE Transactions on neural networks*, 2:843–848, July 2004.

[12] H. Jaeger. Echo state network. *Scholarpedia*, 2(9):2330, 2007.

[13] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[14] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*, pages 318–362, 1986.

[15] H. Jaeger. The ‘‘echo state’’ approach to analysing and training recurrent neural networks. *GMD Report 148*, 2001.

[16] M. Buehner and P. Young. A tighter bound for the echo state property. *Neural Networks, IEEE Transactions on*, 17(3):820 –824, may 2006.

[17] D. Verstraeten, B. Schrauwen, M. D'Haene, and D. Stroobandt. An experimental unification of reservoir computing methods. *Neural Networks*, 20(3):391 – 403, 2007. Echo State Networks and Liquid State Machines.

[18] R. Legenstein and W. Maass. Edge of chaos and prediction of computational performance for neural circuit models. *Echo State Networks and Liquid State Machines*, 20(3):323–334, April 2007.

[19] U. D. Schiller and J. J. Steil. On the weight dynamics of recurrent learning. *in: Proc. ESANN, 2003*, pages 73–78, 2003.

[20] Ulf D. Schiller and Jochen J. Steil. Analyzing the weight dynamics of recurrent learning algorithms. *Neurocomputing*, 63:5 – 23, 2005. New Aspects in Neurocomputing: 11th European Symposium on Artificial Neural Networks.

[21] B Widrow and M E Hoff. Adaptive switching circuits. *Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record, Part 4*, pages 96–104, 1960.

[22] Fei Jiang, Hugues Berry, and Marc Schoenauer. Supervised and evolutionary learning of echo state networks. *Proceedings of the 10th international conference on Parallel Problem Solving from Nature*, pages 215–224, 2008.

[23] H. Jaeger. Tutorial on training recurrent neural networks, covering bppt, rtrl, ekf and the â echo state networkâ approach. Technical report, Fraunhofer Institute AIS, St. Augustin-Germany, 2002.

[24] K. Ishii, T. van der Zant, V. Becanovic, and P.-G. Plöger. Identification of motion with echo state network. 2004.

[25] Keith Bush and Batsukh Tsendjav. Improving the richness of echo state features using next ascent local search. *In Proceedings of the Artificial Neural Networks in Engineering Conference*, 2005.

[26] R. H. Cudmore and N. S. Desai. Intrinsic plasticity. *Scholarpedia*, 3(2):1363, 2008.

[27] Jochen Triesch. A gradient rule for the plasticity of a neuronâs intrinsic excitability. *Proceedings of the International Conference on Artificial Neural Networks, 2005*, pages 65–70, 2005.

[28] Benjamin Schrauwen, Marion Wardermann, David Verstraeten, Jochen J. Steil, and Dirk Stroobandt. Improving reservoirs using intrinsic plasticity, 2007.

[29] L. Glass and M. Mackey. Mackey-glass equation. *Scholarpedia*, 5(3):6908, 2010.

[30] Andrew R. Webb. Statistical pattern recognition. 2002.