Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA
SCUOLA DI SCIENZE
Scienze e Tecnologie Informatiche

Introduction to Reservoir Computing Methods

Relatore:
Chiar.mo Prof.
Andrea Roli

Presentata da:
Luca Melandri

Sessione III
Anno Accademico 2013/2014

”An approximate answer to the right problem
is worth a good deal more than an exact
answer to an approximate problem.”
– John Tukey

Contents

Introduction

Since the creation of the first Computer, the idea of an electronic brain, able of thoughts similar to those of humans, has pervaded minds of a lot of scientist all over the world. This led in 1955 to the introduction of the term **Artificial Intelligence (AI)** defined as "the science and engineering of making intelligent machines" [26]. Within this enormous field of study, a particular typology of application is **Machine Learning**, a data science close-connected to statistics, which studies algorithms with the ability of learning through experiences. The construction of intelligent machines, involve the necessity of perform tasks similar to what humans can do. Since there are few basilar things that we could program a machine to do, and thanks to the numerous datasets we have acquired in years with the growth of the web, machine learning has been re-discovered as a new capability for computers, that today touches many segments of industry and science. Two are the major definitions that we have today of what machine learning is:

- ▷ More informal and, historically the ancestor of any definition ever given, is that of Arthur Samuel in 1959 who defines machine learning as

  Definition 1 *Field of study that gives computers the ability to learn without being explicitly programmed*

- ▷ More formal the definition provided by Tom M. Mitchell in 1997, which states

  Definition 2 *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E*

Both are valid and, while the first asserts what we want, the second, states what should happen in order to obtain it. In this discipline we can distinguish three major categories of problems approached in different manners:

- In **Supervised learning**, the algorithm is fed with a couple ({example input},{desired output}) with the objective to find a map between the input and the output;

- **Unsupervised learning** uses the input given without requiring a correct output, as in Supervised learning, allowing the algorithm itself to find any hidden pattern in data;

- With **Reinforcement learning**, the algorithms learns how to interact with a dynamic environment, in order to optimize results on a certain predetermined goal, without any right choice given from the outside (i.e. learning to play a game by playing against another player or learn to drive a vehicle, interacting with the outer world)

Based on the wanted output, we can find instances of Supervised learning problems like *Regression and classification*, as we will do in chapter 1, tasks as *Clustering*, solved in an Unsupervised way, or a kind of *Robotics and Control* problems solved using Reinforcement learning. After this basic overview on machine learning, we will define incrementally in the next chapters a background of Supervised learning methods, which will end up to cover a comprehensive summary of new methodologies of training for recurrent networks.

The thesis has the following structure:

- **Chapter 1** introduces Regression methods, applied later in the Readout layer of Reservoir Computing;

- **Chapter 2** gives a comprehensive treatment on Neural Network, included an overview on Recurrent Neural Network;

- **Chapter 3** covers the main topic of the thesis: Reservoir Computing methods;

- The **Conclusion** chapter, ends the discussion considering possible improvements for the future of the field of study and providing a few comments on the importance of the methods treated in real life problems.

Chapter 1

Regressions Analysis

1.1 Methodology

Regression analysis is a statistical process used to estimate relationships between two variables. [9] A **Statistical Model** is a simplified representation of a data-generating process. [11] Within a dataset, multiple aspects can be taken in account to pull out interesting predictions. One of the most used models, known in the literature from 1805 [37], consists in the *linear regression*: a statistical linear models used to predict output variables. Lots of researchers try every day to find out relations in data collections, using the discipline of machine learning merged with statistics methods to analyze relationships among variables.

1.2 Linear Regression

Linear predictor functions (LPF) are linear functions which combine independent variables with a set of weights coefficients to predict a dependent variable [5]. A general model for this function is as follows:

$$f(i) = \beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip}, \qquad i = 0, .., n \tag{1.1}$$

where β are variables that limit the influences of each single independent variable.

Linear regression [29, 6] in statistics, is an approach based on the conditional probability of **y** given **X**, used to model relationships between scalar independent variables and one or more, dependent variables. In this approach, data are modeled through a LPF and weights are computed to allow credible predictions for unknown inputs of the same type which the regressor was trained on. **Linear regression** in machine learning is a supervised learning algorithm whose output computes as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + .. + \theta_n x_n. \tag{1.2}$$

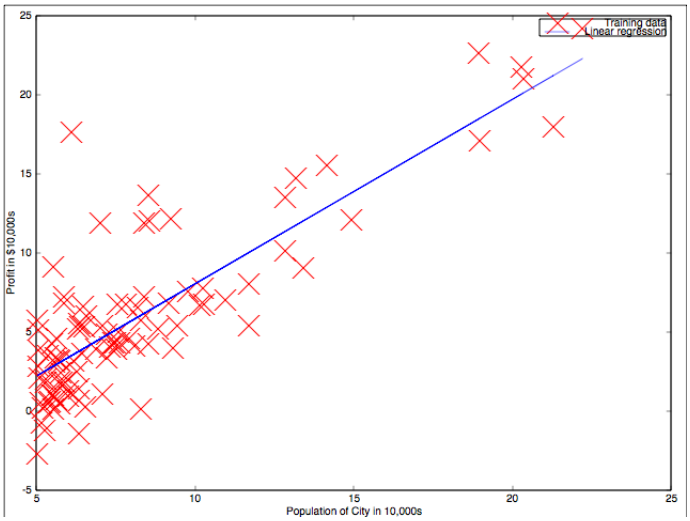

Figure 1.1: An example of Linear Regression over a dataset of houses

Terminology:

- $\mathbf{h}_\theta$: hypothesis formulated by the regression;
- $\mathbf{n}$: number of features;
- $\mathbf{m}$: number of training examples;
- $\vec{\mathbf{x}}$: input vector;
- $\mathbf{y}$: output value;

The purpose of a linear regression model, is to obtain a vector $\vec{\theta}$ containing all weights involved in the regression with values good enough to get satisfactory predictions on unknown future inputs. To obtain a vectorized implementation of the calculus, a $x_0 = 1$ bias term is added and left untouched in any modification to original features. To obtain a good set of θ, we need to solve a minimization problem with respect to $\vec{\theta}$

$$\min_\theta \boxed{J(\vec{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} cost(h_\theta(x), y)} \tag{1.3}$$

where:

$$cost(h_\theta(x), y) = \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \tag{1.4}$$

In particular, we minimize the squared error cost function $J(\vec{\theta})$ between the output of the linear regression $h_\theta(x)$, and the correct output y which is the correct output we would expect if our predictor had done a good prediction.

1.2.1 Gradient Descent and applications

Minimizing this equation means to find a $\vec{\theta}$ that minimize the distance between the output value computed and the real one. To achieve this result we will need to change, iteratively, $\vec{\theta}$ to obtain a value of $J(\vec{\theta})$ smaller at each iteration. This operation can be done using an algorithm, known as **Gradient Descent (GD)** [4, 29] whose looping step states as follows:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\vec{\theta}), \qquad \text{(simultaneous update of all } \theta_j) \tag{1.5}$$

In equation (1.5), a critical point consists in the α factor, known as *Learning Rate*: a constant that scales how much quickly we descend by higher values to lower values. If the scale is appropriate, the algorithm will converge. However, if α is too small the convergence will be reached in a very long time leading to poor performance while, if the rate is too high we risk to pass beyond the minimum and never converge. Use information from partial derivatives to update $\vec{\theta}$, means to search a point where our cost function will be minimal [3]. Due to this behavior, this value has to be hand tuned, based on various attempts and the experience of the user to get a well performing regression layer. To improve convergence, sometimes can be useful to apply **Feature Scaling** to the input values. This operation consists of a uniform rescale to obtain any feature approximately in the range $-1 \leq x_i \leq 1$. Often an operation called *mean normalization* is also applied, to ensure that features (except x_0) have approximately mean zero:

- Compute the average of the features;
- Replace each x_i with

$$\frac{x_i - \mu_i}{\sigma} \tag{1.6}$$

Gradient Descent, is an instance of the family of algorithms based on the use of the gradient, used massively in Machine Learning approaches, and base of state-of-the-art techniques in a lot of tasks. Using this model as seen until now, we can approximate a various set of linear functions. However, not all features are arranged linearly in the space and in that cases our linear model would not fit data in a realistic approximation. It could be useful to have some kind of higher level terms that manipulate our objective function in a smoother curve. This can be obtained through **Polynomial Regression** employing a deformation of more high degree of data employed in the regression in a similar following way:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3, \qquad \Longleftrightarrow \qquad \begin{aligned} x_1 &= (feature), \\ x_2 &= (feature)^2, \\ x_3 &= (feature)^3, \end{aligned} \tag{1.7}$$

1.2.2 Normal Equations

Another methodology to obtain a vector of optimal parameters is known as **Normal Equations** and consists in an analytical solution of an optimization problem with respect to θ. This approach is often valuable when the number of features is small, otherwise an iterative approach could be preferable due to a lower computational complexity. In this manner, $\vec{\theta}$ is obtained using the following expression:

$$\vec{\theta} = \left(X^T X\right)^{-1} X^T y \tag{1.8}$$

where

- X is a matrix containing each of the m examples;
- X^T is the transpose of X;
- $\left(X^T X\right)^{-1}$ is the Moore-Penrose pseudo inverse matrix of the multiplication between X transpose and X;
- y is the correct output vector;

1.3 Logistic Regression

In problems of the real world, it often occurs the necessity to classify data over a dataset. This problem is considerably different from the linear regression one, composing another field in the Analysis of regressions, called **Logistic Regression** [29, 7]. Given a set of data, it studies the probability of a data to belong or not, to a specific class. Compared to *Linear Regression*, this algorithm solves the task providing an hypothesis $h_\theta(x)$ in the range $[0, 1]$ computing the probability $p(y = 1|x; \vec{\theta})$ that a data belongs to a positive class ($y = 1$) or a negative one ($y = 0$).
This is done computing the hypothesis using a logistic function, in the range $[0, 1]$:

$$h_\theta(x) = g(z), \quad \Longrightarrow \quad g(z) = \frac{1}{1 + e^{-z}} \quad \Longrightarrow \quad z = \vec{\theta}^T x; \tag{1.9}$$

Using variations over data comparable with the equation (1.7)

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2), \quad \vec{\theta} = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{1.10}$$

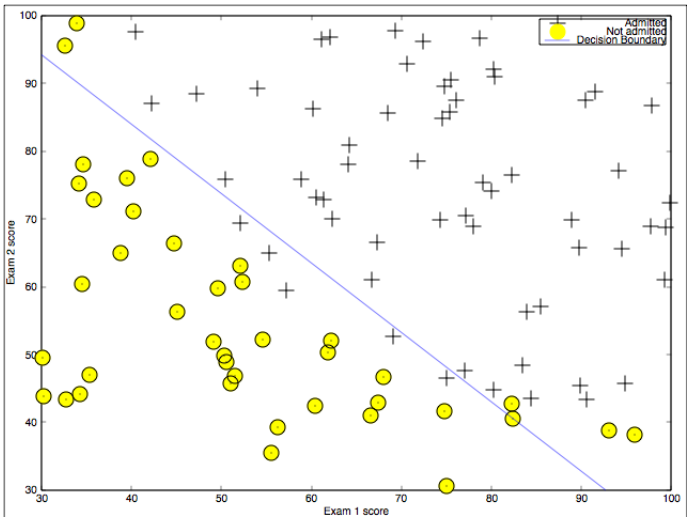

Figure 1.2: An example of Logistic Regression over a dataset of exam's scores

more complex decision boundaries can be obtained and applied to the classification problem. Compute a set of weights for a classification task, consists in the same problem of the (1.3), however the objective cost function changes in order to obtain a *convex function* that allows *GD* to converge to the global minimum.
It is defined as follows:

$$cost(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & y = 1, \\ -\log(1 - h_\theta(x)) & y = 0 \end{cases} \tag{1.11}$$

This function embodies the properties researched in a cost function:

- **convex** behavior to achieve the global minimum;
- **cost**$\rightarrow$ inf if $h_\theta(x) \neq y$, to minimize $J(\vec{\theta})$ towards minimum;

To optimize $J(\vec{\theta})$, the previously used *GD* in the equation (1.5) can be an option since, although a very different hypothesis is given in the (1.9), directives like the equation (1.6) or α choice criteria are still valid. Gradient descent is a straightforward and popular algorithm to decrease objective function's value that has in simplicity one of its major advantage. Other optimization algorithms can be used to obtain a point of minimum:

a) Conjugate Gradient;

b) BFGS;

c) L-BFGS;

These methods are more complex than GD, but have interesting advantages over it because they do not need to choose a learning rate to appropriately converge in short time and often they are faster than gradient descent with which they share values, $J(\vec{\theta})$ and $\frac{\partial}{\partial\theta_j}J(\vec{\theta})$, required to converge.

1.3.1 One-vs-All

In real problems however, it often happens that more than one class can fit the data so a different paradigm of logistic regression needs to be applied, called one-vs-all classification, which consist in

$$h_\theta^i(x) = P(y = i|x;\theta), \quad i = 1,..,N \tag{1.12}$$

that is the training of a logistic regression classifier for each class i to predict the probability that $y = i$. The prediction is then done, choosing the class i that maximizes the hypothesis

$$\max_i \boxed{h_\theta^{(i)}(x)} \tag{1.13}$$

1.4 Fitting the data

1.4.1 Regularization

If we have too many features, the learned hypothesis can adapt very well to the training set $\left(J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \approx 0\right)$ but not being able to generalize new examples. On the other hand, use a set of features too small can lead to the opposite problem, where the induced hypothesis has a strong preconception on the output, approximating a function that does not fit the dataset. The first problem is called **Overfitting** and in this case, $h_\theta(x)$ is said to have *High Variance*, overestimating the data; The second one is known as **Underfitting** where the hypothesis is said to have *High Bias*, underestimating the data. To produce an hypothesis that generalizes well, options available are

- Reduce number of features, manually or by carefully choosing the model;
- Regularization, keeping all features that contributes to the prediction y by scaling down $\vec{\theta}$ values;

Use smaller values for parameters θ allows to obtain a simpler hypothesis less prone to overfitting. This result can be obtained adding a regularization parameter

- $\lambda\sum_{j=1}^{n}\theta_j^2$ for linear regression,

- $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ for logistic regression,

to the cost function that mitigates the influence of all theta, except θ_0, over the cost. The parameter λ, assumes a fundamental role in the regularization function.
Given $\widehat{dom}$ = (order of magnitude of the data of the problem),

- $\diamond$ $\lambda \gg \widehat{dom}$:
  in this case all thetas are strongly penalized, leading to an hypothesis close to 0 with an high bias;
- $\diamond$ $\lambda \ll \widehat{dom}$:
  where $\lambda \approx 0$, the regularization term does not influence the hypothesis, leading to overfitting as if it were not present;

However, as the learning rate, λ has to be chosen through experience and attempts on data.

1.4.2 Model Selection

To obtain an optimized $\vec{\theta}$, a straightforward strategy is try to minimize the error as illustrated above. However, a low error does not necessarily mean a good parameter set, indeed this could also be index of *Overfitting*. To recognize this issue, a possibility is to plot a graph of data although this is not ever possible, usually due to an high number of features. A generally applicable numerical way, adoptable for linear regression as well as for logistic regression, is the **Train and Test Scheme** to test the goodness of the model, that consists in:

- Split the dataset in two distinct pieces, one for training and another for testing, usually in a 70:30 ratio;
- Optimize the cost function using the *Train Set*;
- Use (1.3) to compute the error on the *Test Set* using learned parameter;

A variation for logistic regression that sometimes best fits the analysis, is the misclassification test:

$$errorTest = \frac{1}{m_{test}}\sum_{i=1}^{m_{test}} err(h_\theta(x_{test}^{(i)}), y_{test}^{(i)}) \tag{1.14}$$

where

$$err(h_\theta(x), y) = \begin{cases} 1 & \textbf{if } \ (h_\theta(x) \geq 0.5 \wedge y = 0) \textbf{ or } (h_\theta(x) \leq 0.5 \wedge y = 1) \\ 0 & \textbf{otherwise} \end{cases} \tag{1.15}$$

Fundamental to avoid overfitting is to apply a good-fitting model to our dataset, using at various grade of deepness a polynomial regression that includes more rich features. Adding features however can lead to overfit data. *Train and test* can be further enhanced, to allow evaluation of the best model to use for our problem, splitting the dataset in a 60:20:20 ratio, respectively sets of *Training-Cross-validation-Test* sets, to apply best control on the model selection, following this algorithm:

Step 1: Minimize the cost function for each model using the *Training set*;

Step 2: Test each hypothesis on the *Cross-validation set* to compute the cross-validation error and pick $\vec{\theta}$ which result to have the lowest error;

Step 3: Estimate generalization error of the model using the *Test set*;

In case of underfitting, we find both cross-validation and training error are high, while in overfitting the cross-validation error is high but the training error is low. Thus, this *Train, cross and test* methodology allows to choose the model whose grade best fit to the data, resolving from a point of view the overfitting that can occur.

Chapter 2

Neural Networks

2.1 Methodology

In the previous chapter, we talked about *Regressions* as a method to predict or classify. These techniques work well on data with a relatively simple behavior (e.g. a linear model, a quadratic model, ..) while, for complex patterns, to obtain precise results, the computational complexity needed to compute the parameters leads to a hard application of Regression methods. An instance of problems that need a different approach is computer vision: this branch of computer science which analyzes images involves a high number of features and complex hypothesis to recognize and catalog objects. These reasons have led to search an alternative way to solve such problems. Since 1980, neuro-scientists do experiments of brain rewiring, currently still conducted [28, 18], to study the brain response to the alteration of stimuli. Evidences showed a sort of plasticity that allows the brain to readjust in some way to respond to input changes. Hand in hand, it is formalized an idea inspired by these researches on the brain, that suggests a single learning algorithm used by the brain to learn everything, observing positive examples and consequently learning to reproduce them. This is the basilar idea that took to the creation of *artificial Neural Networks*, one of the most powerful learning algorithms known today and state-of-the-art technique in various fields.

2.2 Artificial Neural Networks

2.2.1 Biological counterpart

As mentioned before, the aim to create a general algorithm that can learn everything, led to the development of **Artificial Neural Networks (aNN)**: a family of statistical algorithms inspired by their biological counterpart observed studying the brain. In particular, looking at the composition of a neuron in the brain [29] we can see that a neuron

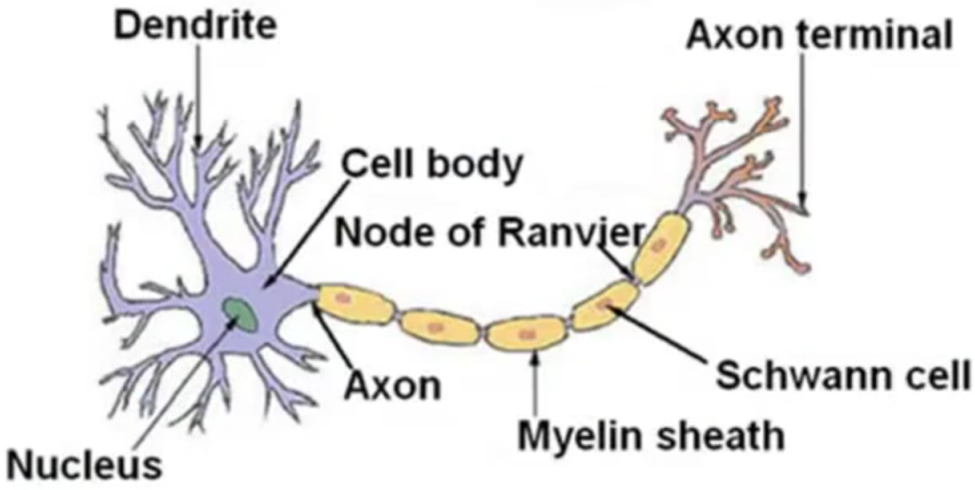

Figure 2.1: Schematic model of a biological neuron found in the brain, with highlight on most important parts

is composed of a body that contain a nucleus, lots of dendrites that act as "input wires" connected to the body, while from the body comes out the axon, that we can consider as an "output wire" with lots of terminations of connection to other neurons. At high level, an interaction between a neuron "A" and a neuron "B" is:

- ◇ The neuron A receives some input to its dendrites, elaborates the signal and send a change of polarity, known as "spike", across the membrane of the neuron that ensures the propagation of the signal through the axon;
- ◇ The neuron "B" receives the input from its dendrites which are connected to the axon of Neuron "A";

2.2.2 Multilayer perceptrons

A simplified model of a neuron is represented by a computational unit that receive inputs, does some computation and then outputs the result to other neurons. [29] An artificial Neural Network, is a set of artificial neurons that work together to achieve an higher computational power. Terminology:

- $a_i^{(j)}$: activation of unit i in layer j;

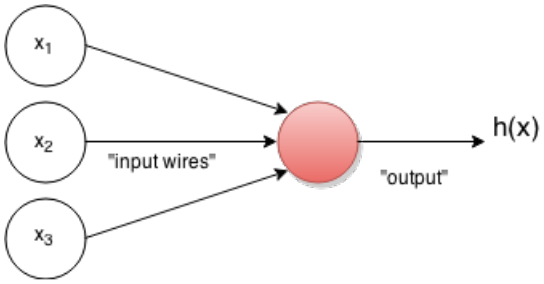

Figure 2.2: A representation of a neuron's model that uses a logistic activation function $g(z) = \frac{1}{1+e^{-(\theta^T X)}}$. It is also known as **Perceptron** and it is the simplest representation of a neuron.

- $\theta^{(j)}$: matrix of weights that control function mapping, from layer j to layer $j+1$. If the networks has u_j units in layer j and u_{j+1} units in layer $j+1$, then $\theta^{(j)} \in \mathbb{R}^{u_{j+1} \times (u_j+1)}$
- m: training examples, as couples $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), ..., (x^{(m)}, y^{(m)})\}$;
- L: total number of layers in the network;
- K: number of classes in a multi-class classification problem;
- u_l: number of units in layer l, without bias unit

By inputing data in the network coupled with the correct result respect which the error is calculated, we bring the network to adapt its weights to approximate better and better the data given as input, and most likely to issue a correct value when unknown values will be provided as input to the algorithm.

2.2.3 Train a Neural Network

Using the Figure 2.3 as instance, network's activations compute as follows:

$$z_1^{(2)} = \theta_{10}^{(1)} x_0 + \theta_{11}^{(1)} x_1 + \theta_{12}^{(1)} x_2 + \theta_{13}^{(1)} x_3 \tag{2.1}$$
$$a_1^{(2)} = g(z_1^{(2)}) \tag{2.2}$$
$$z_2^{(2)} = \theta_{20}^{(1)} x_0 + \theta_{21}^{(1)} x_1 + \theta_{22}^{(1)} x_2 + \theta_{23}^{(1)} x_3 \tag{2.3}$$
$$a_2^{(2)} = g(z_2^{(2)}) \tag{2.4}$$
$$z_3^{(2)} = \theta_{30}^{(1)} x_0 + \theta_{31}^{(1)} x_1 + \theta_{32}^{(1)} x_2 + \theta_{33}^{(1)} x_3 \tag{2.5}$$
$$a_3^{(2)} = g(z_3^{(2)}) \tag{2.6}$$
$$h_\theta(x) = a_1^{(3)} = g(\theta_{10}^{(2)} a_0^{(2)} + \theta_{11}^{(2)} a_1^{(2)} + \theta_{12}^{(2)} a_2^{(2)} + \theta_{13}^{(2)} a_3^{(2)}) \tag{2.7}$$

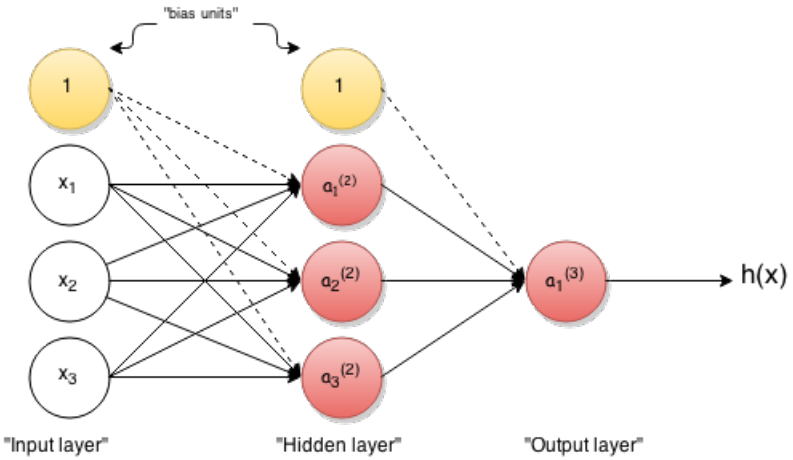

Figure 2.3: A structure of a generic artificial Neural Network, known also as **Multilayer Perceptron**, composed by 3 input units (plus a bias term), 3 hidden units (plus a bias term) and 1 output unit

where $a_i^{(j)} = g(z_i^{(j)})$ is the activation of the unit i in the layer j and uses a sigmoidal function to compute the activation of its internal states. Such sigmoid function [10] is a mathematical function having "S" shape, referred to the logistic activation $\frac{1}{1+e^{-z}}$ or other functions like arctangent or hyperbolic tangent which are usually involved in each unit's state update.

The calculus of activation in this sequentially chain (2.1) is known as **Forward Propagation algorithm** and it is the first step in the training of a neural network, also called **Feedforward Neural Network** in reference to this procedure. The application of a neural network to a classification problem, results in a similar problem to the one-vs-all methodology treated previously in logistic regression with the difference that in this case, the network independently calculates its parameters for the classification. To achieve a multi-class classification we need to have an output unit for each class we want to recognize. In order to find the best approximation we have to minimize the cost function $J(\theta)$, as done before for other learning algorithms, defined here for neural networks as follows:

$$h_\theta(x) \in \mathbb{R}^K \quad \Longrightarrow \quad (h_\theta(x))_i = i^{th}\text{output} \tag{2.8}$$

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}\left(\theta_{ji}^{(l)}\right)^2 \quad (2.9)$$

where the logistic regression hypothesis for the current data is summed over all K classes and it is regularized by a parameter that take in account all weights involved in the computation. Besides the cost, we must also calculate the partial derivatives with respect to each $\theta_{ij}^{(l)}$. In neural networks, this is done using the **Backpropagation algorithm**, the second fundamental step to train a neural network.

Consider $\delta_j^{(l)}$ the "error" committed by unit j in layer l, formally $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(i)$, $(for j \geq 0)$ where $cost(i) = y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log h_\theta(x^{(i)})$. Below we provide the backpropagation algorithm used to compute these values. Notable is that the

Algorithm 1: artificial Neural Networks Backpropagation

Data: a training set of m examples
Result: partial derivatives $\frac{\partial}{\partial \theta_{ij}^{(l)}} J(\theta) = D_{ij}^{(l)}$

Initialize the variables $\Delta_{ij}^{(l)} = 0$ (for all i, j, l);
for $i = 1$ **to** m **do**
  Set $a^{(1)} = x^{(i)}$;
  Perform forward propagation to compute $a^(l)$ for $l = 2, 3, .., L$;
  Using $y^{(i)}$, compute $\delta^{(L)} = a^{(L)} - y^{(i)}$;
  Propagate the error through the network computing $\delta^{(L-1)}$, $\delta^{(L-2)}$, . . ,$\delta^{(2)}$;
  Accumulate error for each unit through $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$;
end

$$D_{ij}^{(l)} = \begin{cases} \frac{1}{m}\Delta_{ij}^{(l)}, & \text{if } j = 0 \\ \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\theta_{ij}^{(l)}, & \text{if } j \neq 0 \end{cases}$$

first layer never gets involved in the computation of the error because is composed by inputs and therefore considered right. Also, the error terms as defined can be directly computed only in the last state with respect to the correct output. Then, in order to obtain other values we must backpropagate our error's term, $\delta^{(L)}$, to the previous nodes that competed to form the current output value, to obtain the corresponding weight that the term have had in the decision. The propagation is done in analogous terms to the Forward ones, even if here we compute the errors from the last to the first hidden layer included, through the following equation

$$\delta^{L-h} = \left(\theta^{(L-h)}\right)^T \delta^{(L)} \circ \boxed{g'(z^{L-h})} \Longrightarrow \quad g'(z^{L-h}) = a^{L-h} \circ (1 - a^{L-h}) \quad (2.10)$$

Even if this algorithm is conceptually simple, its application usually is a bit insidious since the cost could decrease also with an incorrect calculus of the gradient and give non optimal results. A solution to this problem can be a procedure called **Gradient Checking** that consists in the calculus of the derivative using the definition, approximating the slope of the function by the ratio between the cost function difference on a minimal variation of a specific θ and the variation doubled

$$\frac{\partial}{\partial\theta_j}J(\theta) \approx \frac{J(\theta_1,\theta_2,..,\theta_j+\varepsilon,..,\theta_n) - J(\theta_1,\theta_2,..,\theta_j-\varepsilon,..,\theta_n)}{2\varepsilon} \tag{2.11}$$

The value obtained should not differ more than few decimal places from the backpropagation ones to consider right the algorithm implementation. To train a neural network the initial θ is chosen random in an interval $[-\epsilon,\epsilon]$ to obtain a symmetry breaking in the update of the weights; Otherwise, when the update occurs, the weights would advance coupled without generating a really interesting function. Below, to correctly train a Neural Network, the sequence of steps to take is:

- ▷ Random **initialization** of all θ;
- ▷ Use of **Forward Propagation** to compute $h_\theta(x^{(i)})$ for each $x^{(i)}$;
- ▷ Computation of the **cost function** $J(\theta)$;
- ▷ Use of **Backpropagation** to compute partial derivatives $\frac{\partial}{\partial\theta_{ij}^{(l)}}J(\theta)$;
- ▷ [**Optional**] Use of **gradient checking** to compare backprop values to numerical estimate of gradient of $J(\theta)$;
- ▷ Use gradient descent or other **optimization** algorithms to minimize $J(\theta)$ as a function of parameters θ;

Since the neural network's cost function is non-convex, gradient descent could being stuck in a local minimum however this is not commonly a problem obtaining very good approximations on various problems.

2.3 Artificial Recurrent Neural Networks

Based on the already known aNNs, a more biological design inspired by brain modules has been developed with the introduction of **Artificial Recurrent Neural Network (aRNN)** that are distinguished from the widely used feedforward networks by the presence of cycles in their connection topology. Artificial Neural Networks are comparable to functions, able to represent data in the domain of space, and to map input's features to the domain of output. The structure of a Recurrent network is characterized by cycles

between units; this allows the development of self-sustained temporal activations along network's connection pathways, even in absence of inputs. The influence of inputs in the network is maintained through cycles among nodes, allowing to model a dynamic system in function of time. This effect is known as *dynamic memory*. It has been mathematically demonstrated [19] that recurrent networks have the *universal approximation property* and thus, able to model dynamic systems with arbitrary precision; in addition, studies show that with a sufficient number of neurons, an aRNN can be computationally Turing-Equivalent [1]. aRNNs can be seen from two major perspectives: for an emulation purpose of biological models of brain processes in neuroscience or as a tool, a sort of black-box to model engineering problems and signal processing. In machine learning, the second instance of tasks mostly applies, so it will be the most focused in the next analysis, but important influences between approaches will be seen in the next chapter.

Formally, a Recurrent Neural Network is defined in a similar way to a feedforward network: a set of neurons, also called units, connected each other by synaptic links whose strengths are defined by a set of weights. *Input units* defined as $\mathbf{u(n)}$ are introduced into the network which generates its *internal units activation* $\mathbf{x(n)}$ and output some value $\mathbf{y(n)}$. Terms that will be used in the upcoming definitions, follow a network's

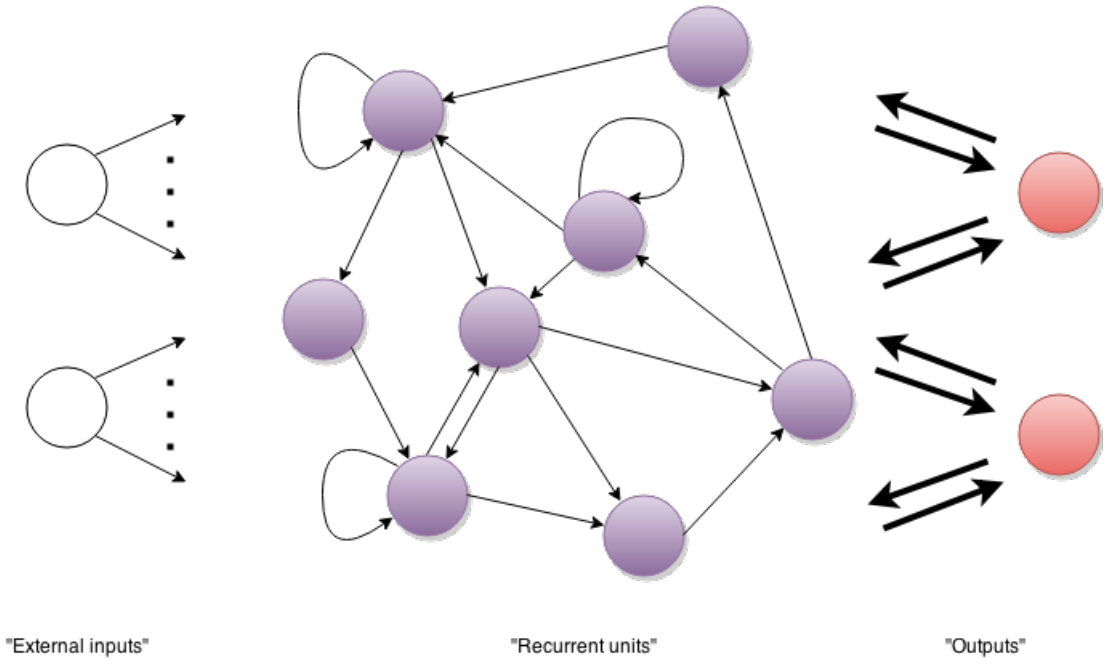

Figure 2.4: A structure of a generic artificial Recurrent Neural Network

architecture with **K** input units

$$\mathbf{u(n)} = (u_1(n), ..., u_K(n))^T; \tag{2.12}$$

N internal units

$$\mathbf{x(n)} = (x_1(n), ..., x_N(n))^T; \tag{2.13}$$

and **L** output units

$$\mathbf{y(n)} = (y_1(n), ..., y_L(n))^T; \tag{2.14}$$

These series of values are connected each other with a set of matrices, where 0 means no connection [19]:

- $\mathbf{W}^{in} = (w_{ij}^{in}), \quad W^{in} \in \mathbb{R}^{N \times K}$ for connections between inputs and internal units;
- $\mathbf{W} = (w_{ij}), \quad W \in \mathbb{R}^{N \times N}$ for connections within the network.,
- $\mathbf{W}^{out} = (w_{ij}^{out}), \quad W^{out} \in \mathbb{R}^{N \times N}$ for connections between inputs and internal units,
- $\mathbf{W}^{back} = (w_{ij}^{back}), \quad W^{back} \in \mathbb{R}^{N \times L}$ for optional backprojection from output, to internal units of the network;

To give an update equation for internal states of the aRNN, we have to consider the external input, $u(n+1)$, the current internal state, $x(n)$, and eventually the backprojection into the system. In addition, the choice of the internal unit's activation needs a bit of attention since it determines an important piece of network dynamics. For considerations done above on the sigmoidal function, and for the large role it covers in the literature of neural networks in general, this will be the choice almost anywhere, in particular adopting the hyperbolic tangent definition, tanh. Thus, the activation of internal units is computed using the following formula [19]:

$$x(n+1) = f(W^{in}u(n+1) + Wx(n) + W^{back}y(n)), \tag{2.15}$$

Then define $\vec{o} = (u(n+1), x(n+1))$ as the vector composed jointly by input and internal activations and f^{out} the activation of the output units that will be mostly sigmoidal functions as considered over; The output is released in the following manner:

$$y(n+1) = f^{out}(W^{out}\vec{o}). \tag{2.16}$$

As already reported, Recurrent networks are very powerful tools to model complex systems; However, since their first theorical appearance [8] in 1980, were not exploited due to the high computational requirements required by the known optimization methods involved in the training (before "RC" methods).

2.3.1 Classical methods

Historical approaches to the train of Recurrent Networks involve the change of weight matrices in a similar way that occurs with feedforward networks using training algorithms more or less derived from what used in standard networks, adapted for processing data through time. It is the case of **Backpropagation Through Time (BPTT)**, the most commonly used algorithm in aNN's training, adapted to aRNN, "unfolding" the recurrent network in time and generating multiple copies connected each other, as a materialized time stream of the input sequence $\{..., u(n-1), u(n), u(n+1), ...\}$, using in each copy the same weights and minimizing the error *through time* between computed output and what is given as correct result, called also teacher-output, like normally occurs in backprop for aNN. This method has a computational complexity of backprop applied over T, time inputs in which the network is splitted, obtaining an $O(TN^2)$ load. This requirement of an high computational power, the long time required to converge to an acceptable solution and the **Vanishing Gradient** [12, 19] problem who does not allow the capture of the effects of previous inputs for a time longer than a dozen of time steps, make this algorithm a poor choice for the training of a recurrent network. Other methods quotable which obtain usually better performance over BPTT while continuing to use the same approach to the network are [19]

- **Real-time Recurrent learning (RTRL)**
  A method that compute the derivatives of states and outputs with respect to all weights as the network processes the sequence, during each time step of the forward phase;

- **Extended Kalman-filtering (EKF)**
  A state estimation technique for non linear systems derived by linearizing the Kalman filter around the current state estimate.

- **AtiyaParlos learning (APRL)**
  An $O(n^2)$ complexity [33] method that leads to use directions not pointed by the gradient to try to minimize the error;

All these methods, exception done for APRL which will be also the base for subsequent talk, suffer of gradient vanishing and this mean that through time the effects of the gradient tend to fade with obvious negative results.

2.3.2 RNN Architectures

From the first attempt of aRNN development, various network architectures were proposed. Here we give a massive overview on the most importants:

- ▷ **Fully recurrent network**: It is the basic architecture developed in 1980s and is composed of neuron units each one is connected to all others. Each connection has a modifiable real-valued weight. Some of these units are called input nodes while others are output nodes. What is not input nor output is considered an hidden node.

- ▷ **Hopfield network**: Not designed to recognize sequence of patterns, serves as content-addressable memory system with binary threshold nodes. It is composed entirely by symmetric connections which are trained using the Hebbian learning rule and has an assured convergence to a local minimum;

- ▷ **Jordan network**: Developed in 1986 [21], it is composed of three layers interconnected; an additional "context layer", linearly connected to the output layer, holds the previous output and propagates it as input to the middle hidden layer. Its major use was prediction thanks to the "context layer" that granted a short-term memory , allowing predictions of sequences.

- ▷ **Elman network**: Similar in the structure to the Jordan described above, it linearly store in its "context layer" [14] the entire previous activation of the hidden layer at each propagation, allowing as above, tasks unavailable to standard aNNs. The architectures of Elman and Jordan are also known as **Simple Recurrent Networks**;

- ▷ **Long short-term memory network (LSTM)**: A special class of recurrent networks that does not suffer of the *Vanishing Gradient issue*, hence reaching optimal results with a training based on gradient's informations. The particular characteristic of the LSTM architecture is the memory cell, a linear unit which holds the state of the cell surrounded by three gates:

  - ⊗ G_I: the modify of the neuron internal state is allowed only when the input gate is open;
  - ⊗ G_O: controls when data flow to other parts of the network, that is, how much and when the cell fires;
  - ⊗ G_F: the forget gate, determines how much the state is attenuated at each time step.

  Terminology:

  - g^{in}: activation of input gate;
  - g^{out}: activation of output gate;
  - g^{forget}: activation of forget gate;

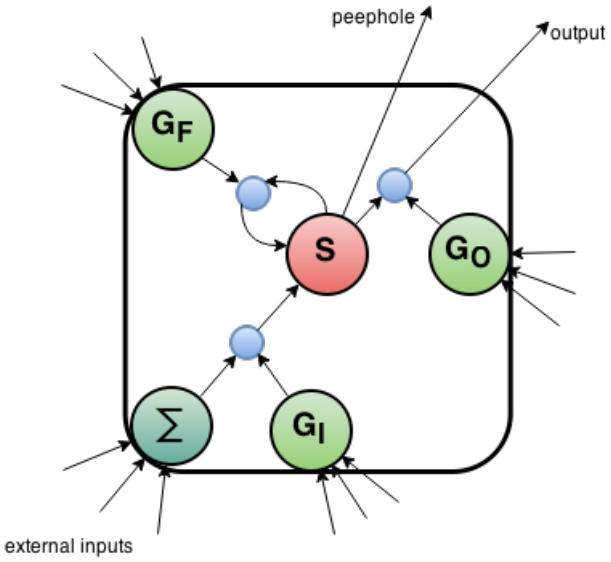

Figure 2.5: Long short-term Memory cell composition.

- net: weighted sum of external inputs ($\sum$);
- h: identity function;
- c_j: output of cell j;
- σ: sigmoid function;
- g_i^{type}: amount for gate of $type \in \{in, out, forget\}$ that determines if it is open or not;

The activation state of cell i is given by

$$s_i(t) = net_i(t) g_i^{in}(t) + g_i^{forget}(t) s_i(t-1), \tag{2.17}$$

$$net_i(t) = h\left(\sum_j w_{ij}^{cell} c_j(t-1) + \sum_k w_{ik}^{cell} u_k(t)\right) \tag{2.18}$$

$$c_j(t) = \tanh(g_j^{out}(t) s_j(t)), \tag{2.19}$$

$$g_i^{type} = \sigma\left(\sum_j w_{ij}^{type} c_j(t-1) + \sum_k w_{ik}^{type} u_k(t)\right) \tag{2.20}$$

The definition of *dynamic engineering systems* as treated until now, apply to various application fields, for instance: filtering of informations, predictions, data compression,

pattern classification, . . . and some interesting applications currently in strong expansion, have been applied in telecommunication, video data analysis, robotics, biomedical diagnostics and man-machine interfaces. These are only some of the possible uses of this family of algorithms whose exploit will be treated in the next chapter.

Chapter 3

Reservoir Computing

3.1 Methodology

We now want to investigate a relatively new approach in aRNN training called Reservoir Computing. This technique has been developed in three different methods which we are covering in the next sections called "Liquid State Machine", "Echo State Network" and "Backpropagation-Decorrelation learning rule". These methods aim to promote a new approach of modeling complex dynamic systems in mathematical and engineering fields via an artificial Recurrent Neural Network. Each approach covered consists of a fixed-weight recurrent network that, fed by a dataset, outputs a series of activation's states. These intermediate values are then used to train output connections to the second part of the system which will output a description of original model's dynamics obtained from datas. The first part of the system, called Reservoir, is an aRNN with fixed weights that acts as "black-box" model of a complex system; The second one is known as Readout, a classifier layer of some kind, usually a simple linear one, connected by a set of weights to the Reservoir. A fundamental property belonging to all these techniques is to have a sort of intrinsic *memory effect*, due to recurrent connections in the reservoir than whose size, represented by the time steps needed to exhaust the effect of the th-input in reservoir's computed output. During reservoir's construction, one of the major behavior to take in account is the activation function in use to characterize nodes' behavior. In literature we see use examples of various models of artificial neurons from simple linear models to more elaborated non-linear ones, like the sigmoidal often used in the "Echo state" and Backpropagation-decorrelation's approach, or biological-inspired LIF model mainly employed in "Liquid State" technique we will see later.

3.2 Echo State Network

In this first section, we analyze the **Echo State Network (ESN)** approach introduced in Jaeger 2001 [20]. The term "echo" mean that the activation state **x(n)** of an arbitrary assembled aRNN is a function of the input history **u(n)**, **u(n-1)**, ... presented to the network. Networks used in this case are usually discrete-time composed of sigmoidal units, we will refer to as **Dynamic Reservoir (DR)**. A generic ESN model [20] is composed of a discrete-time neural network with K input units, N internal network units and L output units. Activations of network's units at time step n are described by $\mathbf{u}(n) = (u_1(n), ..., u_K(n))$, $\mathbf{x}(n) = (x_1(n), ..., x_N(n))$ and $\mathbf{y}(n) = (y_1(n), ..., y_L(n))$ for inputs, internal and output units respectively. Consider four weight matrices real-valued:

1. **Input nodes:**
   An N x K weight matrix $\mathbf{W}^{in} = (w_{ij}^{in})$ collects connections between inputs and internal units

2. **Internal nodes:**
   An N x N weight matrix $\mathbf{W} = (w_{ij})$ collects internal units weights and recurrent pathways between each other

3. **Output nodes:**
   Connection's weight from system to output units, an L x (K + N + L) matrix is prepared $\mathbf{W}^{out} = (w_{ij}^{out})$

4. **Backprojection nodes:**
   An N x L weight matrix $\mathbf{W}^{back} = (w_{ij}^{back})$ is stored for the connections that project back from output to internal units.

Internal units' activation through time is updated according to the following activation scheme where $f = (f_1, ..., f_N)$ are output functions of internal units:

$$x(n+1) = f(W^{in}u(n+1) + Wx(n) + W^{back}y(n)) \tag{3.1}$$

The output is computed through the following activation scheme where $f^{out} = (f_1^{out}, ..., f_L^{out})$ are output functions of the output units and $(u(n+1), x(n+1), y(n))$ is the concatenation of input, internal and previous output activation vectors:

$$y(n+1) = f^{out}(W^{out}(u(n+1), x(n+1), y(n))) \tag{3.2}$$

Given a generic model for ESNs, we want to guess characteristics that a network must show to have Echo State, a property that belongs to the weight matrix **W** and is influenced by external inputs used during the training. With regards to this last statement, is required that training input vectors $\mathbf{u}(n)$ belong to a compact interval U and training output vectors $\mathbf{y}(n)$ belong to a compact interval Y [19].

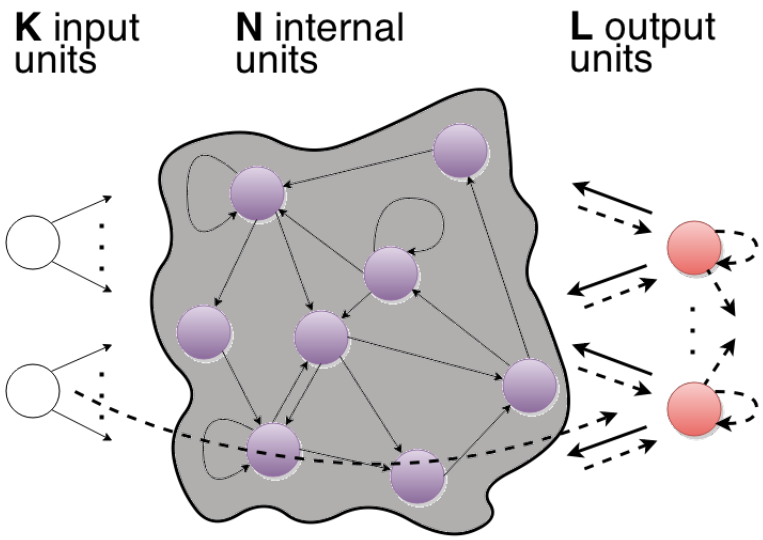

Figure 3.1: General structure of an ESN, where dashed arrows represents possible optional connections.

Definition 3 *(echo states) Assume an untrained network with weights* $\boldsymbol{W}^{in}$, $\boldsymbol{W}$ *and* $\boldsymbol{W}^{back}$ *is driven by teacher input* $\boldsymbol{u}(n)$ *and teacher-forced by teacher output* $\boldsymbol{y}(n)$ *from compact intervals U and Y, if for every left-infinite input/output sequence* $(\boldsymbol{u}(n), \boldsymbol{y}(n-1))$, *where* $n = ..., -2, -1, 0$, *and for all state sequences* $\boldsymbol{x}(n), \boldsymbol{x'}(n)$ *compatible with the teacher sequence, e.g. with*

$$\begin{aligned} x(n+1) &= f(W^{in}u(n+1) + Wx(n) + W^{back}y(n)) \\ x^{'}(n+1) &= f(W^{in}u(n+1) + Wx^{'}(n) + W^{back}y(n)) \end{aligned} \tag{3.3}$$

it holds that $\boldsymbol{x}(n) = \boldsymbol{x}^{'}(n)$ *for all* $n \leq 0$.

The definition, states that as long as we train a network, its state in a finite time T is determined by the history of the input and the teacher-forced output so, for every internal signal $x_i(n)$ exists an echo function e_i which maps input/output histories to the current state:

$$\begin{aligned} e_i : (UxD)^{-1} &\rightarrow \nabla \\ (..., (u(-1), y(-2)), (u(0), y(-1))) &\rightarrow x_i(0) \end{aligned} \tag{3.4}$$

From Jaeger 2002 [19] we know there is a connection between algebraic properties of the internal weight matrix $\mathbf{W}$ and the echo state property (**ES property**) even if Jaeger himself in its work states that no known algebraic conditions allows, given $(\mathbf{W}^{in}, \mathbf{W}, \mathbf{W}^{back})$, to certainly assert the network own the echo state property. However he formulates a sufficient condition [20] for the *non-existence of echo state*

Proposition 1 *Assume an untrained network* ($\boldsymbol{W}^{in}$, $\boldsymbol{W}$, $\boldsymbol{W}^{out}$) *with state update according to* (3.1) *and with transfer functions* tanh. *Let* $\boldsymbol{W}$ *have a spectral radius* $|\lambda_{max}| > 1$, *where* $|\lambda_{max}|$ *is the largest absolute value of an eigenvector of* $\boldsymbol{W}$. *Then the network has no echo states with respect to any input/output interval U x D containing the zero input/output (0,0).*

This proposition gives a condition which does not allows the existence of echo state property in the weight matrix $\mathbf{W}$ with spectral radius major than one. Tests [19] showed that usually when the spectral radius is below one, $\mathbf{W}$ has the ES property. Other matrices part of ESN definition such $\mathbf{W}^{in}$ and $\mathbf{W}^{back}$ can be freely chosen because are not involved in echo state property definition. Following these ideas, in Jaeger (2002) [19] is given an empiric algorithm used to train a complete Echo State Network that should be able to approximate data generated by the same system the network was trained on.

3.2.1 Algorithm

Step 1: Generate an untrained DR ($\mathbf{W}^{in}, \mathbf{W}, \mathbf{W}^{back}$) which has echo state property and choose arbitrarily $\mathbf{W}_{in}$ and $\mathbf{W}_{back}$. Attention and experimental attempts must be addressed to use an appropriate scale, based on task's values, to obtain an appropriate activation of internal sigmoidal-units. To obtain the echo state property on $\mathbf{W}$ no specific rules has been discovered, however was observed with references to 1 that if ($|\lambda_{max}| < 1$) the system has the echo state property. In order to obtain a weight matrix $\mathbf{W}$ with desired characteristics an heuristic has been listed:

- Random generation of a sparse, uniform distributed in values, DR internal weight matrix $\mathbf{W}_0$. The size (N) of $\mathbf{W}_0$ should reflect the length of training data and the difficulty of the task, so it should not exceed, when possible, an order of magnitude of T/10 to T/2 as precaution against overfitting.
- Normalization of $\mathbf{W}_0$ to a matrix $\mathbf{W}_1$ with unit spectral radius by putting $\mathbf{W}_1 = \frac{1}{|\lambda_{max}|}\mathbf{W}_0$ where $|\lambda_{max}|$ is the spectral radius of $\mathbf{W}_0$ computable in a finite polynomial time.
- Scale $\mathbf{W}_1$ to $\mathbf{W} = \alpha\mathbf{W}_1$, where $\alpha < 1$ to give $\mathbf{W}$ a $|\lambda_{max}| = \alpha$. The value of α has to be chose with respect to input dataset dynamics changes, with smaller values for faster dynamics an larger values for slower ones. Right now no known rules are available to choose the best fit α value to use in matrix scale and the parameter must be hand tuned trying out several settings.

Step 2: Network training involves a series of mechanical steps as follows

- Initialize network to an arbitrary state (e.g. x(0) = 0);

- Train the network using training data, for times n = 0, ... , T, presenting teacher's input $\mathbf{u}$(n) and teacher-forcing output y(n - 1), by computing the activation (3.1).
- At time n = 0, where $\mathbf{y}$(n) is not defined, use $\mathbf{y}$(n) = 0
- For each time larger or equal than an initial washout time T_0, collect input/ reservoir/previous-output states ($\mathbf{u}$(n) $\mathbf{x}$(n) $\mathbf{y}$(n - 1)) concatenated as a new row into a state collecting matrix $\mathbf{M}$. In output we obtain a state collecting matrix of order (T - T_0 + 1) x (K + N + L).
- For each time larger or equal to T_0, collect the sigmoid-inverted teacher output $\tanh^{-1}\mathbf{y}(n)$ row-wise into a teacher collection matrix $\mathbf{T}$, to end up with a teacher collecting matrix $\mathbf{T}$ of size (T - T_0 + 1) x L.

Step 3: Compute output weights multiplying the pseudo-inverse of $\mathbf{M}$ with $\mathbf{T}$, obtaining a (K + N + L) x L sized matrix whose i-th column contains the output weights from all network units to the i-th output unit

$$(\mathbf{W}^{out})^t = \mathbf{M}^{-1}\mathbf{T} \tag{3.5}$$

Transpose the resulting matrix to obtain $\mathbf{W}^{out}$.

Step 4: In this stage the resulting network ($\mathbf{W}^{in}, \mathbf{W}, \mathbf{W}^{back}, \mathbf{W}^{out}$) is ready for use and can be driven by novel input sequences $\mathbf{u}$(n) using (3.1) and (3.2).

3.2.2 Stability Improvements

Some instability issues can occur when using the trained network; a possible solution [20] consist in the addition of a small white noise source $0.0001 \leq v(n) \leq 0.1$ to the state activation equation (3.1)

$$x(n+1) = f(W^{in}u(n+1) + Wx(n) + W^{back}y(n) + v(n)). \tag{3.6}$$

Another possible way to stem the over-fitting is applying Tikhonov regularization to $\mathbf{W}$ and find the parameter using cross-validation [32]

3.2.3 Augmented States Approach

Due to the high non-linear behaviors that systems sometimes presents it may be useful to model them with *augmented network states* that means add some non-linear transformation of activation states during Sampling phase.

3.2.4 Echo node type

Since the first approach and in most of the implemented ESNs, reservoir's internal nodes' activations were created as standard sigmoidal functions without any time dependence, limiting somehow tasks whose relies totally on this feature. To make up for this behavior another type of model known as **Leaky Integrator Neuron** (LIN) has been studied to absolve this task. The model $\mathbf{x}(t)$ is defined as

$$\dot{x} = \frac{1}{\tau}(-ax_s + f(w\mathbf{x})). \tag{3.7}$$

In the over expression, $\mathbf{w}$ is a weight vector of connections from all units $\mathbf{x}$ into the neuron x_s while f is the neuron's output non-linearity activation, in this case a sigmoid function tanh. The constant τ is a positive quantity of time use to manipulate activation dynamics; The a term is a non-negative decaying constant of neuron's previous state x_s. An update state of a Reservoir composed entirely by LINs with decay constant a_i, is described by

$$\dot{x} = \frac{1}{\tau}(-Ax + f(W^{in}u + Wx + W^{back}y)). \tag{3.8}$$

In this representation, A is a diagonal matrix containing decay constants in its diagonal. A state update equation can be obtained from the (3.8) in function of the *retainment rate* $r_i = 1 - a_i$ [19]

$$\mathbf{x}(n+1) = \mathbf{R}\mathbf{x}(n) + f(\mathbf{W}^{in}u(n+1) + \mathbf{W}\mathbf{x}(n) + \mathbf{W}^{back}\mathbf{y}(n)). \tag{3.9}$$

This model's condition take the Echo State Property existence to be compromised [20] if the spectral radius of $W + R$ in the (3.9) become greater than one ($|\lambda_{max}| > 1$).

3.2.5 Lyapunov Exponent

From [32] is know that spectral radius is influenced by input scale and/or bias terms: large inputs or bias leads to smaller effective spectral radius; For this reason, a more accurate measure of performance of the reservoir with respect to task's inputs has been investigated. The Lyapunov exponent $\mathbf{L}_k$ of a dynamical system is a measure that characterize the rate of separation of infinitesimally close trajectories in phase space, a space in which all possible states of the system are represented as an unique point in the space. In case of Reservoir, due to its input-driven dynamic nature, this value can not be calculated but in a sigmoidal model using a Jacobian matrix J_n calculated over a map of reservoir's internal units activation, a close related pseudo-Lyapunov exponent of a trajectory of N timesteps can be computed through the following equation, as reported in [35]

$$\widetilde{h} = \max_k \prod_{n=1}^{N} (r_K)^{\frac{1}{n}} \tag{3.10}$$

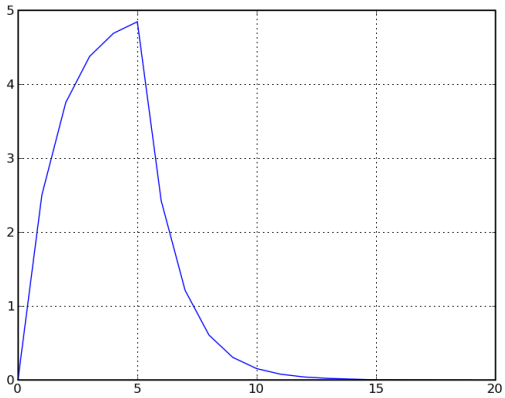

Figure 3.2: Leaky Integrator neuron model reach the peak and progressively leaks its state, described by $x(t + \delta_t) = x(t) + (-A_{leakRate}x + I) * \delta_t$.

where $r_K = \sqrt{|\lambda_K|}$, λ_K represents the kth eigenvalue of $J_n J_n^T$. Has been demonstrated [35] the validity of the pseudo-Lyapunov exponent $\widetilde{h}$ as a measure of input-output reservoir dynamics.

3.3 Liquid State Machines

Liquid State Machine approach has close links to the Echo State Network although these two theories have been independently developed and released. The technical approach behind this idea is based on the concept of "Liquid Computer" [27] imagined by Maass that consists in a liquid medium (a cup of coffee) that act as a filter perturbed by time-series inputs $\mathbf{u}(\cdot)$ in function of time, and a Readout (a pc with a camera) that captures all state changes in the liquid without memorize them. This idea is not applicable on real liquids due to physical limitations but has found a well applicable field in neuro-computation using a neural circuits base that acts as the "liquid", and a readout that maps output signal to specialize on a specific task. A mathematical model of "liquid computer" is called **Liquid State Machine (LSM)** and consists of a Reservoir, in this case called liquid, which processes an input time-series $\mathbf{u}(\cdot)$ into a liquid state $\mathbf{x}$(t) who integrates influences from inputs at all times prior **t**.

To be an LSM, a system with these characteristics needs to supply two fundamental properties [27],

1) Separation

All output-relevant differences in the preceding part of two input time series $\mathbf{u}_1(\cdot)$

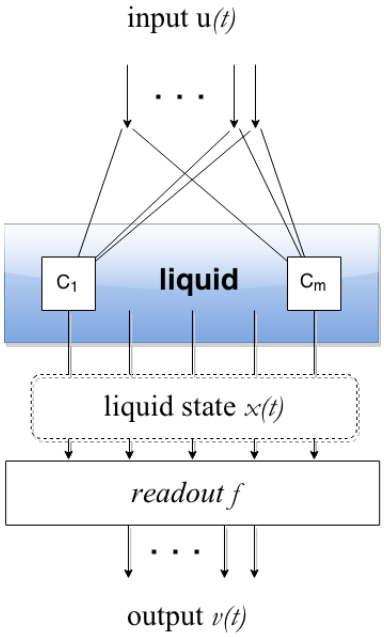

Figure 3.3: General structure of a LSM. A time-series input crosses the liquid and the resulting liquid state $\mathbf{x}$(t) is mapped by the readout who outputs the result.

and $\mathbf{u}_2(\cdot)$ (before time t) are reflected in the corresponding liquid states $\mathbf{x}_1(t)$ and $\mathbf{x}_2(t)$ of the system. This property should be fulfilled by the liquid reservoir.

2) Approximation
The readout has the capability to approximate any given continuous function f that maps current liquid states $\mathbf{x}(t)$ on current outputs $v(t)$. This property should be fulfilled by the readout function.

Regarding the theory, a basic and very high-level approach to the implementation of a specific target filter consist in

1. Choosing a suitable liquid as reservoir.
2. Elaborate numerous inputs $\mathbf{u}(\cdot)$ through the liquid and collect the output states $\mathbf{x}(t)$ at various time points.
3. Apply a supervised learning algorithm on a dataset of the form $(x(t), y_u(t))$ to train a readout function f such that the actual outputs $f(\mathbf{x}(t))$ are as close as possible to $y_u(t)$

The over stated procedure does not specify what liquid, nor a learning algorithm to choose for a specific filter implementation, such that could be possible choose a simpler reservoir composed by a collection of delay lines and use a more complex readout function like a neural network. However was observed [27] that a single perceptron is able to accomplish all type of classification tasks if inputs are first projected into an high, dimensional space. Hence a trade-off between reservoir and readout complexity, unbalanced in favor of the first, must be applied in a certain measure to achieve good performance in the resolution of the task.

3.3.1 Liquid node type

Given these considerations on system's properties, one major characteristic of LSM approach consist of a model of the reservoir based on neurons with activation functions based on biological synapse model theorized observing natural patterns found in the microcircuits of the brain. One example above the other is the **Leaky Integrate and Fire (LIF)** neuron model that in its basic form appears as [2]

$$I(t) - \frac{V_m(t)}{R_m} = C_m \frac{dV_m(t)}{dt} \tag{3.11}$$

that is an evolution of **Integrate and Fire** model which represents a neuron as the time derivative of the law of capacitance $Q = CV$. The term "Leaky" refers to $I_{th} = \frac{V_{th}}{R_m}$ that is a threshold for the cell who can fire an output if the input was enough intense or cancel any change in membrane potential. This type of node has an internal memory comparable with the leaky integrator neuron over cited that competes with the Reservoir intrinsic memory effect as stated above [35].

3.4 Backpropagation-Decorrelation Learning Rule

Two years after the approaches proposed by Jaeger and Maass, another independent study on recurrent networks has been published under the name of **Backpropagation-decorrelation (BPDC)** learning rule [34]. In its linear, $O(N)$ complexity, solution he combines

- Backpropagation of errors in one step;
- Temporal memory in network dynamics, adapted on the base of the decorrelation of the activations;
- Internal reservoir of non-adaptive neurons;

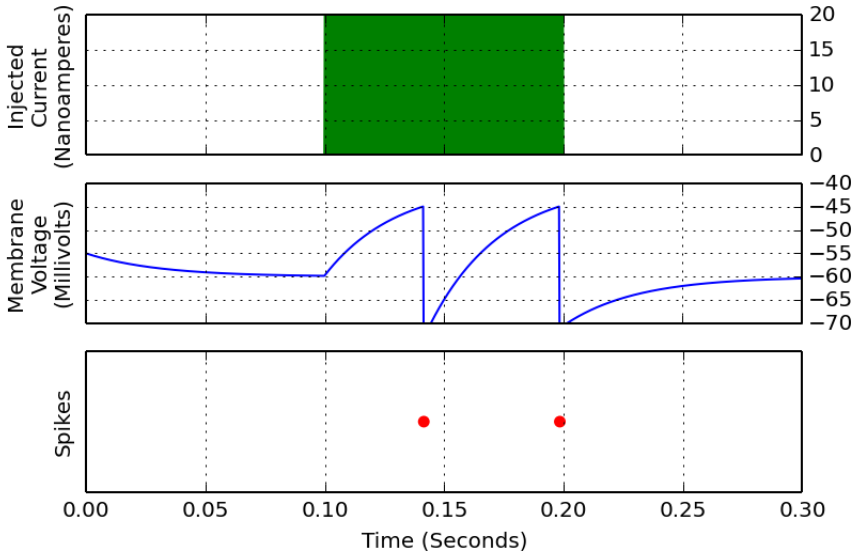

Figure 3.4: Leaky Integrate and Fire neuron model fires [13].

- A linear readout function implemented through output weights, and a feedback provided back to the reservoir;

And in addition, a formal technique has been developed to analyze and improve online the stability of network's configuration. A general model for this methodology as reported in [34] is composed of a fully connected recurrent reservoir with fixed weights, which receives a constant dummy bias input in addition to external inputs, connected through the only set of trainable weights in the system to the output neuron who provides feedback connections into the reservoir.

3.4.1 BPDC bases

The equation of activation states in the reservoir is as follows

$$\mathbf{x}(k + \Delta t) = (1 - \Delta t)\mathbf{x}(k) + \Delta t W f(x(k)) + \Delta t W_u u(k). \tag{3.12}$$

where the terms means:

- $x_i, i = 1, ..., N$ are the states at time $k < (k + \Delta t)$;
- f is a standard, sigmoidal, differentiable activation function applied component wise to the vector $\mathbf{x}$;

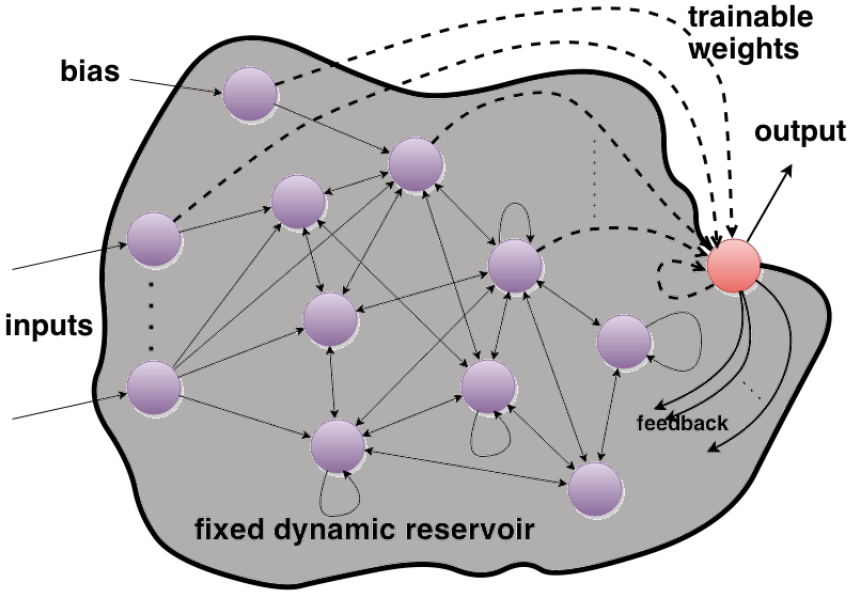

Figure 3.5: A generic BPDC model as described in Steil 2006 [34]

- $\mathbf{W} \in \mathbb{R}^{NxN}$ is the internal weight matrix, initialized with small random values in defined weight initialization interval $[-a, a]$ which can be adaptively rescaled to achieve system's stability;
- $\mathbf{W}_u$ is the input weight matrix;
- k is a discrete time variable defined as $k = \widehat{k}\Delta t, \widehat{k} \in N_+$, where Δt determines the discrete or continuous dynamics of the reservoir;

Also in this case, inner neurons behave as a dynamical reservoir triggered by external inputs and providing a dynamic memory as discussed previously for other methods, and the output layer linearly combines the outcoming values to predict the desired output. The weight update's equation in **Backpropagation-decorrelation** is:

$$\Delta w_{ij}^{BPDC}(k+1) = \frac{\eta}{\Delta t} \frac{f(x_j(k))}{\sum_{s \in O} f(x_s(k))^2 + \varepsilon} \gamma_i(k+1) \tag{3.13}$$

where

$$\gamma_i(k+1) = \sum_{s \in O}((1 - \Delta t)\delta_{is} + \Delta t w_{is} f^{'}(x_s(k)) \times e_s(k) - e_i(k+1). \tag{3.14}$$

In the equation (3.13)

- $O \subset \{1, ..., N\}$ is a set of indices of output neurons;
- η is the learning rate;
- ε is a regularization constant usually around 0.002;
- $e_s(k)$ are the non-zero error components for $s \in O$ at time $k : e_s(k) = x_s(k) - y_s(k)$ with respect to the teaching signal $y_s(k)$.

To justify the provided rule in [34, 33] a constraint's optimization problem has been solved, minimizing the quadratic error with respect to the target output y for K timesteps

$$E = \frac{1}{2} \sum_{k=1}^{K} \sum_{s \in O} [x_s(\widehat{k}\Delta t) - y_s(\widehat{k}\Delta t)]^2 \tag{3.15}$$

where constraint's equations for $k = 0, ..., K-1$ are obtained from the activation state equations (3.12)

$$g(k+1) \equiv -x(k+1) + (1 - \Delta t)\mathbf{x}(k) + \Delta t W f(x(k)) = 0. \tag{3.16}$$

In the case of BPDC, this minimization problem has been approached [33] using an algorithm proposed by Atiya and Parlos (APRL) to compute weight changes, using g constraint equations to obtain a "virtual target" by differentiating E with respect to the state x:

$$\Delta x = -\left(\frac{\partial E}{\partial x}\right)^T = -(e^T(1), ..., e^T(K))^T \tag{3.17}$$

where

$$e_s(k) = \begin{cases} x_s(k) - y_s(k), & s \in O, \\ 0, & s \notin O \end{cases} \tag{3.18}$$

Then, *virtual teacher forcing* has been applied to compute weight's updates Δw to guide network's changes by $x + \eta \Delta x$ expression:

$$\frac{\partial g}{\partial w} \Delta w \approx -\eta \frac{\partial g}{\partial x} \Delta x \tag{3.19}$$

and this is done applying APRL to solve the (3.19), obtaining a full autocorrelation matrix C_k of network activities. In BPDC, some adjustment are done to APRL algorithm and the result is the (3.13) pointing out the Backpropagation-decorrelation learning rule as an improvement over Atiya-Parlos method, mixing the new point of view in Reservoir Computing methodology with algorithms of the literature and in a certain way, acts as a link between the new and old school of thought regarding aRNN's training.

3.5 EVOlution of recurrent systems with LINear Output (Evolino)

The last method who will be covered in this paper, regards a technique [31] much different from others which relies on *Long short-term memory (LSTM)* [17] aRNN.

LSTM networks overcome traditional aRNN problems (e.g. gradient vanishing) allowing the use of standard backpropagation rule, obtaining unreached results before with standard recurrent networks. However can sometimes occur, due to characteristics of algorithms based on gradient, a lock of the result in a local minimum obtaining thus a sub-optimal output. A possible solution to avoid the problem consist in the use of evolutionary algorithms to search in the space of aRNN's weight matrices, learning quickly how to solve reinforcement learning jobs. Evolutionary methods, since they do not rely on teacher's input, can be very slow in supervised learning applications; However in this last section, we are going to cover a general framework for supervised sequence learning called **EVOlution of recurrent systems with LINear Output (Evolino)** [31] which combines *neuroevolution* and *linear methods* (e.g. linear regression) to solve time-series tasks.

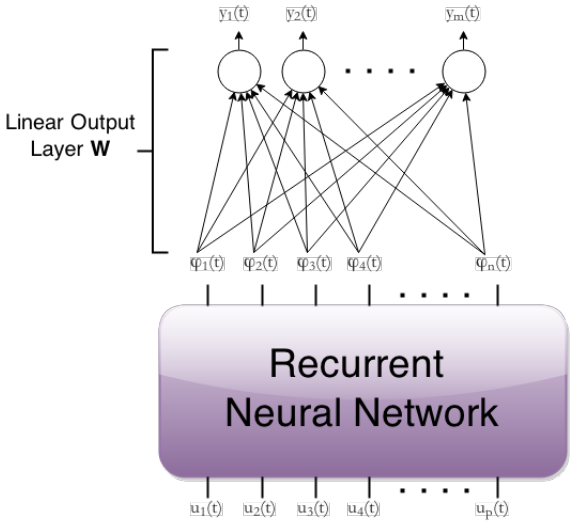

Figure 3.6: Generic Evolino network.

Network's output at time t is computed with the following equations:

$$\phi(t) = f(u(t), u(t-1), ..., u(0)), \tag{3.20}$$

where terms

- $\phi(t) \in \mathbb{R}^n$ is the Reservoir output;
- $f(\cdot)$ is the network's activation function in function of the entire input history;

and the output equation

$$y(t) = W\phi(t). \tag{3.21}$$

where terms

- $y(t) \in \mathbb{R}^m$;
- W is a weight matrix.

To evolve $f(\cdot)$ that minimizes the error between the correct output d and y, no specific algorithms are specified but a two-phase procedure [31] must be applied on the network:

Phase 1: submit to the network a training set of sequences $(u^i, d^i), i = 1, .., k$ of length l^i. For each input pattern $u^i(t), t = 0, .., k$, feed the network, produce vector activation $\phi^i(t)$ and store it as a row in a matrix $\phi \in M_{L_k \times n}(\mathbb{R})$. In another matrix D store the teacher output for each time step. After the computation of all activations, output weights **W** are computed using linear regression from ϕ to D. The row vectors in ϕ form a non-orthogonal basis that, combined linearly by **W** approximates D.

Phase 2: Present again the dataset to the network to obtain predictions $y(t)$. Computes ΔE, errors with respect to desired teacher output D, and use it as fitness measure to minimize by evolution.

Evolino tries to evolve not the network model directly, but instead its output bases to obtain a good representation for the model. Most used Evolino's preset is composed of an LSTM network, evolved using a variant of **Enforced SubPopulation (ESP)** neuroevolution algorithm that coevolve, through a cross-over algorithm, separate subpopulations of neurons, accelerating neuron's specialization in different subfunctions needed to form good networks, due to the closeness of evolution that ensures members of different sections will never be mated. The division of population, in addition to performance boost, reduces noise in the neuron's fitness measure ensuring a more balanced representation of each neuron in every evolved network. These features of ESP, allows more efficiency than its ancestor method called **Symbiotic Adaptive NeuroEvolution (SANE)**, which evolves neurons in a single population. In Evolino, ESP promotes individual evolution through *Cauchy-distributed mutation* as in the following algorithm:

Step 1: Initialization

- Set H, number of hidden units that will be evolved;

- Create a subpopulation of n neuron's chromosome, each of which encodes a neuron's input and recurrent connection weights with a string of random real numbers, for each $h_i, i = 1, .., H$;

Step 2: Evaluation

- Randomly, select a neuron from each of the H subpopulations and combine them to create a new aRNN.
- Evaluate the freshly created network on the task and collect a fitness score;
- Add score to the cumulative fitness value of each neuron that partecipated in the network.
- Repeat the procedure until each neuron participated in m evaluations.

Step 3: Reproduction

- Rank each subpopulation by fitness function using neuron's score;
- Duplicate top quarter chromosomes in each subset, then alter their weight values adding noise obtained through the Cauchy distribution $f(x) = \frac{\alpha}{\pi(\alpha^2+x^2)}$ where α determine the width of the distribution.
- Replace the lowest-ranking half of the original corresponding population with the copies;

Step 4: Repeat

- If the fitness of the best network does not improve for a predetermined number of generations, apply **burst mutation** procedure.
- Repeat **Step 2** and **Step 3** until a sufficiently fit network is found;

3.5.1 Burst Mutation procedure

This procedure consists in a research in the space of modifications to find best solution.

1. Save best neurons in each subpopulation and discard the others;
2. Create from the saved elite set, substitutes of deleted neurons, through the addition of some Cauchy noise to each copy.

This operation allows ESP to continue evolving after a first population convergence, injecting new diversity into the subpopulation's set.

3.6 Different approaches

In previous sections, we looked at the actual techniques in the panorama of the training for artificial Recurrent Neural Network, collected under the name of Reservoir Computing. Each of them is driven by almost the same approach that consists in feeding a time-series input into an assembled network with fixed internal weight that acts as a sort of *resonance box*, connected through some weights to one or more output units composing an *interpreter* with the task of approximate signals generated by the system after an appropriate tuning. The only trainable weights in the entire system are the output ones. A good set of output weights together with an appropriate scale of inputs and the well-position of the reservoir system, can actually produce a state-of-art technique for the "training" of recurrent networks both in software as hardware implementation [30] for various tasks. One of the differences to be noted is the basic idea that attends with each method. **Echo state networks** were originally thought to exploit aRNN as a sort of black-box suited for modeling of dynamical systems, often involved in engineering tasks, modeling signals with a sigmoidal activation state, not aimed to emulation of any biological features, free of memory. While, on the other hand, **Liquid State Machines** embody the attempt to give biological similar dynamics to aRNN with the use of spiking neurons, using a model of synapse obtained in years from studies on brain's biological structures. We treated then nearly the latest discovery technique, note as **Backpropagation-decorrelation** learning rule which improves a method already known in aRNN's training, to achieve the solution of the problem in a way similar to that pursued by the ESN but without the integral cut in relation to *old* methods. A fourth most recent [2007] methodology, known as **Evolino** has the most different approach in RC assumed until now. While it maintains a "physical" structure comparable with others approaches, it operates on LSTM models and attempts to train the Reservoir using evolutionary algorithms, hypothesize as others RC methods' limit could reside in the randomic and static nature of the recurrent neural network which composes the Reservoir. Conceptually [24], aRNN's training methods increasingly deviate from standard methods applied on the entire network, towards exclusive output-connection enhancement:

↓ **0:** BPTT

↓ **1:** APRL

↓ **2:** BPDC

↓ **3:** ESN/LSM

3.7 Technology example: Echo State Network

In this short section we aim to provide a practical example of how an implementation of Reservoir Computing can be approached, given a set of data known in literature. Indeed, the data that the ESN should approximate, are taken from an online publication at [22] and obtained from the equation of Mackey-Glass (delay 17) [16].

Taken the data, what has been done is to follow the directives from the guide [23] to implement an ESN that approximates the input signal.
Taken into account all dynamics of the equation, the following basic code in Octave (a language compliant with Matlab) has been implemented:

```
% ESN implementation: rc_esn.m
clear;more off;clc;     %some global cleaning
disp '% % % % % % % % % % % % % % % % % % % % %'
disp '%  MackeyGlass signal reproduction  %'
disp '% % % % % % % % % % % % % % % % % % % % %'
% Global Parameters
data = load('MackeyGlass_t17.txt'); %data
alpha = 0.3;            %leak rate
beta = 1e-8;            %regularization term
K = 1;                  %input nodes
N = 1000;               %DR size
L = 1;                  %output nodes
Ttrain = 2000;          %train time
Ttest = 2000;           %test time
Twashout = 100;         %leak init time
Terr = 1000;            %error time
rhoScale = 1.25;        %DR scale factor
M = zeros(1+K+N,Ttrain-Twashout);%bias;input;state foreach T
x = zeros(N,1);         %store an activation
D = data(Twashout+2:Ttrain+1)'; %correct data
Y = zeros(L,Ttest);         %predicted data
y = data(Ttrain+1);         %store a computed output
Win = rand(N,1+K) - 0.5;    %input connections
W = randi([-1.5,1.5],N);    %DR internal structure
Wout = zeros(1,1+K+N);      %output connection to readout
opt.tol = 1e-3;         %tollerance for eigs search
% complete the scale factor for W
rhoScale = rhoScale / abs(eigs(W,1,'lm',opt));
W = W .* rhoScale;      %scale spectral radius of W
for T = 1:Ttrain
    % leaky integrator - take in account the past x
    %   to predict future states
    % run for the first Twashout time steps to sync
    %   internal states with the input
    x = (1 - alpha) * x + alpha * tanh(Win*[1;data(T)] + W*x);
```

```
36      if T > Twashout
37          %bias;input;internal state x
38          M(:,T-Twashout) = [1;data(T);x];
39      end
40  end
41  Mt = M'; %transpose of state collecting matrix
42  %compute output connection using normal equations
43  Wout = D * Mt * inv(M*Mt + beta * eye(1+K+N));
44  for T = 1:Ttest
45      %compute predicted output, starting from last train
46      x = (1 - alpha) * x + alpha * tanh(Win * [1;y] + W * x);
47      y = Wout * [1;y;x];
48      Y(:,T) = y;
49  end
50  %compute Mean Squared Error measure
51  mse = sum((data(Ttrain+2:Ttrain+Terr+1)'-Y(1,1:Terr)).^2)./Terr;
52  disp( ['Mean-Squared-Error = ', num2str( mse )] );
53
54  % plot generated data vs. correct ones
55  figure(1);
56  plot( data(Ttrain+2:Ttrain+Ttest+1),'r');
57  hold on;
58  plot( Y', 'b' );
59  hold off;
60  axis tight;
61  legend('Target signal', 'Free-running predicted signal');
62  % plot a subset of internal activation
63  figure(2);
64  plot( M(1:20,1:200)' );
65  % plot a bar graph of output connection's weights
66  figure(3);
67  bar(Wout')
```

The output:

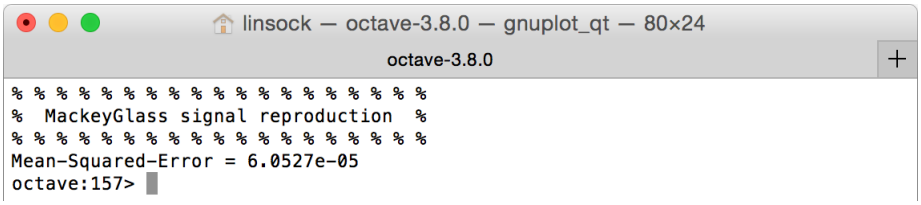

Figure 3.7: The output represents the mean squared error between the signal in input and the generated

The results obtained fit well the input signal, however not always the output model adapt so tight to that in input. Indeed there is a strong dependency of performance

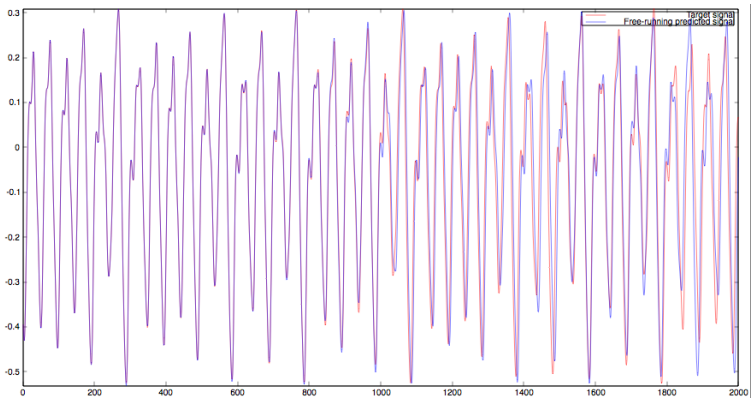

Figure 3.8: Generated signal $\mathbf{y(n)}$ in the first 2000 steps

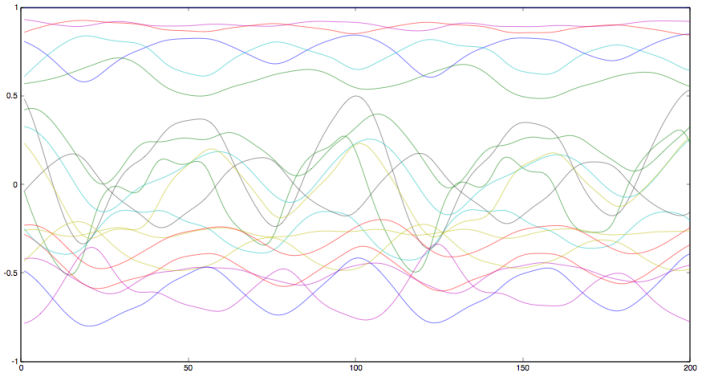

Figure 3.9: Plot of a restricted subset of internal activations of the reservoir $\mathbf{W}$

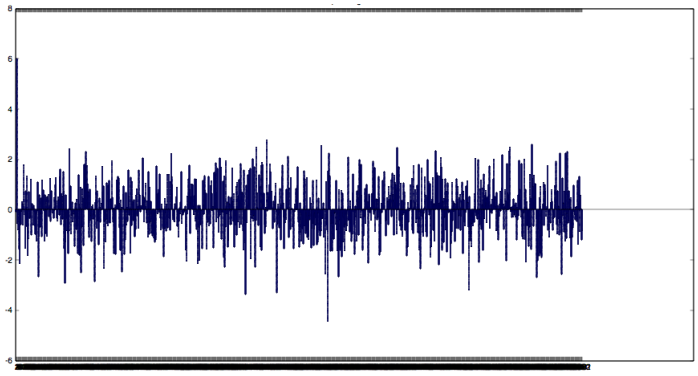

Figure 3.10: Bar graphic of output weights contained in $\mathbf{W^{out}}$

with respect to the the random initialization of internal structures of reservoir and input connections, a constraint on performance which is still object of investigation.

3.8 Application Domains and Future Steps

RC methods have been widely employed in various kinds of engineering tasks [25] like temporal pattern classification and/or generation, time series prediction [36], memorizing, or controlling nonlinear systems. Now we are going to provide a report of the actually applied methods in various fields:

- ◇ **Speech Recognition:**
  The first approach has been focused on recognition of Japanese vowels and digits [25]. After that, the first effective test of recognition on a continuous speech has been based on a large set of predictive Echo state networks, who yielded good results, performing better than the actual state-of-the-art technique in this field. Attempts to speech recognition in an LSTM context, evaluated the Evolino approach on a 100.000 units network worst in comparison to a Gradient approach allowed by this particular network architecture [31]. Enhanced performance was achieved in ESN-HMM hybrid models and an active research is based on a neuro-inspired LSM approach [15] who also denotes high potentialities for future developments.

- ◇ **Handwriting Recognition:** Currently, most interest in handwriting recognition was directed onto an aRNN employment to handle time-series of pixels. This is the approach studied in RC's recent attempts: texts has been sampled in a time-series input of pixels and used, coupled with teacher-output, to feed a hierarchical architecture of reservoirs. There has not been necessity to segment the data before use them thanks to the composition of the recognizer used in the experiment, actually an improvement compared to state-of-the-art techniques employed until now.

- ◇ **Robotics:** Deadbeat controllers can be obtained through a careful training of Echo State Networks as described in detail in ESN patent document. ESNs are currently explored also as mouldable neural pattern generators in the European FP7 project AMARSi.

- ◇ **Financial Forecasting:** In the field of time-series many studies has been headed to success results using RC approaches and researches has shown [36] how much data regularization and reservoir size influences the resulting performance. Some techniques, like seasonal decomposition and a collective vote approach using many of "small size" reservoir to obtain a more balanced result, were introduced in order to achieve satisfactory results.

- ◇ **Medical:** An improvement over state-of-the-art technique (already held by RC methods) has been obtained in a study taken at Ghent university, applying reservoir computing to epileptic seizures real-time detection.

Other RC methods possibilities are biological and cognitive phenomena modeling, in particular using LSM paradigm within includes spiking neurons to mime biological neuron's behavior. The application of Reservoir Computing methodologies is taking place with very good results, leading the use of recurrent neural networks for many tasks in various branches of scientific environment which do not have never applied seriously due to poor performance got with previous learning algorithms. Although the exponential improvements have revitalized aRNN as an usable tool, researches continue to enlarge the spectrum of possible applications. This objective in mind, one of major research fields consists of *Automated Reservoir's Optimization* for a particular task, operation until now done by a manual search on specific problem [24]. Another fundamental research's topic consists of the *Stability* of the reservoir's states during training, achieved nowadays in some ways adding noise or ridge regression as regularization parameter, solutions that needs to be improved a lot with a consistent indications behind the application of a regularization of some type. *Reservoir Architectures* has became another research field for future improvements, indeed a direct correlation to output's goodness was observed in spread applications of these techniques.

Conclusion

In this overview, we presented Machine Learning from one of its various points of view, to denote its actual applications in various fields of engineering and data analysis, since today it is the base of the execution of a huge quantity of automated tasks. We have started our report from the simplest techniques involved in prediction and classification like the Regressions (chapter 1), passing through the application of more complex algorithms like Neural Networks that currently dominate the applications of automated learning. Moreover, we looked at three recent techniques, independently developed each others, involved in the training of artificial Recurrent Neural Networks (chapter 2), gathered under the common name of Reservoir Computing, that overwhelm in efficiency the historical methods, in almost every aspect (chapter 3). This family of approaches has caught on, in response to poor results obtained using methodologies derived from the resolution methods of feedforward networks, historically adopted to train aRNN. From a look at each method, is observable that each one has some aspects not enough satisfying to unleash of the full computational power of the network, summarizable in:

- ⋆ Reservoir production, that consists in each process involved in the production of the reservoir: size, node type, architecture involved and weights assumed by the connections in the network are all fundamental aspects of this piece of the system. For instance, in biological brain, most structures have a predefined constitution and various types of learning are involved in the formation of the "networks". This collides with the actual random constitution of the reservoir and highlights a source of improvements;

- ⋆ Readout production, that consists in the choice of the readout layer placed as output of the reservoir to analyze and elaborate signals generated from the first.

Reservoir computing counts among its merits that it has initiated the use of aRNNs in real world problem such before was not achievable and also, to have taken a step forward to the creation of processes similar to that who occur in biological brains, using specific models of neuron and biological-inspired architectures. Another characteristic of RC methods, is the separation between the part that generates signals from that who interprets them, implicitly providing an easy testbed for modifications and evaluation of new best-practice for this methodology. The research for methods that define formally a

satisfying reservoir who permits a better exploitation of aRNN for each purpose, compose today the main field of research in Reservoir Computing, that will allow a day perhaps to achieve full computational power from this powerful tool.

Bibliography

[1] *Turing equivalence of neural networks with second order connection weights*, 1991.

[2] K. Christof and S. Idan. Methods in neuronal modeling : from ions to networks, 1999.

[3] Crowd-Edited. Gradient. `http://en.wikipedia.org/wiki/Gradient`, February.

[4] Crowd-Edited. Gradient descent. `http://en.wikipedia.org/wiki/Gradient_descent`, February.

[5] Crowd-Edited. Linear predictor function. `http://en.wikipedia.org/wiki/Linear_predictor_function`, February.

[6] Crowd-Edited. Linear regression. `http://en.wikipedia.org/wiki/Linear_regression`, February.

[7] Crowd-Edited. Logistic regression. `http://en.wikipedia.org/wiki/Logistic_regression`, February.

[8] Crowd-Edited. Recurrent neural network. `http://en.wikipedia.org/wiki/Recurrent_neural_network`, February.

[9] Crowd-Edited. Regression analysis. `http://en.wikipedia.org/wiki/Regression_analysis`, February.

[10] Crowd-Edited. Sigmoid function. `http://en.wikipedia.org/wiki/Sigmoid_function`, February.

[11] Crowd-Edited. Statistical model. `http://en.wikipedia.org/wiki/Statistical_model`, February.

[12] Crowd-Edited. Vanishing gradient problem. `http://en.wikipedia.org/wiki/Vanishing_gradient_problem`, February.

[13] ekaakurniawan. 3nb - neural network notebook. GNU Project mantained at http://ekaakurniawan.github.io/3nb/.

[14] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[15] A. Ghani, T. McGinnity, L. Maguire, L. McDaid, and A. Belatreche. Neuro-inspired speech recognition based on reservoir computing. Technical report, University of Ulster, 2010.

[16] L. Glass and D. M. Mackey. Mackeyglass equation. `http://www.scholarpedia.org/article/Mackey-Glass_equation`.

[17] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[18] S. H. Horng and M. Sur. Visual activity and cortical rewiring: activitydependent plasticity of cortical networks. *Progress in Brain Research*, 157, 2006.

[19] H. Jaeger. *A tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the "echo state network" approach.* Fraunhofer Institute for Autonomous Intelligent Systems (AIS).

[20] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. Technical report, German National Research Center for Information Technology, 2001.

[21] M. I. Jordan. Serial order: A parallel distributed processing approach. Technical Report 8604, San Diego: University of California, Institute for Cognitive Science., 1986.

[22] M. Lukoeviius. Mackeyglass distribution. `http://minds.jacobs-university.de/pubs`.

[23] M. Lukoeviius. A practical guide to applying echo state networks. Technical report, Jacobs University Bremen, Campus Ring 1, 28759 Bremen, Germany, 2012.

[24] M. Lukoeviius and H. Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3:127–149, 2009.

[25] M. Lukoeviius, H. Jaeger, and B. Schrauwen. Reservoir computing trends. Technical report, ENS Cachan, 2012.

[26] J. McCarthy. What is artificial intelligence? `http://www-formal.stanford.edu/jmc/`.

[27] T. Natschlager, W. Maass, and H. Markram. The "liquid computer": A novel strategy for real-time computing on time sries. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8:39–43, 2002.

[28] J. R. Newton and M. Sur. Rewiring cortex: functional plasticity of the auditory cortex during development. Technical report, Massachusetts Institute of Technology, 2008.

[29] A. Ng. Stanford university - machine learning. `http://ml-class.org`.

[30] Y. Paquot, F. D. adn A. Smerieri, J. Dambre, B. Schrauwen, M. Haelterman, and S. Massar. Optoelectronic reservoir computing. *Scientific Reports*, 2(287), 2012.

[31] J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. Gomez. Training recurrent networks by evolino. *Neural Computation*, 19:757–779, 2007.

[32] B. Schrauwen, D. Verstraeten, and J. V. Campenhout. An overview of reservoir computing: theory, applications and implementations. In *ESANN'2007 proceedings*, 2007.

[33] J. J. Steil. Backpropagation-decorrelation: online recurrent learning with o(n) complexity. Technical report, Neuroinformatics Group, Faculty of Technology University of Bielefeld, Germany, 2004.

[34] J. J. Steil. Online stability of backpropagation-decorrelation recurrent learning. *Neurocomputing*, 69:642–650, 2006.

[35] D. Verstraeten, B. Schrauwen, M. D'Haene, and D. Stroobandt. An experimental unification of reservoir computing methods. *Neural Networks*, 20, 2007.

[36] F. Wyffels and B. Schrauwen. A comparative study of reservoir computing strategies for monthly time series prediction. *Neurocomputing*, 73:1958–1964, 2010.

[37] X. Yan. *Linear Regression Analysis: Theory and Computing*. World Scientific, june 2009.