Università di Pisa

Dipartimento di Informatica

Implementazione di una Echo State Network

Esperienze di programmazione

Professore:
Francesco Romani

Presentata da:
Eleonora Di Gregorio
(520655)

Aprile
A.A. 2017/2018

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il *Reservoir Computing* (RC) è un metodo di grande interesse nel campo della ricerca sulle reti neurali ricorrenti (RNR), grazie alla sua semplicità e alle sue buone performance su una certa varietà di *task*. Con *Reservoir Computing* ci si riferisce ad una classe di modelli di reti neurali ricorrenti che sono caratterizzati da una parte dinamica ricorrente e una parte di output semplice. L'idea alla base del *Reservoir Computing* è quella di prendere una rete ricorrente random di semplici nodi, ad esempio neuroni sigmoidali,chiamata appunto *reservoir*. I nodi sono collegati tra loro, associando dei pesi ai collegamenti che possono essere riscalati per imporre il regime dinamico desiderato. Il *reservoir* permette dunque di mappare un input in uno spazio di dimensione maggiore che nel caso di *task* di classificazione puó aumentare la probabilitá di separazione lineare tra i dati. Oltre a questo permette anche di tenere traccia della storia degli input nel tempo, questi aspetti lo rendono ideale per molti *task* interessanti nel mondo reale che richiedono elaborazione temporale e *mapping* non lineare.
È proprio in questo contesto che si collocano le *Echo State Network* (ESN), affermandosi per la loro praticità , per essere concettualmente semplici e per l'efficienza in fase di allenamento dovuta al fatto che non vengono addestrati i pesi delle connessioni ricorrenti. Per ottenere delle buone performance sui *task* vanno però tenuti in considerazione diversi aspetti e proprietà di

cui si parlerà in seguito. Questa semplice implementazione delle *Echo State Network* vuole essere un primo approccio guidato alla conoscenza dei meccanismi sui cui si basa questo modello e alla loro applicazione. In appendice è riportato il codice matlab della ESN sviluppata.

Capitolo 2

Background

Prima di iniziare la descrizione del modello delle ESN è necessario definire la notazione che verrà utilizzata in seguito e fissare alcuni concetti importanti riguardo alle reti neurali ricorrenti.

2.1 Notazione

Si utilizzano le lettere in corsivo maiuscolo e minuscolo per denotare gli scalari, le lettere in grassetto minuscolo per i vettori, quelle in grassetto maiuscolo per le matrici. Sia $\mathbf{X}$ una matrice, allora $\mathbf{X}_{i,j}$ indica l'elemento alla i-esima riga e la j-esima colonna di $\mathbf{X}$; inoltre $\mathbf{X}_{:,j}$ denota la j-esima colonna di $\mathbf{X}$ e $\mathbf{X}_{i,:}$ la i-esima riga.
In particolare un vettore di input è rappresentato da $\mathbf{u}$, oppure da $\mathbf{u}(t)$ se si vuole indicare la posizione che occupa l'input all'interno di una sequenza. Quest'ultima viene rappresentata usando la notazione s($\mathbf{u}$) = [$\mathbf{u}$(1),$\mathbf{u}$(2)... $\mathbf{u}$(n)], dove n è il numero di elementi che appartengono ad una sequenza, definiti anche *timesteps* nel caso in cui questa rappresenti l'input di un *task* temporale. Analogamente per un elemento e una sequenza di output si utilizzano $\mathbf{y}$ e s($\mathbf{y}$), rispettivamente.

2.2 Formulazione dei task

Un *task* nel contesto del *machine learning* consiste nell'apprendere una relazione funzionale tra un dato input $\mathbf{u}(t) \in \mathbb{R}^{N_u}$ ed un output atteso $\hat{\mathbf{y}}(t) \in \mathbb{R}^{N_y}$, dove $t = 1, ..., T e T$ è il numero di elementi $(\mathbf{u}(t), \hat{\mathbf{y}}(t))$ del *dataset* per il training, chiamati anche *data points*.

2.2.1 Task non temporali

Si ha un *task* non temporale quando i *data points* sono indipendenti gli uni dagli altri, l'obiettivo è approssimare una funzione $\mathbf{y}$(t)= y($\mathbf{u}(t)$) che minimizzi la misura dell'errore $E(\mathbf{y}, \hat{\mathbf{y}})$.

2.2.2 Task temporali

Si ha un *task* temporale quando $\mathbf{u}$ e $\hat{\mathbf{y}}$ sono segnali in un dominio discreto nel tempo n = 1,...,T, e l'obiettivo è approssimare una funzione $\mathbf{y}(t) = y(..., \mathbf{x}(t-1), \mathbf{x}(t))$ tale che minimizzi $E(\mathbf{y}, \hat{\mathbf{y}})$. A differenza dei *task* non temporali la funzione che bisogna apprendere ha memoria, ovvero dipende dalla storia degli input.

2.3 Reti Neurali Ricorrenti

Le reti neurali ricorrenti sono una classe di modelli di reti neurali biologicamente ispirata e ampiamente utilizzata per il trattamento di dati sequenziali. In una RNN gli elementi che rappresentano i neuroni, chiamati unità, sono collegati tra loro a simulare le connessioni sinaptiche e sono l'origine delle attivazioni che percorrono la rete attraverso questi collegamenti. La principale caratteristica delle reti neurali ricorrenti, dalla quale ne deriva anche il nome, è che la topologia delle connessioni presenta dei cicli, ovvero sono ammesse connessioni sinaptiche ricorrenti (chiamate anche *feedback*). Per tale ragione

la RNN è il modello più simile al cervello umano. L'esistenza dei cicli ha profondi impatti come riportato in [1]:

- Una RNN può sviluppare autonomamente un sistema dinamico nel tempo lungo i suoi percorsi di connessione ricorrenti.
- Se guidata da un input, una RNN preserva nel suo stato interno una trasformazione non lineare della storia dell'input,ovvero ha una memoria dinamica.

Ogni unità della RNN rappresenta uno stato $x(t) \in \mathbb{R}$, la sua attivazione è calcolata in base al suo stato corrente ed al valore che riceve in input. Sia $\mathbf{x}(t\text{-}1)$ il vettore che rappresenta lo stato delle unità che al tempo t ricevono in input $\mathbf{u}(t)$, le loro nuove attivazioni saranno:

$$\mathbf{x}(t) = f_x(\mathbf{u}(t), \mathbf{x}(t-1)) = f(\mathbf{W_{in}}u(t) + \mathbf{W}\mathbf{x}(t-1) + \mathbf{b}) \tag{2.1}$$

dove:

- f è la funzione di attivazione, generalmente la funzione simmetrica *tanh* o la funzione sigmoidale logistica,
- $\mathbf{W_{in}}$ è la matrice dei pesi di ingresso, ovvero pesi associati alle connessioni tra l'input e lo stato interno,
- $\mathbf{W}$ è la matrice dei pesi delle connessioni tra le diverse unità della rete,
- $\mathbf{b}$ è un vettore di bias che vengono applicati alle unità.

L'output della rete viene poi calcolato secondo la seguente formula:

$$\mathbf{y}(t) = f_y(\mathbf{x}(t)) = f_{out}(\mathbf{W_{out}}\mathbf{x}(t)) \tag{2.2}$$

dove la f_{out} può essere una funzione di attivazione lineare, ad esempio la funzione identità, o una funzione non lineare. L'approccio classico per l'allenamento delle RNN è quello basato su discesa di gradiente, cioè vengono

adattati tutti i pesi $\mathbf{W_{in}}$, $\mathbf{W}$, $\mathbf{W_{out}}$ rispettivamente in base a $\partial E/\partial \mathbf{W_{in}}$, $\partial E/\partial \mathbf{W}$, $\partial E/\partial \mathbf{W_{out}}$, con lo scopo di minimizzare $E(\mathbf{y}, \hat{\mathbf{y}})$.

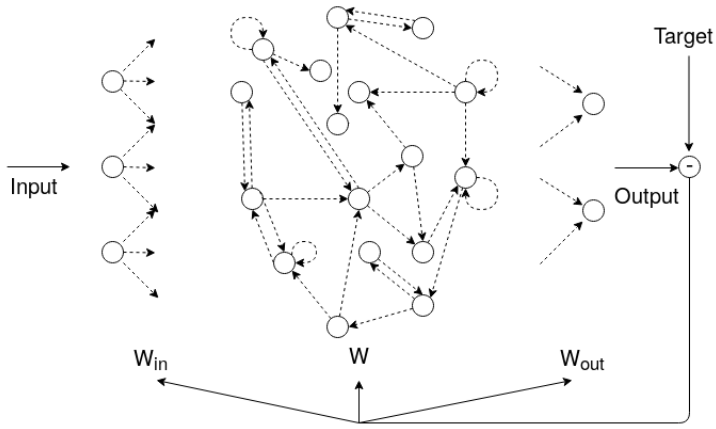


Figura 2.1: Metodo di allenamento basato su discesa di gradiente che adatta tutti i pesi delle connessioni.

Le reti neurali ricorrenti si presentano come strumenti molto promettenti per elaborare sequenze temporali non lineari per due ragioni. Innanzitutto rappresentano una approssimazione dei sistemi dinamici, in secondo luogo si avvicinano al modello del cervello biologico grazie alle connessioni ricorrenti. Tuttavia vi sono delle problematiche legate all'apprendimento evidenziate in [1] e riportate qui di seguito:

- Non può essere garantita la convergenza del metodo di allenamento a discesa di gradiente, vi sono dei punti in cui le informazioni sul gradiente possono essere mal definite.

- L'aggiornamento di un solo parametro può essere computazionalmente costoso, cioè possono essere necessari molti cicli di aggiornamento. Questo comporta lunghi tempi di addestramento che rendono le RNN modelli convenienti utilizzando solo poche unità.

- È molto difficile apprendere relazioni che richiedono molta memoria perché le informazioni fornite dal gradiente subiscono una dissolvenza esponenziale nel tempo.
- Gli algoritmi di apprendimento necessitano di profonde conoscenze ed esperienza per essere applicati con successo.

In questa situazione di lento progresso, nel 2001 un nuovo approccio al design e all'allenamento delle RNN, noto come *Reservoir Computing*, venne proposto indipendentemente da Wolfgang Maass [2] e da Herbert Jaeger [3].

Capitolo 3

Echo State Network

3.1 Il modello di base

Come evidenziato nella sezione 2.3 le diverse problematiche relative all'allenamento delle reti neurali ricorrenti hanno portato allo sviluppo del *Reservoir Computing* . L'approccio del *Reservoir Computing* differisce dalla tecnica di apprendimento citata in precedenza perché crea una separazione concettuale e computazionale tra una parte dinamica chiamata *reservoir* ed una parte chiamata *readout*. Il *reservoir* è una rete neurale ricorrente che opera come funzione di espansione non lineare, mentre il *readout* produce l'output desiderato dall'espansione.

Una *Echo State Network* consiste in un livello di input di N_u unità, un insieme di N_r unità nascoste che compongono il suddetto *reservoir* e un livello di output di N_y unità tipicamente lineari e non ricorrenti che formano il *readout*. Le equazioni che descrivono la computazione sviluppata da una ESN sono:

$$\mathbf{x}(t) = f_x(\mathbf{u}(t), \mathbf{x}(t-1)) = f(\mathbf{W_{in}}u(t) + \mathbf{W}\mathbf{x}(t-1) + \mathbf{b}) \tag{3.1}$$

$$\mathbf{y}(t) = f_y(\mathbf{x}(t)) = \mathbf{W_{out}}\mathbf{x}(t) \tag{3.2}$$

La separazione concettuale che caratterizza le *echo state network* è basata sulla comprensione dei diversi scopi che assumono f_x e f_y:

- f_x espande la storia dell'input $\mathbf{u}(t), \mathbf{u}(t-1)$,... in uno spazio degli stati del *reservoir* $\mathbf{x}(t) \in \mathbb{R}^{N_x}$,
- mentre f_y combina i segnali ricevuti dai neuroni $\mathbf{x}(t)$ per ottenere gli output attesi $\hat{\mathbf{y}}(t)$.

La principale caratteristica del RC e quindi delle ESN è che la parte ricorrente della rete può non essere allenata dopo essere inizializzata secondo alcune proprietà facili da soddisfare. L'allenamento si riduce dunque alla sola parte non ricorrente, si riducono così i costi computazionali che gravavano sul design delle RNN. Di seguito in figura 3.1 viene illustrato il design e l'allenamento di una ESN.

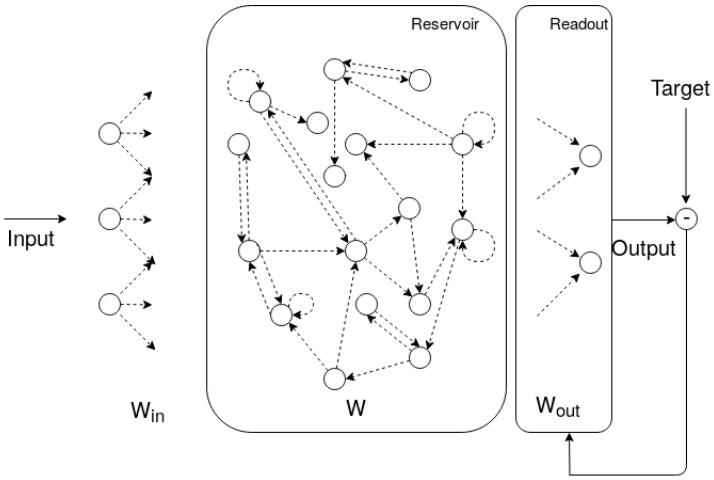


Figura 3.1: *Reservoir Computing* e allenamento del readout.

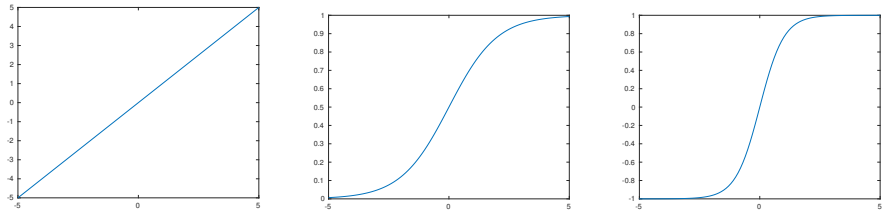
Ciò significa che $\mathbf{W_{in}}$ e $\mathbf{W}$ sono fisse e $\mathbf{W_{out}}$ varia in base all'allenamento. Tuttavia non tutte le scelte di $\mathbf{W_{in}}$ e $\mathbf{W}$ producono una ESN valida, il

reservoir deve essere un eco ("echo") della storia dell'input, cioè le dinamiche del *reservoir* devono dipendere asintoticamente solo dai segnali di input.

3.2 Le funzioni di attivazione

Le funzioni di attivazioni alle quali si fa riferimento nella formula 3.1 possono essere diverse, tra queste:

- la funzione identità : $f_{id}(x) = x$ (3.2a)
- la funzione logistica : $f_{logistic}(x) = \frac{1}{1+e^{-x}}$ (3.2b)
- la tangente iperbolica : $f_{tanh}(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ (3.2c)



(a) Funzione identità (b) Funzione logistica (c) Tangente iperbolica

Figura 3.2: Funzioni di attivazione comuni.

La funzione logistica e la tangente iperbolica sono le più usate, l'idea è che si può mappare ogni numero reale in un numero nell'intervallo $[0, 1]$ e rispettivamente in $[-1, 1]$, in questo modo si può dimostrare che una combinazione di queste funzioni può approssimare ogni funzione non lineare.

3.3 Leaking integration

Le unità nelle reti sigmoidali standard non hanno memoria; il loro valore ad un tempo $n + 1$ dipende solo parzialmente ed indirettamente dai valori

precedenti. Perciò queste reti si adattano al meglio per sistemi discreti nel tempo, d'altra parte è difficile l'apprendimento di dinamiche lente come onde sinusoidali molto lente. Per apprendere sistemi lenti e variabili in uno spazio continuo è più adeguato utilizzare delle reti con dinamiche continue. Per questo già in [3] viene introdotto un approccio ibrido che usa reti discrete nel tempo che approssimano delle reti continue grazie alle unità con *leaky integration*. In questo caso una semplificazione dell'equazione per il calcolo dello stato delle unità è la seguente:

$$\mathbf{x}(t) = (1-\alpha)\mathbf{x}(t-1) + \alpha f_x(\mathbf{u}(t), \mathbf{x}(t-1)) = \tag{3.3}$$
$$(1-\alpha)\mathbf{x}(t-1) + \alpha f(\mathbf{W_{in}} u(t) + \mathbf{W}\mathbf{x}(t-1) + \mathbf{b})$$

dove $\alpha \in (0, 1]$ è il *leaking decay rate*, se si imposta $\alpha = 1$ si ottiene una ESN standard.

3.4 Echo State Property

Una ESN valida soddisfa la cosiddetta *Echo State Property* (ESP) (Jager,2001). Questa dice che lo stato in cui la rete si trova dopo l'analisi di una lunga sequenza di input dipende solo dalla sequenza stessa. La dipendenza dallo stato iniziale della rete è progressivamente persa quando la lunghezza della sequenza di input tende all'infinito. Analogamente, lo stato corrente della rete $\mathbf{x}(t)$ è una funzione della storia dell'input indipendentemente dai valori iniziali dello stato. In formule la ESP può essere espressa come segue:

$$\forall s_n(\mathbf{u}) = [\mathbf{u}(1), ..., \mathbf{u}(n)] \in (\mathbb{R}^{N_u})^n \textit{sequenza di input di lunghezza } n,$$
$$\forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^{N_r} : \tag{3.4}$$
$$\|f_x(s_n(\mathbf{u}), \mathbf{x}) - f_x(s_n(\mathbf{u}), \mathbf{x}')\| \to 0 \textit{ per } n \to \infty$$

che significa che la differenza tra gli stati che la rete assume dopo che viene valutata una sequenza di input di lunghezza n tende a zero al tendere di n

ad infinito, per ogni scelta iniziale dello stato.
Jaeger ha fornito due condizioni, rispettivamente necessaria e sufficiente, affinchè una ESN goda di questa proprietà. La condizione necessaria è che il raggio spettrale, ovvero il maggiore degli autovalori in modulo, della matrice dei pesi del *reservoir* sia minore di uno.

$$\rho(\mathbf{W}) < 1 \tag{3.5}$$

Se questa condizione viene violata, il *reservoir* è localmente asintoticamente instabile nello stato $\mathbf{0} \in \mathbb{R}^{N_r}$ e la *echo state property* non può essere garantita se la sequenza nulla è un input ammissibile per il sistema.
La condizione sufficiente per la validità della proprietà è che il massimo valore singolare di $\mathbf{W}$ sia minore di uno:

$$\sigma(\mathbf{W}) < 1 \tag{3.6}$$

Ciò assicura la stabilità globale del sistema e la presenza di "stati echo", garantendo così la *echo state property*.
Una caratteristica molto importante evidenziata in [4] è la contrattività delle dinamiche dello stato.
La contratività della funzione di transizione dello stato f_x è una condizione sufficiente ma non necessaria per garantire la ESP.

Capitolo 4

Implementazione

Seguendo gli studi sviluppati in [5], vengono presentate le scelte progettuali fatte per l'implementazione e l'uso del *reservoir* e del *readout*.

4.1 Inizializzazione del Reservoir

Dato il modello delle RNN, il *reservoir* è definito dalla tupla $(\mathbf{W_{in}}, \mathbf{W}, \alpha)$. Le matrici dei pesi delle connessioni e dei pesi di input sono generate in modo random secondo dei parametri di cui si discute in seguito, e il *leaking rate* α è selezionato come parametro libero.
Analogamente a quanto viene fatto nella letteratura, quelli che per semplicità vengono chiamati parametri globali in questa sezione potrebbero essere chiamati iper-parametri, poiché non rappresentano direttamente i pesi delle connessioni ma regolano la loro distribuzione.
I parametri globali del *reservoir* sono: la dimensione N_r,la sparsità, ovvero la distribuzione degli elementi non nulli, e il raggio spettrale di $\mathbf{W}$; lo *scaling* di $\mathbf{W_{in}}$; ed il *leaking rate* α. I dettagli di ciascuno di queste scelte di design vengono illustrate in seguito. Per inizializzare un *reservoir* si invoca la funzione *weightMatrix()* con gli opportuni parametri.

```
function [win,w] = weightMatrix(varargin)

    %initialization of default parameters
    par.nr=100;          %reservoir units
    par.nu=1;            %input dim
    par.scale_in=0.1;    %input scaling
    par.rho=0.9;         %spectral radius
    par.alpha=1;         %leaking integrator
    par.dist='ud';       %type of distribution
    par.density_con=1;   %density of connections

    %assignement of values passed as parameters
    n_arg= length(varargin);
    for iArg = 1:2:n_arg        % considering couple of parameters
        name_argument = varargin{iArg};       % arguments's name
        value_argument = varargin{iArg+1};    % arguments's value
        par.(name_argument) = value_argument;
    end

    switch par.dist
            case 'ud'
                %input weignt matrix
                win= (-1 + (2*rand(par.nr, par.nu+1)))*par.scale_in;
                %connection wheight matrix
                aus= -1 + 2*rand(par.nr, par.nr);
            case 'nd'
                %input weignt matrix
                win= ((1/3)*randn(par.nr, par.nu+1))*par.scale_in;
                %connection wheight matrix
                aus= (1/3)*randn(par.nr, par.nr);
    end
    w= (1-par.alpha).*eye(par.nr) + par.alpha.*aus;
    w= w .* (par.rho/max(abs(eig(w))));
    w= (1/par.alpha) .* (w - (1-par.alpha).*eye(par.nr));
    %setting element to zero
    nrz= floor( (1-par.density_con)* par.nr);
    if nrz > 0
        for irow = 1:par.nr
             indz= randperm(par.nr,nrz);
             w(irow,indz) = 0;
        end
    end
end
```

code/weightMatrix.m

4.1.1 Dimensione del Reservoir

Un parametro fondamentale per il modello 3.1 è N_r, il numero delle unità del *reservoir*. In genere più è grande il *reservoir*, migliore è la performance ottenuta, tenendo conto delle tecniche di regolarizzazione appropriate per evitare l'*overfitting*. Poiché allenare ed applicare le ESN è computazionalmente economico rispetto alle altre reti neurali ricorrenti, non è insolito trovare dei reservoir di dimensioni dell'ordine di 10^4.

Più è grande lo spazio dei segnali del *reservoir* $\mathbf{x}(n)$, più facile è trovare una combinazione lineare dei segnali per approssimare $\mathbf{y}_{target}(n)$. Il *reservoir* può essere troppo grande quando il *task* è banale oppure quando non si hanno

molti dati disponibili, ad esempio $T < 1 + N_u + N_r$.
Generalmente nell'ambito accademico la dimensione del *reservoir* viene limitata per convenienza e per compatibilità di risultati; inoltre buoni parametri sono trasferibili a *reservoir* di dimensioni maggiori,quindi può essere conveniente selezionare i parametri globali con *reservoir* ridotti e poi trasferirli a quelli più grandi.
Un limite inferiore per la dimensione del *reservoir* può essere considerato il numero di valori reali che il reservoir deve ricordare dall'input per realizzare con successo il *task*. Il massimo numero di valori conservati in una ESN non può essere superiore di N_r. il parametro che indica la dimensione del *reservoir* passato alla funzione *weightMatrix()* è nr.

4.1.2 Sparsità del Reservoir

Nelle prime pubblicazioni riguardanti le ESN si raccomanda di realizzare *reservoir* con connessioni sparse, cioè con le matrici $\mathbf{W_{in}}$ e $\mathbf{W}$ aventi la maggior parte degli elementi uguali a zero. In generale questo parametro non influenza la performance così tanto ed ha una bassa priorità nell'ottimizzazione, come viene dimostrato anche in [4].
La sparsità fa si che l'aggiornamento della matrice sia più rapido, velocizzando quindi la computazione. Matrici densamente connesse potrebbero rallentare la computazione, soprattutto in base al numero di unità di attivazione considerate, quando la dimensione del *reservoir* è grande si può decidere di generare $\mathbf{W_{in}}$ e $\mathbf{W}$ secondo la stesso tipo di distribuzione ma una rispettivamente densa e sparsa. In particolare nell'implementazione fornita si può scegliere la distribuzione tra quella uniforme e quella normale associando, rispettivamente, le stringhe '*ud*' e '*nd*' al parametro *dist*. Inoltre si può esprimere la densità delle connessioni del *reservoir* attraverso il parametro *density_con*,il cui valore va da 0 a 1, dove con 1 si indica una matrice completamente connessa.

4.1.3 Raggio spettrale

Uno dei principali parametri delle ESN è il raggio spettrale della matrice dei pesi delle connessioni, cioè il massimo tra i valori assoluti degli autovalori della matrice. Questo parametro scala la matrice $\mathbf{W}$, o visto alternativamente scala l'ampiezza della distribuzione dei parametri diversi da zero.
Viene generata in modo random la matrice $\mathbf{W}$, viene calcolato il suo raggio spettrale $\rho(\mathbf{W})$, infine $\mathbf{W}$ viene divisa per $\rho(\mathbf{W})$ così da ottenere una matrice con raggio spettrale unitario da poter moltiplicare per il raggio spettrale desiderato. Si ricorda che questo ultimo deve essere minore di uno affinché valga la *Echo State Property*.
Come principio guida, $\rho(\mathbf{W})$ dovrebbe essere scelto grande per i *task* per i quali è richiesto una storia dell'input vasta , piccolo per i *task* il cui output corrente dipende dalla storia recente. Il valore del raggio spettrale desiderato è espresso dal parametro *rho*.

4.1.4 Input scaling

Lo *scaling* dei pesi della matrice di input è un altro valore chiave da ottimizzare in una Echo State Network. Per $\mathbf{W_{in}}$ generate con distribuzione uniforme generalmente ci si riferisce all'*input scaling* a come al modulo degli estremi dell'intervallo $[-a; a]$ dal quale vengono selezionati i valori di $\mathbf{W_{in}}$; per le matrici generate con distribuzione normale si prende la deviazione standard come misura di *scaling*.
In questa implementazione, per avere un numero esiguo di parametri liberi, tutte le colonne di $\mathbf{W_{in}}$ vengono scalate insieme in base ad un unico valore. In alternativa si può ottimizzare separatamente il valore di *input scaling* relativo alla colonna del *bias*, oppure si possono scalare le colonne di $\mathbf{W_{in}}$ separatamente in base a come queste contribuiscono al *task*. A seconda delle impostazioni il numero di parametri liberi per $\mathbf{W_{in}}$ varia da 1 a $N_u + 1$.
Nelle prime pubblicazioni venne suggerito di scalare e traslare i dati di input, ottimizzando queste operazioni. Lo stesso effetto può essere raggiunto sca-

lando i pesi di input e il *bias*, rispettivamente. Queste operazioni sono molto utili poiché permettono di avere dei valori limitati per i dati di input.
Per *task* molto lineari $\mathbf{W_{in}}$ dovrebbe essere piccolo, permettendo alle unità di operare intorno allo zero dove la funzione di attivazione *tanh*, funzione di attivazione scelta in questa implementazione, è quasi lineare. Par $\mathbf{W_{in}}$ con valori grandi , le unità si saturano facilmente vicino al loro valore -1 o 1, agendo in modo non lineare. È chiaro che l'*input scaling*, insieme al valore $\rho(\mathbf{W})$, determina quanto lo stato corrente $\mathbf{x}(t)$ dipende dall'input corrente $\mathbf{u}(t)$ e quanto dallo stato precedente $\mathbf{x}(t-1)$. Nell'implementazione il valore dell'*input scaling* viene indicato da *scale_in*.

4.1.5 Leaking Rate

La variabile *alpha* esprime il valore del *leaking rate* come visto in 3.3 ed indica la velocità con cui vengono aggiornate le dinamiche del *reservoir*. Sebbene questo parametro viene maggiormente utilizzato nel calcolo dello stato del *reservoir* viene presentato qui poiché si deve tener conto di questo valore nel momento in cui viene riscalata la matrice dei pesi $\mathbf{W}$ come dimostrato in [6].

4.2 Calcolo dello stato del Reservoir

Per calcolare lo stato del reservoir è necessario invocare la funzione, riportata sotto, *setReservoir()*. Questa prende come parametri le informazioni relative al *task* e le matrici che rappresentano il *reservoir*, inizializzate come espresso sopra. Può prendere in ingresso dei parametri opzionali che rappresentano rispettivamente il valore del *leaking rate* e dell'*input bias*. Viene applicata la 3.3 per il calcolo dello stato del *reservoir* e viene eliminato il numero di stati *transient* come espresso nella definizione del *task*.
Ciò significa che i valori iniziali $\mathbf{x}(n)$ non vengono considerati per l'allenamento del *readout* e vengono scartati in quanto influenzati dallo stato iniziale $\mathbf{x}(0)$, che tipicamente è $\mathbf{x}(0) = 0$. Questo introduce uno stato iniziale non

naturale difficile da raggiungere quando la rete comincia ad adattarsi al *task*. Il numero di *step* da saltare dipende dalla memoria della rete e tipicamente è dell'ordine di decine di centinaia. Tuttavia se il *task* è composto da molte sequenze corte il *transient* iniziale è la normale modalità di lavoro della ESN ed in questi casi eliminare gli stati iniziali può essere pericoloso.
Invece questo aspetto diventa fondamentale se si vogliono rendere indipendenti le sequenze, ad esempio quelle associate a *d*iversi.

```
function [X] = setReservoir(task,win,w,varargin)

    %initialization of default parameters
    par.nr=size(w,1);          %reservoir units
    par.alpha=1;          %leaking integrator
    par.bias=0;           %bias

    %assignement of values passed as parameters
    n_arg= length(varargin);
    for iArg = 1:2:n_arg         % considering couple of parameters
        name_argument = varargin{iArg};        % arguments's name
        value_argument = varargin{iArg+1};     % arguments's value
        par.(name_argument) = value_argument;
    end

    %training and test elements without transient for each fold
    training= task.training -(ceil(task.training/task.timesteps)*task.transient);
    design= task.design -(ceil(task.design/task.timesteps)*task.transient);
    test= task.test -(ceil(task.test/task.timesteps)*task.transient);
    % design + test element
    len= task.design+task.test;
    % reservoir states
    X= zeros(par.nr,task.kfold*(design+test));
     %target
    T= cell(1,task.readouts(end));

    % reservoir initialization
    aus=1;
    numseq=1;
    for i = 1:task.kfold
        taskin= task.in{i};
        x = zeros(par.nr,1);
        trans=task.transient;
        for n = 1:len
            u= taskin(:,n);
            x= (1-par.alpha)* x + par.alpha*tanh(win*[u;par.bias] + w*x );
            if n > task.timesteps*numseq
                numseq=numseq+1;
                trans=trans+task.timesteps;
            end
            %delete transient for each input sequence
            if n>trans
                %reservoir states
                X(:,aus)=x;
                aus=aus+1;
            end
        end
    end


end
```

code/setReservoir.m

4.3 Allenamento del Readout

Per questa implementazione si prende in considerazione un *readout* lineare, senza connessioni di *feedback*.
Un *readout* lineare può essere descritto con la seguente equazione

$$\mathbf{Y} = \mathbf{W_{out}}\mathbf{X} \tag{4.1}$$

dove:

- $\mathbf{Y} \in \mathbb{R}^{N_y * T}$ sono tutti gli $\mathbf{y}(n)$;
- $\mathbf{X} \in \mathbb{R}^{(1+N_r) * T}$ sono tutti gli $[\mathbf{x}(n); 1]$
- $\mathbf{W_{out}}$ sono i pesi ottimi che minimizzano l'errore tra $\mathbf{y}(n)$ e $\mathbf{y_{target}}(n)$

Nel caso in cui si volessero connettere i pesi di input al *readout* si avrebbe la matrice $\mathbf{X} \in \mathbb{R}^{(1+N_u+N_r) * T}$, formata da tutti i $[\mathbf{x}(n); \mathbf{u}(n); 1]$.
Trovare i pesi ottimi $\mathbf{W_{out}}$ che minimizzino l'errore tra $\mathbf{y}(n)$ e $\mathbf{y_{target}}(n)$ consiste nel risolvere un sistema di equazioni lineari del tipo:

$$\mathbf{Y_{target}} = \mathbf{W_{out}}\mathbf{X} \tag{4.2}$$

Per allenare il *readout* si deve invocare la funzione *train_readout*(), nelle sottosezioni successive vengono illustrati due approcci diversi per la risoluzione del sistema.

4.3.1 Regolarizzazione di Tikhonov

La più diffusa e stabile soluzione per 4.2 in questo contesto è la regolarizzazione di *Tikhonov* o *Ridge Regression*, in formula:

$$\mathbf{W_{out}} = \mathbf{Y_{target}}\mathbf{X^T}(\mathbf{X}\mathbf{X^T} + \beta\mathbf{I})^{-1} \tag{4.3}$$

dove β è il coefficiente di regolarizzazione.
Per misurare la qualità della soluzione prodotta dall'allenamento è consigliabile controllare i pesi ottenuti di $\mathbf{W_{out}}$, grandi pesi indicano che $\mathbf{W_{out}}$

amplifica piccole differenze tra le dimensioni di $\mathbf{x}(t)$ e può essere molto sensibile nelle situazioni diverse dalle esatte condizioni nelle quali la rete è stata allenata. Questo problema si accentua nel caso in cui la rete riceve il suo output come successivo input.
Per contrastare questi effetti viene introdotta la parte di regolarizzazione $\beta\mathbf{I}$. Per illustrare l'equazione risolta con la *ridge regression* viene considerato il RMSE (*root mean squared error*) come misura di errore:

$$\mathbf{W_{out}} = \arg\min_{\mathbf{W_{out}}} \frac{1}{N_y} \sum_{i=1}^{N_y} \left(\sum_{n=1}^{T} (y_i(n) - y_{i_{target}}(n))^2 + \beta \|w_{i_{out}}\|^2 \right) \tag{4.4}$$

con $w_{i_{out}}$ che indica la *i-esima* riga di $\mathbf{W_{out}}$ e $\| \cdot \|$ la norma euclidea. È evidente il compromesso che si ha tra avere un basso *training error* e pesi di output piccoli, ed è regolato proprio dal parametro di regolarizzazione β. Il coefficiente di regolarizzazione ottimo β dipende dall'istanza di $\mathbf{W}$, quindi è buona norma scegliere questo parametro attraverso la validazione.
Settare β uguale a *zero* rimuove la regolarizzazione: la funzione in 4.4 diventa uguale al semplice calcolo del RMSE(*root mean squared error*), rendendo la regolarizzazione di *Tikhononov* una generalizzazione della regressione lineare. La soluzione con $\beta = 0$ diventa:

$$\mathbf{W_{out}} = \mathbf{Y_{target}} \mathbf{X^T} (\mathbf{X X^T})^{-1} \tag{4.5}$$

Nella pratica, tuttavia, fissare $\beta = 0$ spesso causa instabilità numerica quando si inverte $(\mathbf{X X^T})$ in 4.5, è per questo che si raccomanda di usare per la selezione di β una scala logaritmica che non raggiunge lo *zero* o di usare la pseudoinversa invece che l'inversa come mostrato di seguito.

4.3.2 Soluzione con pseudoinversa

Una soluzione semplice per la risoluzione del sistema visto prima è:

$$\mathbf{W_{out}} = \mathbf{Y_{target}} \mathbf{X^+} \tag{4.6}$$

dove $\mathbf{X}^+$ è la pseudoinversa di *Moore-Penrose* di $\mathbf{X}$. Se $\mathbf{XX^T}$ è invertibile questa formula diventa uguale alla 4.5, ma funziona anche quando non lo è. Tuttavia, ha un elevato costo in memoria per grandi matrici $\mathbf{X}$, dunque si deve limitare la dimensione del *reservoir* e/o il numero di esempi di *training* T. Poichè non si ha la regolarizzazione il sistema di equazioni lineari 4.2 deve essere sovraddeterminato, cioè $1 + N_u + N_r << T$. In altre parole, il *task* deve essere difficile in relazione alla capacità del *reservoir* così che non avvenga l'*overfitting*.

In molte librerie viene fornita una implementazione della pseudoinversa, tuttavia ogni implementazione varia nella precisione, nell'efficienza computazionale e nella stabilità numerica. Per *task* di alta precisione, si deve controllare se la regressione $(\mathbf{Y_{target}} - \mathbf{W_{out}X})\mathbf{X}^+$ sull'errore $\mathbf{Y_{target}} - \mathbf{W_{out}X}$ è effettivamente uguale a *zero*, e aggiungerla a $\mathbf{W_{out}}$ nel caso non lo sia. Questo trucco computazionale non dovrebbe funzionare in teoria ma a volte funziona in partica in Matlab, probabilmente a causa di qualche ottimizzazione interna.

Capitolo 5

Test

In questo capitolo viene testata la rete neurale implementata attraverso il *task* NARMA e vengono confrontati i risultati ottenuti con quelli della letteratura.

5.1 10^{th} order NARMA system

Questo *task* consiste nella predizione di un sistema di 10^{th} ordine di media mobile autoregressivo non lineare. Il *task* è stato introdotto in Atiya e Palos (2000) ed è stato trattato con le ESN in Jaeger (2002) e in Cernanský e Tiňo (2008). L'input del sistema è una sequenza di elementi $\mathbf{u}(n)$ scelti secondo una distribuzione uniforme in $[0, 0.5]$. L'output del sistema è calcolato come:

$$\bar{\mathbf{y}}(n) = 0.3\bar{\mathbf{y}}(n-1) + 0.05\bar{\mathbf{y}}(n-1)\left(\sum_{i=1}^{10}\bar{\mathbf{y}}(n-i)\right) + 1.5\mathbf{u}(n-10)\mathbf{u}(n-1) + 0.1. \tag{5.1}$$

Dato il valore di input $\mathbf{u}(n)$, il *task* è di predire il corrispondente valore di $\bar{\mathbf{y}}(n)$. Il *training set* è formato da $N_{train} = 2200$ esempi *input-target*,dei quali $N_{transient} = 200$ sono di *transient* iniziale. Una sequenza di lunghezza $N_{test} = 2000$ viene usata per il test.
Il *task* viene generato attraverso la funzione *genetateTask()*, passando come argomento *Tasks.narma*, questa funzione permette di generare tutti i *task* che

sono elencati nell'enumerazione *Tasks*, questi ultimi vengono creati secondo dei criteri ben precisi in modo da poter fornire alla rete tutte le informazioni necessarie. Per verificare che un *task* abbia tutti i campi necessari per essere ben definito è utilizzata la funzione *isTask()*.

5.2 ESNtrain() ed ESNtest()

Per facilitare l'uso della rete sviluppata viene fornita una funzione chiamata *ESNtrain()*. La funzione si occupa dell'inizializzazione delle matrici dei pesi, del calcolo dello stato del *reservoir* e del *training* del *readout* invocando le funzioni illustrate nel capitolo 5. A questa funzione devono essere passati come argomenti un *task* valido e tutti i parametri che si intendono usare nella rete, qualora questi non fossero presenti vengono utilizzati dei parametri standard illustrati nella tabella sottostante.

parametri	**nome parametro**	**valore default**
input scaling	*scale_in*	0.1
unità reservoir	*nr*	100
tipo di distribuzione	*dist*	'ud'
densità connessione	*density_con*	1
raggio spettrale	*rho*	0.9
leaky integrator	*alpha*	1
input bias	*bias*	1
par regolazizzazione	*lambda*	0
misura errore	*error*	'mse'
risultati test	*test*	true

Tabella 5.1: Parametri default *ESNtrain()*

In particolare si ha il parametro *test* che se settato a *true* permette di ottenere anche il risultato di test,eseguito dopo il *training*. Se viene usata questa opzione bisogna far attenzione a non usare il valore di test per la scelta del modello, in questo caso infatti la *model selection* verrebbe compromessa.

Per il test della rete deve essere usata la funzione *ESNtest()*. Questa funzione prende in ingresso il *task* ed eventualmente altri parametri per la realizzazione della rete, infatti si può decidere se passare o meno come parametri un *reservoir* ed un *readout* allenato. Nel primo caso viene eseguita solo una fase di test, altrimenti viene inizializzata un'altra rete secondo i parametri passati come argomento; quest'ultima operazione assume un certo valore soprattutto nel caso in cui i parametri passati come argomento sono quelli di una *model selection* precedente.

5.3 Risultati

Per poter ottenere dei risultati confrontabili con quelli in [4] viene fatta una *model selection* al variare del parametro *rho*, ovvero del raggio spettrale. Nelle figure successive sono mostati in ordine i valori dell'errore di train, dell'errore di test ottenuto con la funzione *ESNtrain()*, di quello di test ottenuto con la funzione *EStest()* avente come parametri rispettivamente quelli ottenuti della *model selection* ed il *reservoir* ed il *readout* prodotti in precedenza.

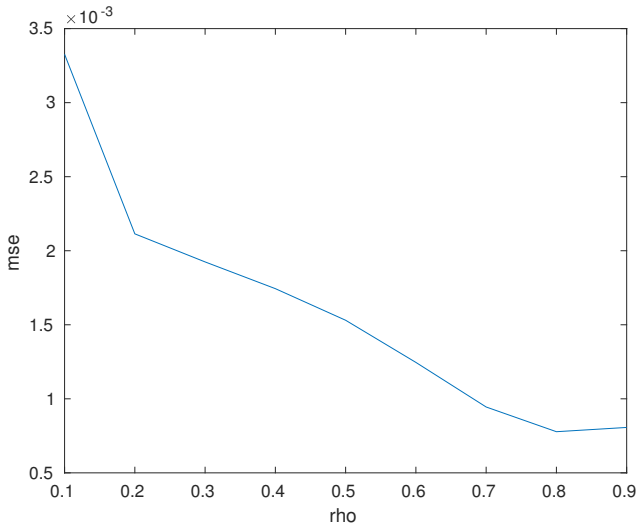


Figura 5.1: Training error generato da *ESNtrain()*.

L'errore di training è naturalmente più basso dell'errore di test che riflette comunque lo stesso andamento.

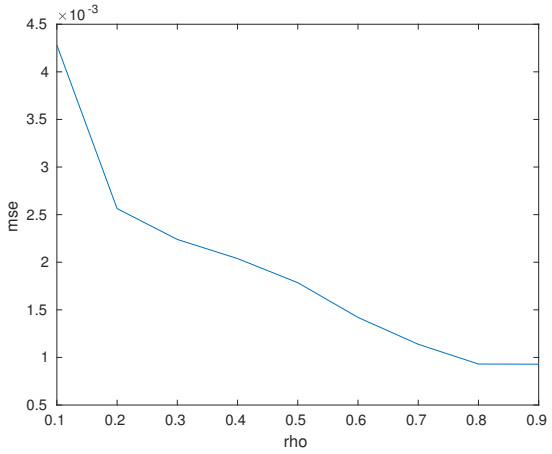


Figura 5.2: Test error generato da *ESNtrain()* dopo l'allenameto del readout.

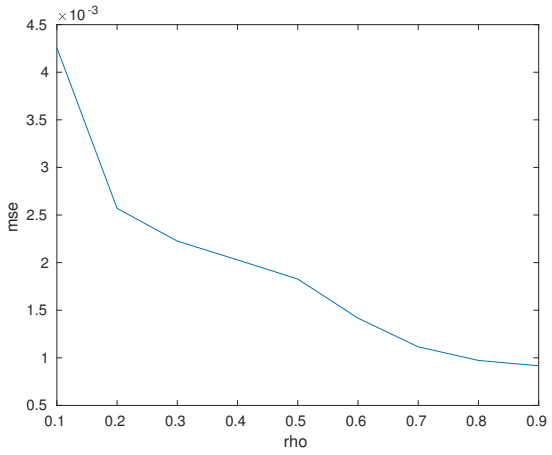


Figura 5.3: Test error generato da *ESNtest()* dopo una nuova inizializzazione di *reservoir*.

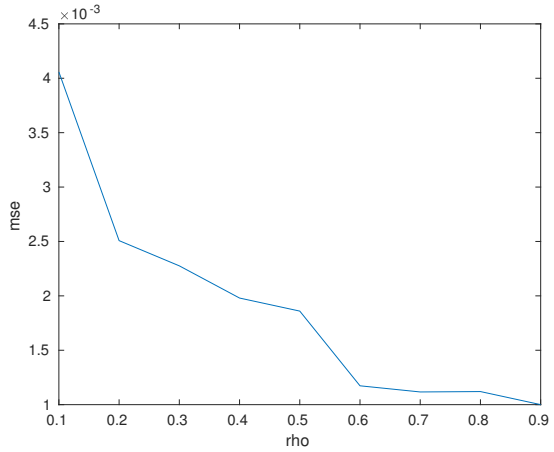


Figura 5.4: Test error generato da *ESNtest()* passando il readout già allenato.

Si può notare come l'errore diminuisca al crescere del valore del raggio spettrale, andamento aspettato nel caso di *task* non lineare come questo. Di seguito è mostrato il valore dell'*input scaling* al variare del raggio spettrale. Nel paper questo valore è fissato prima della model selection proprio perchè imporre l'*input scaling=0.1* risulta essere la scelta migliore.

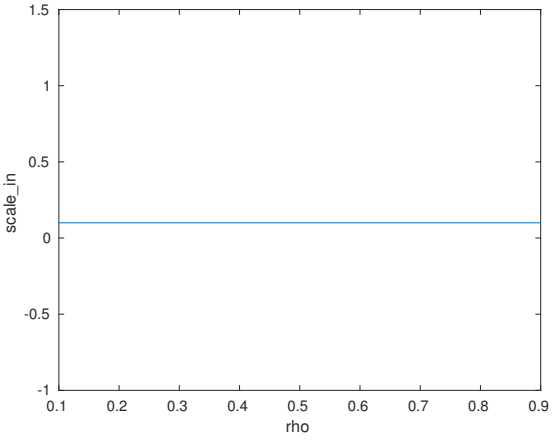


Figura 5.5: Miglior scelta *input scaling* al variare di rho.

Per avere un confronto preciso al livello numerico con il *paper* citato, viene allenta una rete neurale con 500 unità totalmente connessa.

Risultati paper	Risultati rete implementata
$3.1413 \times 10^{-4}(\pm 1.14197 \times 10^{-5})$	$3.2159 \times 10^{-4}(\pm 1.1701 \times 10^{-5})$

Tabella 5.2: Error test *ESNtrain()*

5.4 Conclusioni

I risultati ottenuti con la rete implementata dimostrano il corretto funzionamento di quest'ultima. La rete può dunque essere utilizzata per la computazione di altri *task*, come quelli inseriti nella funzione per la generazione degli stessi. Il modello dunque risulta utile per un primo approccio alle *Echo State Network* e come punto di partenza per la realizzazione di altri modelli. La sua modularità è stata pensata proprio per poter facilitare l'implementazione di reti con configurazioni diverse del *reservoir* per studi futuri.

Bibliografia

[1] Mantas Lukoševičius, Herbert Jaeger, Reservoir computing approaches to recurrent neural network training, Computer Science Review, Volume 3, Issue 3, 2009, Pages 127-149.

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

[3] Herbert Jaeger, The "echo state" approach to analysing and training recurrent neural networks, Technical Report GMD Report 148, German National Research Center for Information Technology, 2001.

[4] Claudio Gallicchio, Alessio Micheli, Architectural and Markovian factors of echo state networks, Neural Networks, Volume 24, Issue 5, 2011, Pages 440-456.

[5] Mantas Lukoševičius, A Practical Guide to Applying Echo State Networks. In: Montavon G., Orr G.B., Müller KR. (eds) Neural Networks: Tricks of the Trade. Lecture Notes in Computer Science, vol 7700, Springer, Berlin, Heidelberg, 2012.

[6] Herbert Jaeger, Mantas Lukoševičius, Dan Popovici, Udo Siewert, Optimization and applications of echo state networks with leaky- integrator neurons, Neural Networks, Volume 20, Issue 3, 2007, Pages 335-352.

Appendice A

Codice

```
function [X] = setReservoir(task,win,w,varargin)

    %initialization of default parameters
    par.nr=size(w,1);           %reservoir units
    par.alpha=1;          %leaking integrator
    par.bias=0;           %bias

    %assignement of values passed as parameters
    n_arg= length(varargin);
    for iArg = 1:2:n_arg          % considering couple of parameters
        name_argument = varargin{iArg};        % arguments's name
        value_argument = varargin{iArg+1};     % arguments's value
        par.(name_argument) = value_argument;
    end

    %training and test elements without transient for each fold
    training= task.training -(ceil(task.training/task.timesteps)*task.transient);
    design= task.design -(ceil(task.design/task.timesteps)*task.transient);
    test= task.test -(ceil(task.test/task.timesteps)*task.transient);
    % design + test element
    len= task.design+task.test;
    % reservoir states
    X= zeros(par.nr,task.kfold*(design+test));
     %target
    T= cell(1,task.readouts(end));

    % reservoir initialization
    aus=1;
    numseq=1;
    for i = 1:task.kfold
        taskin= task.in{i};
        x = zeros(par.nr,1);
        trans=task.transient;
        for n = 1:len
            u= taskin(:,n);
            x= (1-par.alpha)* x + par.alpha*tanh(win*[u;par.bias] + w*x );
            if n > task.timesteps*numseq
                numseq=numseq+1;
                trans=trans+task.timesteps;
            end
            %delete transient for each input sequence
            if n>trans
                %reservoir states
                X(:,aus)=x;
                aus=aus+1;
            end
        end
    end


end
```

code/setReservoir.m

```
function [T] = getTarget(task)

    istask(task);

    % design + test element
    len= task.design+task.test;
    %target
    T= cell(1,task.readouts(end));

    % target storage
    parfor r= task.readouts(1):1:task.readouts(end)
        aus= 1;
        numseq=1;
        for i= 1:task.kfold
            trans=task.transient;
            taskout= task.out{i,r};
            for n= 1:len
                if n > task.timesteps*numseq
                    trans=(task.timesteps*numseq)+task.transient;
                    numseq=numseq+1;
                end
                %delete transient for each target sequence
                if n>trans
                    T{r}(:,aus)= taskout(:,n);
                    aus=aus+1;
                end
            end
        end
    end

end
```

code/getTarget.m

```
function [wout, results] = train_readout(task,X,T,varargin)

     istask(task);

     par.nr= size(X,1);
     par.lambda= 0;
     par.error= 'mse';

    %assignement of values passed as parameters
    n_arg= length(varargin);
    for iArg = 1:2:n_arg          % considering couple of parameters
        name_argument = varargin{iArg};        % arguments's name
        value_argument = varargin{iArg+1};     % arguments's value
        par.(name_argument) = value_argument;
    end

    training= task.training -(ceil(task.training/task.timesteps)*task.transient);
    design= task.design -(ceil(task.design/task.timesteps)*task.transient);

    if task.kfold==1
        ausX= X(:,1:training);
        ausT= T(:,1:training);

        if par.lambda == 0
            wout= ausT*pinv(ausX);
        else
            ausX2= zeros(par.nr+1,size(ausX,2));
            for i= 1: size(ausX,2)
                ausX2(:,i)= [ausX(:,i);1];
            end
            ausX=ausX2;
            Xt= ausX';
            if det(ausX*Xt + par.lambda*eye(par.nr+1))==0
                wout= ausT*Xt*pinv(ausX*Xt + par.lambda*eye(par.nr+1));
            else
```

```
                wout= ausT*Xt/(ausX*Xt + par.lambda*eye(par.nr+1));
            end
        end
        %output rete
        if (training < design)
            ausX= X(:,training+1:design);
            ausT= T(:,training+1:design);
        end
        Y= wout*ausX;
        switch par.error
            case 'mse'
                results= immse(ausT,Y);
            case 'nmse'
                results= mean(mean((ausT-Y).^2) )/ mean(mean((ausT).^2));
            case 'rmse'
                results=abs( sqrt( mean(mean((ausT-Y).^2) ) ));
            case 'nrmse'
                results= sqrt(mean(mean(((ausT-Y).^2)./(max(max(ausT))-min(min(ausT)))))
    );

            case 'mc'
                c= (corrcoef(ausT,Y)).^2;
                results= c(1,2);
        end

    else
        res= zeros(task.kfold,1);
        parfor k= 1:task.kfold
            ausX=0;
            ausT=0;
            for t= 1:task.kfold
                if t~= k
                    if ausX==0
                        ausX= X(:, (design*(k-1)+1):(design*k));
                        ausT= T(:, (design*(k-1)+1):(design*k));
                    else
                        ausX= [ausX X(:, (design*(k-1)+1):(design*k))];
                        ausT= [ausT T(:, (design*(k-1)+1):(design*k))];
                    end
                end
            end

            if par.lambda == 0
                auswout{k}= ausT*pinv(ausX);
            else
                ausX2= zeros(par.nr+1,design);
                for i= 1: size(ausX,2)
                    ausX2(:,i)= [ausX(:,i);1];
                end
                ausX=ausX2;
                Xt=ausX2';
                if det(ausX2*Xt + par.lambda*eye(par.nr+1))==0
                    auswout{k}= ausT*Xt*pinv(ausX2*Xt + par.lambda*eye(par.nr+1));
                else
                    auswout{k}= ausT*Xt/(ausX2*Xt + par.lambda*eye(par.nr+1));
                end
            end

            Y{k}= auswout{k}*ausX;

            switch par.error
                case 'mse'
                    res(k) =immse(ausT,Y{k});
                case 'rmse'
                    res(k)=abs( sqrt( mean(mean((ausT-Y{k}).^2) ) ));
                case 'nrmse'
                     res(k)= sqrt(mean(mean(((ausT-Y{k}).^2)./(max(max(ausT))-min(min(
    ausT))))));
                case 'mc'
                    c= (corrcoef(ausT,Y{k})).^2;
                    res(k)= c(1,2);
            end
        end
        %retutn readout with best result
        switch par.error
            case 'mc'
                [~, idx]=max(res);
```

```
                wout= auswout{idx};
            otherwise
                [~, idx]=min(res);
                wout= auswout{idx};
        end
        results=0;
        for k= 1:task.kfold
            results= results + res(k);
        end
        results= results/ task.kfold;
    end
end
```

code/train_readout.m

```
function [results_test] = test_readout(task,wout,X,T, varargin)

    istask(task);

    par.nr= size(X,1);
    par.lambda= 0;
    par.error= 'mse';

    %assignement of values passed as parameters
    n_arg= length(varargin);
    for iArg = 1:2:n_arg        % considering couple of parameters
        name_argument = varargin{iArg};       % arguments's name
        value_argument = varargin{iArg+1};    % arguments's value
        par.(name_argument) = value_argument;
    end

    design= task.design -(ceil(task.design/task.timesteps)*task.transient);
    test= task.test -(ceil(task.test/task.timesteps)*task.transient);

    if task.kfold==1

        ausX= X(:,design+1:design+test);

        if size(wout,2)~= size(ausX,1)
            for i= 1: size(ausX,2)
                ausX2(:,i)= [ausX(:,i);1];
            end
            ausX=ausX2;
        end

        Ytest= wout*ausX;
        ausT= T(:,design+1:design+test);

        switch par.error
            case 'mse'
                results_test= immse(ausT,Ytest);
            case 'rmse'
                results_test=abs( sqrt( mean(mean((ausT-Ytest).^2) ) ));
            case 'nrmse'
                 results_test= sqrt(mean(mean(((ausT-Ytest).^2)./(max(max(ausT))-min(min
(ausT))))));
            case 'mc'
                c=(corrcoef(ausT,Ytest)).^2;
                results_test= c(1,2);
        end
    else
        res = zeros(task.kfold, 1);
        for k= 1:task.kfold
            ausX= X(:, (design*(k-1)+1):(design*k));
            if size(wout,2)~= size(ausX,1)
                for i= 1: size(ausX,2)
                    ausX2(:,i)= [ausX(:,i);1];
                end
                ausX=ausX2;
            end
            Ytest= wout*ausX;
            ausT= T(:,(design*(k-1)+1):(design*k));
            switch par.error
                case 'mse'
                    res(k,1) =immse(ausT,Y);
```

```
                case 'rmse'
                    res(k,1)=abs( sqrt( mean(mean((ausT-Ytest).^2) ) ));
                case 'nrmse'
                    res(k,1)= abs( sqrt( sum(mean((ausT-Ytest).^2)) ) )/(max(max(ausT))-
    min(min(ausT)));
                case 'mc'
                    c=(corrcoef(ausT,Ytest)).^2;
                    res(k,1)=c(1,2);
            end
        end

        results_test=0;
        for kf= 1:task.kfold
            results_test= results_test+ res(kf,1);
        end
        results_test= results_test./ task.kfold;
    end

end
```

code/test_readout.m

```
classdef Tasks
    enumeration
        Simple, Narma, Verst, MC, Laser, MG, sinMC, Mixture
    end
end
```

code/Tasks.m

```
function [] = istask(task)
    if ( ~isfield(task, 'dim_in') || ~isfield(task, 'dim_out') || ~isfield(task, 'kfold'
    ) ||...
            ~isfield(task, 'readouts') || ~isfield(task, 'timesteps') || ~isfield(task,
    'design') ||...
            ~isfield(task, 'transient') || ~isfield(task, 'training') ||~isfield(task, '
    test'))
        error('Task is not well defined, use function generateTask()');
    end
end
```

code/istask.m

```
function task = generateTask(taskname,varargin)
%Generazione dei task
% task.size : dimensione del task
% task.nrreadout : nr di readout da allenare
% task.kfold : numero di folds

switch taskname
    case Tasks.Simple
        %inizializzazione default per task casuale
        task.dim_in= 3;
        task.dim_out= 2;
        task.kfold=1;
        task.readouts=1;
        task.timesteps=4200;
        task.design= 2200;
        task.transient=200;
        task.training=2000;
        task.test=2000;
        %inizializzazione con parametri passati come argomento
        n_arg= length(varargin);
        for iArg = 1:2:n_arg % considerare i parametri in coppie
            name_argument = varargin{iArg}; % nome dell'argomento
            value_argument = varargin{iArg+1}; % valore dell'argomento
            task.(name_argument) = value_argument;
        end


```

```
28          for k = 1:task.kfold
29              for i= 1: (task.design+task.test)
30                  for j = 1:task.dim_in
31                      task.in{k}(j,i) =-1 + 2*rand();
32                  end
33                  for j = 1 :task.dim_out
34                      task.out{k,1}(j,i) =-1 + 2*rand();
35                  end
36              end
37          end
38
39      case Tasks.Narma
40          %inizializzazione default per Narma task
41          task.dim_in= 1;
42          task.dim_out= 1;
43          task.kfold=1;
44          task.readouts=1:1;
45          task.timesteps=4200;
46          task.design= 2200;
47          task.transient=200;
48          task.training=2200;
49          task.test=2000;
50
51          %inizializzazione con parametri passati come argomento
52          n_arg= length(varargin);
53          for iArg = 1:2:n_arg % considerare i parametri in coppie
54              name_argument = varargin{iArg}; % nome dell'argomento
55              value_argument = varargin{iArg+1}; % valore dell'argomento
56              task.(name_argument) = value_argument;
57          end
58
59          if task.dim_in ~= 1 || task.dim_out ~= 1
60             disp( "Error: dim_in and dim_out must be 1 in narma task.");
61          else
62            for k = 1:task.kfold
63                task.in{k}= abs(rand(1,task.design+task.test) -0.5);
64                task.out{k,1}= abs(rand(1,task.design+task.test) -0.5);
65                for n= 11:task.design+task.test
66                    task.out{k,1}(1,n)= 0.3*task.out{k,1}(1,n-1) + 0.05*task.out{k,1}(1,n
    -1)*(task.out{k,1}(1,n-1)+task.out{k,1}(1,n-2)+task.out{k,1}(1,n-3)+task.out{k,1}(1,
    n-4)+task.out{k,1}(1,n-5)+task.out{k,1}(1,n-6)+task.out{k,1}(1,n-7)+task.out{k,1}(1,
    n-8)+task.out{k,1}(1,n-9)+task.out{k,1}(1,n-10))+ 1.5*task.in{k}(1,n-10)*task.in{k
    }(1,n-1) + 0.1;
67              end
68            end
69          end
70
71      case Tasks.Verst
72          task.dim_in= 1;
73          task.dim_out= 1;
74          task.kfold=5;
75          task.readouts=1:150;
76          task.timesteps=1000;
77          task.design= 2000;
78          task.training= 2000;
79          task.transient=200;
80          task.test=0;
81          task.p=10;
82          task.d=15;
83          n_arg= length(varargin);
84          for iArg = 1:2:n_arg % considerare i parametri in coppie
85              name_argument = varargin{iArg}; % nome dell'argomento
86              value_argument = varargin{iArg+1}; % valore dell'argomento
87              task.(name_argument) = value_argument;
88          end
89          aus=0;
90
91          for k= 1:task.kfold
92              inaus{k}=zeros(1,task.d+2+task.timesteps);
93              inbus{k}=zeros(1,task.d+2+task.timesteps);
94              for n = 1:1:task.d+1+task.timesteps
95                  inaus{k}(n)= -0.8 + 1.6*rand();
96                  inbus{k}(n)= -0.8 + 1.6*rand();
97              end
98              task.in{k}(1,1:task.timesteps)=inaus{k}(1, task.d+2:task.d+1001);
99              task.in{k}(1,task.timesteps+1:2*task.timesteps)=inbus{k}(1, task.d+2:task.d
    +1001);
```

```
        end
        for d = 1:1:task.d
             for p =1:1:task.p
                  aus=aus+1;
                  for k= 1:task.kfold
                       kt=task.d+2;
                       for j= 1:task.timesteps
                            task.out{k,aus}(1,j)=sign(inaus{k}(kt-d)*inaus{k}(kt-d-1))*(
    abs(inaus{k}(kt-d)*inaus{k}(kt-d-1))).^p;
                            task.out{k,aus}(1,j+task.timesteps)=sign(inbus{k}(kt-d)*inbus{k
    }(kt-d-1))*( abs(inbus{k}(kt-d)*inbus{k}(kt-d-1))).^p;
                       kt=kt+1;
                       end
                  end
             end
        end

    case Tasks.MC
        task.dim_in=1;
        task.dim_out=200;
        task.kfold=1;
        task.readouts=1:200;
        task.timesteps=6000;
        task.design= 5000;
        task.transient=200;
        task.training=5000;
        task.test=1000;

        D= importdata('mc100.csv');
        task.in{1}= (D.data(1:6000, 1))';
        for r=1:task.readouts(end)
            task.out{1,r}=(D.data(1:6000, r+1))';
        end
    case Tasks.Laser
        task.dim_in=1;
        task.dim_out=1;
        task.kfold=1;
        task.readouts=1;
        task.timesteps=10092;
        task.design= 5000;
        task.transient=1000;
        task.training=4000;
        task.test=5092;

        D= importdata('laser.csv');
        disp(D);
        task.in{1}= (cellfun(@str2num, D.textdata(17:10108, 1)))';
        for r=1:task.readouts
            task.out{1,r}=cellfun(@str2num,D.textdata(17:10108, r+1))';
        end
    case Tasks.MG
        task.dim_in=1;
        task.dim_out=1;
        task.kfold=1;
        task.readouts=1;
        task.timesteps=10000;
        task.design= 5000;
        task.transient=1000;
        task.training=5000;
        task.test=5000;

        data= load('MG17_task.mat');

        task.in{1}=data.input;
        for r=1:task.readouts
            task.out{1,r}=data.target;
        end

    case Tasks.sinMC
        task.dim_in=1;
        task.dim_out=200;
        task.kfold=1;
        task.readouts=1:16;
        task.timesteps=6000;
        task.design= 5000;
        task.transient=200;
        task.training=5000;
```

```
        task.test=1000;
        task.v=exp(-1);

        n_arg= length(varargin);
        for iArg = 1:2:n_arg
            name_argument = varargin{iArg};
            value_argument = varargin{iArg+1};
            task.(name_argument) = value_argument;
        end

        D= importdata('mc100.csv');
        task.in{1}= (D.data(1:6000, 1))';
        for r=1:task.readouts(end)
            task.out{1,r}=sin(task.v*(D.data(1:6000, r+1))');
        end
    case Tasks.Mixture
        task.dim_in=1;
        task.dim_out=1;
        task.kfold=1;
        task.readouts=1:1;
        task.timesteps=4200;
        task.design= 2200;
        task.transient=200;
        task.training=2200;
        task.test=2000;
        task.alpha=0;
        task.delay=13;

        n_arg= length(varargin);
        for iArg = 1:2:n_arg
            name_argument = varargin{iArg};
            value_argument = varargin{iArg+1};
            task.(name_argument) = value_argument;
        end

        if task.dim_in ~= 1 || task.dim_out ~= 1
           disp( "Error: dim_in and dim_out must be 1 in narma task.");
        else
          for k = 1:task.kfold

              inaus=abs(rand(1,task.design+task.test + task.delay) -0.5);
              task.in{k}= inaus(1,(task.delay+1):(task.design+task.test+task.delay));
              outaus=zeros(1,task.design+task.test + task.delay);
              outausmem=zeros(1,task.design+task.test + task.delay);

              for n= task.delay+1:task.design+task.test+task.delay
                  outaus(1,n)= 0.3*outaus(1,n-1)+ 0.05* outaus(1,n-1)*(outaus(1,n-1)+
    outaus(1,n-2)+outaus(1,n-3)+outaus(1,n-4)+outaus(1,n-5)+outaus(1,n-6)+outaus(1,n-7)+
    outaus(1,n-8)+outaus(1,n-9)+outaus(1,n-10))+ 1.5*inaus(1,n-10)*inaus(1,n-1) + 0.1;
                  outausmem(1,n)= inaus(1,n-task.delay);
                  task.out{k,1}(1,n-task.delay)= (1-task.alpha)*(outausmem(1,n))+ (task.
    alpha)*(outaus(1,n));
                  outaus(1,n)=task.out{k,1}(1,n-task.delay);
              end
          end
        end

    case default
        disp("type of task is unknown");

end
end
```

code/generateTask.m

```
function [results, results_test, wout, X] = ESNtrain(task,varargin)
%   initialization and training of a Echo State Network.
%   nu: input dim,              default 1
%   scale_in: input scaling,    default 0.1
%   nr: reservoir dimention,    default 100
%   dist: type of distribution, default ud
%   density_con: weights~= null,defaut 1
%   rho: spectra radius w,      default 0.9
%   alpha: leaking rate,        default 1
%   bias: input bias,           default 1
```

```
%   ny: output dimention,       default 1
%   readouts: number of readout,default 1
%   lambda: par reg tikhonov,   default 0
%   error: error mesure,        default mse


    par.scale_in = 0.1;
    par.nr = 100;
    par.dist= 'ud';
    par.density_con=1;
    par.rho= 0.9;
    par.alpha = 1;
    par.bias= 1;
    par.lambda = 0;
    par.error= 'mse';
    par.test= true;

    nu=task.dim_in;
    ny=task.dim_out;
    readouts=task.readouts(end);

    %assignement of values passed as parameters
    n_arg= length(varargin);
    for iArg = 1:2:n_arg    % considering couple of parameters
        name_argument = varargin{iArg}; % arguments's name
        value_argument = varargin{iArg+1}; % arguments's value
        par.(name_argument) = value_argument;
    end

    [win, w]= weightMatrix('nr', par.nr, 'nu', nu, 'scale_in', par.scale_in, 'rho', par.
    rho, 'alpha', par.alpha, 'dist', par.dist, 'densit_con', par.density_con );

    %readout weight
    wout= cell(1,readouts);
    %output readout
    Y=cell(task.kfold,readouts);
    %results for each readout
    results=zeros(1, readouts);

    if task.dim_in ~= nu || task.dim_out ~= ny
        disp('Dimentions of this task do not match with nu and ny,');
        disp('Initialize a new ESN with the right parameterd');
    end

    [ X ]= setReservoir(task, win, w, 'alpha', par.alpha, 'bias', par.bias );
    [ T ]= getTarget(task);
    parfor r= task.readouts(1):1:task.readouts(end)
        [wout{1,r}, results(1,r)]= train_readout(task,X,T{r},'lambda', par.lambda,'error
    ',par.error);
        results_test(1,r)=0;
    end

    if (par.test)
        parfor r= task.readouts(1):1:task.readouts(end)
            [results_test(1,r)]= test_readout(task,wout{1,r},X,T{r},'lambda',par.lambda,
    'error', par.error);
        end
    end
end
```

code/ESNtrain.m

```
function [results] = ESNtest(task,varargin)
%   initialization and training of a Echo State Network.
%   nu: input dim,              default 1
%   scale_in: input scaling,    default 0.1
%   nr: reservoir dimention,    default 100
%   dist: type of distribution, default ud
%   rho: spectra radius w,      default 0.9
%   alpha: leaking rate,        default 1
%   bias: input bias,           default 1
%   ny: output dimention,       default 1
%   readouts: number of readout,default 1
%   lambda: p reg tikhonov,   default 0
%   error: error mesure,        default mse
```

```
14
15
16      p.scale_in = 0.1;
17      p.nr = 100;
18      p.dist= 'ud';
19      p.density_con=1;
20      p.rho= 0.9;
21      p.alpha = 1;
22      p.bias= 1;
23      p.lambda = 0;
24      p.error= 'mse';
25      p.wout={};
26      p.X={};
27
28      nu=task.dim_in;
29      ny=task.dim_out;
30      readouts=task.readouts(end);
31
32      %assignement of values passed as pameters
33      n_arg= length(varargin);
34      for iArg = 1:2:n_arg    % considering couple of pameters
35          name_argument = varargin{iArg}; % arguments's name
36          value_argument = varargin{iArg+1}; % arguments's value
37          p.(name_argument) = value_argument;
38      end
39
40      T= getTarget(task);
41
42      %if readout are not passed as argument, they are produced and trained
43      %on both training and validation data.
44
45      if ((iscell(p.wout) && isempty(cell2mat(p.wout))) || (~iscell(p.wout) && isempty(p.
        wout)) ||...
46              (iscell(p.X) && isempty(cell2mat(p.X))) || (~iscell(p.X) && isempty(p.X)) )
47          [win, w]= weightMatrix('nr', p.nr, 'nu', nu, 'scale_in', p.scale_in, 'rho', p.
        rho, 'alpha', p.alpha, 'dist', p.dist, 'densit_con', p.density_con );
48
49          %readout weight
50          wout= cell(1,readouts);
51          %output readout
52          Y=cell(task.kfold,readouts);
53          %results for each readout
54          results=zeros(1, readouts);
55
56          if task.dim_in ~= nu || task.dim_out ~= ny
57              disp('Dimentions of this task do not match with nu and ny,');
58              disp('Initialize a new ESN with the right pameterd');
59          end
60
61          task.training=task.design;
62
63          [ X ]= setReservoir(task, win, w, 'alpha', p.alpha, 'bias', p.bias );
64          [ T ]= getTarget(task);
65          parfor r= task.readouts(1):1:task.readouts(end)
66              [wout{1,r}, results(1,r)]= train_readout(task,X,T{r},'lambda', p.lambda,'
        error',p.error);
67
68          end
69          p.wout=wout;
70          p.X=X;
71      end
72
73      parfor r= task.readouts(1):1:task.readouts(end)
74          if(iscell(p.wout))
75              [results(1,r)]= test_readout(task,p.wout{1,r},p.X,T{r},'lambda',p.lambda,'
        error', p.error);
76          else
77              [results(1,r)]= test_readout(task,p.wout,p.X,T{r},'lambda',p.lambda,'error',
         p.error);
78          end
79      end
80
81  end
```

code/ESNtest.m