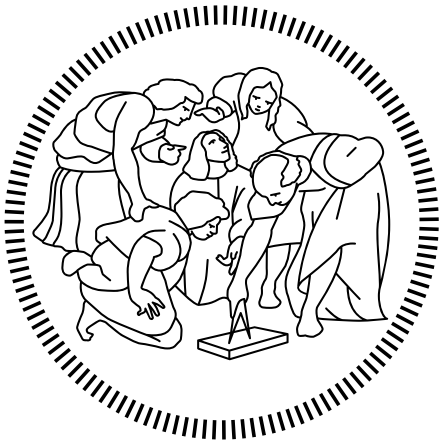
POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informatica e Bioingegneria
Ingegneria Informatica



Apprendimento di Movimenti Ritmici in Robot a Zampe con Reti Neurali Caotiche

Relatore:

Giuseppina Gini

Tesi di Laurea Magistrale di:

Matteo Bana
Matricola 816636

Anno Accademico 2014-2015

Sommario

I robot dotati di zampe hanno ricevuto molta attenzione negli ultimi anni, perché offrono una tecnologia promettente per varie applicazioni in ambienti dove i robot con ruote hanno difficoltà. Il problema della locomozione a zampe, però, è solo parzialmente risolto: attualmente questo tipo di robot cammina, ma non in modo particolarmente agile.

Per risolvere questo problema, quindi, è abbastanza naturale ispirarsi alle soluzioni adottate dagli animali: dopotutto la natura ha avuto milioni di anni per adattare al meglio i movimenti. Questo lavoro si inserisce nel contesto della robotica mobile bio-ispirata. Lo scopo è lo studio dei *Central Pattern Generator* (abbreviati con *CPG*) e l'introduzione delle *Reti Neurali Caotiche* (*CNN*, acronimo dall'inglese) al fine di controllare l'andatura di un robot con zampe.

Tra questi due modelli, i Central Pattern Generator sono già stati usati per controllare robot, principalmente quadrupedi, ma esistono anche alcune applicazioni per bipedi. Invece, le reti neurali caotiche sono state considerate più come metodi di machine learning. L'obiettivo di questa tesi è quindi il verificare quale di questi due metodi sia più adatto al movimento di robot con zampe.

È stata effettuata una analisi di entrambi i metodi al fine di illustrare le caratteristiche principali. Inoltre sono state definite alcune euristiche per trovare dei parametri corretti per le CNN. È seguito uno studio più dettagliato dei costi computazionali. Per poter simulare un controllore per robot con zampe si è sviluppata una nuova libreria per la simulazione delle reti neurali caotiche, la cui implementazione è descritta in dettaglio. In seguito sono descritte delle applicazioni sia dei CPG sia delle CNN alla locomozione: inizialmente di robot quadrupedi e bipedi, e in seguito per approssimare dati acquisiti da soggetti umani.

Infine, dopo alcuni esperimenti in ambiente simulato, sono discusse le caratteristiche principali dei due metodi, cercando di evidenziare le scelte da effettuare per preferire uno dei due.

Il nuovo metodo proposto si è rivelato efficace, nonostante alcuni minori problemi che derivano sia dalla novità del modello utilizzato sia dalle caratteristiche dei sistemi robotici considerati. Questi problemi saranno sicuramente risolti nel futuro prossimo, con l'obiettivo principale di sviluppare un controllore per robot a zampe basato su reti neurali caotiche.

Abstract

Legged robots received an increased attention these past years because they are a promising technology for many applications in environments where wheeled robots are not suited. However, locomotion for legged robots is only a partially solved problem: these robots walk, but they don't reach the same level of performances that are found in animals.

Therefore, to solve in a meaningful way this problem, an easy source of inspiration is found in the solutions adopted by animals: nature has had millions of years to improve their movements. In this work we present a new method for bioinspired mobile robotics. The aim is the study of *Central Pattern Generators* (*CPG*) and the introduction of *Chaotic Neural Networks* (*CNN*) to control the gait of a legged robot.

Circuits inspired by Central Pattern Generators were already used to control robots, especially quadrupeds, but a few applications for bipedal robots also exist. Instead, chaotic neural networks were considered mainly as a machine learning method. The objective of this work is to audit which one of these two methods is the most suitable to control the locomotion of legged robots.

In order to explain the features of both methods, a theoretical analysis was made. Moreover some heuristic to find the values for the parameters of a CNN was defined. A detailed study of the computational costs followed. To simulate a controller for legged robots a new simulator for CNNs was developed, whose implementation is detailed. Afterwards, applications of both CPGs and CNNs to locomotion are described: at first focused on quadruped and bipedal robots and then on data acquired from human subjects.

Lastly, a discussion of the features of the two methods is presented. In this section the choices to make when deciding between them are then highlighted.

The proposed method was found to be effective. However, it suffers from some minor problems that can be attributed to both the novelty of the model and the features of the considered robots. These problems will surely be solved in the near future, with the main objective to develop a working controller for legged locomotion based on CNN.

Ringraziamenti

Desidero innanzitutto ringraziare la Professoressa Gini, relatrice di questa tesi, per la disponibilità e il tempo dedicatomi nel correggere le numerose bozze di questo lavoro. Ringrazio sentitamente anche l'ingegner Franchi per il prezioso aiuto nel laboratorio di robotica e il professor Folgheraiter per i consigli su come procedere.

Ringrazio anche i compagni di corso, senza di voi questi 5 anni sarebbero stati sicuramente più lunghi.

Infine, ringrazio la mia famiglia per il sostegno e l'aiuto che mi hanno dato in questi anni.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

Lo scopo principale di un agente autonomo è raggiungere un obiettivo all'interno di un ambiente sconosciuto. È quindi un presupposto naturale che l'agente sia in grado di muoversi. In particolare, la locomozione di robot può essere divisa in molti tipi, ma i più comuni per applicazioni sulla terraferma sono il movimento con ruote, con zampe o strisciando.

I robot che strisciano sono adatti principalmente per scopi ricognitivi in ambienti ristretti, quindi possono risolvere una classe di problemi più piccola rispetto agli altri tipi di robot mobili. I robot dotati di ruote hanno il vantaggio di essere efficienti dal punto di vista energetico e hanno un sistema di controllo relativamente semplice, visto che non si deve occupare dell'equilibrio. Però sono adatti a muoversi su terreni solo con lievi asperità. D'altra parte, un robot con zampe è adatto a dei terreni più irregolari, siccome può superare alcuni piccoli ostacoli semplicemente scavalcandoli. Da un punto di vista pratico, quindi, i robot con gambe offrono una tecnologia promettente per varie applicazioni in cui l'utilizzo di robot con ruote è limitato.

La maggior parte degli ambienti in cui si può muovere un robot, inoltre, sono progettati o modificati per essere ottimali per gli esseri umani. Quindi ci si può aspettare che in questi luoghi un robot con caratteristiche antropomorfe si possa muovere più agevolmente. Oltre al punto di vista prettamente pratico, c'è anche la visione più "futuristica", legata ovvero all'interazione uomo-robot: gli esseri umani tendono ad interagire meglio con altri esseri umani, quindi un robot bipede che assomigli ad un uomo è molto probabilmente più adatto a scopi sociali rispetto ad una macchina dotata di ruote (alcuni risultati che supportano questa idea — forse un po' fantascientifica — sono riportati in [50]).

Alcune applicazioni per questo tipo di robot possono essere le missioni di esplorazione (e salvataggio) di ambienti in cui l'intervento umano è difficile (per

esempio, dopo un terremoto) o addirittura impossibile (in siti radioattivi). Inoltre, naturalmente, comprendere i meccanismi che generano il moto delle gambe può aiutare lo sviluppo di nuovi metodi per la riabilitazione di persone che hanno perso l'uso delle gambe in seguito ad incidenti.

Ovviamente i problemi relativi alla locomozione coprono vari aspetti sia di meccanica sia di controllo. Per esempio, qual è un buon design meccanico per un robot bipede? Quanti gradi di libertà dovrebbe avere e come dovrebbero essere controllati? Le risposte a queste domande impongono dei vincoli che devono essere poi rispettati sia dal progettista meccanico sia dal designer del sistema di controllo. Un caso tipico riguarda la potenza computazionale e l'energia necessaria per alimentare i motori e i calcolatori: sia l'unità d'elaborazione sia le batterie devono essere all'interno del robot, e quindi sono limitate dalle dimensioni di esso, che a loro volta dipendono dall'applicazione per cui questo robot è stato pensato. Inoltre ci sono anche dei vincoli di tipo real-time perché non si può interrompere l'azione delle forze esterne, e un robot in caduta deve agire in maniera estremamente veloce.

Siccome attualmente gli animali sono migliori dei robot a muoversi, ispirarsi alla biologica è piuttosto naturale. Ovviamente non è una cosa nuova, nemmeno nel campo dell'informatica: esistono diversi metodi con una radice nelle soluzioni "naturali": gli algoritmi genetici o l'ottimizzazione ad ant-colony sono due esempi di tecniche ispirate dalle soluzioni provviste dalla natura che vedono un utilizzo piuttosto diffuso.

Al momento lo scopo di questo lavoro è a "senso unico": si prende ispirazione dalla biologia per risolvere i problemi di robotica, ma si può procedere anche al contrario, ovvero utilizzare la robotica per comprendere meglio il regno animale. Questa seconda strada non è così interessante dal punto di vista ingegneristico, perché non tiene in conto l'efficienza (di qualsivoglia tipo), ma ha lo scopo di replicare in maniera precisa i processi biologici. Visto che in questo lavoro si è interessati ad un problema molto pratico, si considera una soluzione interessante se produce risultati validi. Naturalmente non è necessario che queste soluzioni siano migliori delle metodologie esistenti in una fase ancora primitiva (come può essere questa) della loro progettazione, ma devono fornire un buon punto di partenza per eventuali miglioramenti, in modo tale che, sul lungo periodo, sia effettivamente possibile metterle in pratica.

1.1 Robotica mobile

Questa tesi si riferisce principalmente alla locomozione con zampe. Attualmente esistono diversi modelli di robot quadrupedi. Il più famoso è probabilmente BigDog, un robot quadrupede sviluppato dalla DARPA per scopi militari, per trasportare carichi su terreni troppo accidentati per veicoli convenzionali. I robot quadrupedi sono anche piuttosto comuni nell'ambito consumer perché sono relativamente stabili e facili da costruire.

Da un lato i quadrupedi sono piuttosto semplici, con pochi gradi di libertà nelle zampe ed le andature stabili sono abbastanza facili da ottenere. I robot bipedi, invece, sono più rari perché sono più complessi meccanicamente e la potenza di calcolo necessaria per il controllore è diventata disponibile solo da poco tempo. Il più famoso è sicuramente ASIMO, sviluppato dalla Honda. Anche NAO, sviluppato da Aldebaran è molto usato: è un robot piccolo, relativamente economico con una interfaccia piuttosto semplice. Altri modelli abbastanza utilizzati in vari studi sono HRP, WABIAN-R e THBIP-II. Tutti i robot di questo tipo hanno al più 7 gradi di libertà per ciascuna gamba, ma la maggior parte arrivano solo a 6 (divisi in 3 nell'anca, 1 nel ginocchio e 2 nella caviglia).

In confronto, le gambe umane hanno circa 30 gradi di libertà di cui molti nel piede. Si capisce quindi che i robot bipedi attuali forniscono solo una approssimazione, che è abbastanza valida se si considera solo la gamba, ma considerando invece l'intero sistema piede e gamba si vede che è piuttosto grossolana. Infatti i piedi nei robot sono progettati come un unico pezzo, senza alcun giunto, che quindi riduce molto le possibilità di movimento.

1.2 Metodi esistenti per la locomozione dei robot

La maggior parte degli approcci tradizionali per trovare strategie di controllo per robot dotati di zampe si fondano su metodi di ottimizzazione oppure sono basati sul modello del robot. Quindi questi metodi funzionano molto bene in contesti ben definiti, ma possono avere difficoltà nell'adattarsi ad ambienti differenti ([21]). Correntemente la ricerca si focalizza su metodi basati sulla traiettoria[1] e un criterio comune è lo *zero moment point* (*ZMP*), che garantisce la stabilità del movimento.

[1] Ovvero metodi in cui la traiettoria del giunto è disegnata completamente.

Recentemente sono stati definiti altri metodi euristici e altri modelli di controllo predittivo, ma sono caratterizzati da un costo computazionale molto importante, quindi risultano difficili da implementare in real-time.

1.2.1 ZERO MOMENT POINT

L'idea fondamentale consiste nel imporre un punto di contatto che non produca momento in direzione orizzontale, e poi calcolare, in maniera offline, una traiettoria che rispetti questo criterio sotto qualche vincolo (per esempio, limiti nella potenza dei motori). La stabilità che si ottiene tramite questi criterio è limitata e piccole perturbazioni possono causare una caduta. Attualmente ZMP è molto usato; per esempio sia Asimo sia NAO lo utilizzano.

Il criterio Zero Moment Point può essere usato anche se il robot deve fare qualcos'altro oltre a camminare: se il robot deve afferrare qualcosa da una superficie bassa, esso può piegare le ginocchia senza cadere. Quindi ZMP è molto pratico, nonostante abbia delle limitazioni: richiede una conoscenza precisa delle dinamiche del robot ed è impossibile da definire per andature nelle quali esiste un instante nel quale tutte le zampe sono sollevate (per esempio la corsa). Un'altra limitazione della camminata ottenuta con ZMP è che, benché sia stabile, essa non ricordi per niente la tipica andature umana: anzi, l'andatura bipede non è stabile secondo questo criterio.

1.2.2 PRIMITIVE DI MOVIMENTO

Le primitive di movimento sono un metodo recente (lo studio che le ha introdotte è del 2002 [38]) per controllare sistemi mobili. Semplificando i concetti, questo tipo di controllo è formato da due parti: una grossa librerie di traiettorie più un controllore che si occupa di modularle per produrre il segnale voluto.

Questo metodo è piuttosto interessante poiché le primitive di movimento sono relativamente semplici da calcolare, ed una volta che la libreria di movimenti è stata generata, il planner può selezionare e concatenare le primitive durante il moto per rispondere in maniera dinamica alla situazione. Lo svantaggio principale è la possibile perdita di informazioni sulle correlazioni tra i vari gradi di libertà (che possono essere utili per una uscita migliore). Inoltre è difficile determinare quali siano le primitive più efficaci, di conseguenza la maggior parte delle volte le primitive sono disegnate tramite l'intervento umano.

1.2.3 CENTRAL PATTERN GENERATOR

I Central Pattern Generator sono una rete neurale biologica che è in grado di produrre segnali ritmici anche in assenza di ingresso. Biologicamente non esiste una vera definizione: sono caratterizzati dalla loro funzionalità. Si è dimostrato sperimentalmente che essi sono presenti negli animali e controllano i movimenti ritmici: esistono infatti CPG per il movimento, per la respirazione, ...

L'utilizzo in robotica è relativamente recente: i primi studi risalgono agli anni 1990, ma sono stati ottenuti numerosi risultati a partire dagli anni 2000. Solitamente i Central Pattern Generator vengono modellati come sistemi dinamici accoppiati e quindi tramite le interazioni fra più processi riescono a generare dei segnali ritmici da fornire ai motori del robot. Questo metodo fornisce applicazioni molto interessanti per vari motivi, tra cui la stabilità del sistema risultante e la possibilità di utilizzarli in configurazioni distribuite. Il principale svantaggio è sicuramente la mancanza di una solida metodologia di sviluppo. Una discussione più dettagliata dei Central Pattern Generator con una spiegazione delle caratteristiche principali si trova nel capitolo 2.

1.3 Obiettivi della tesi

In questa tesi si vuole analizzare la generazione di movimenti ritmici in robot a zampe mediante un nuovo metodo, le Reti Neurali Caotiche. In particolare si effettua un confronto tra di esse ed i Central Pattern Generator con l'obiettivo finale di generare le traiettorie per i giunti di robot dotati di zampe.

Il confronto effettuato riguarda sia gli aspetti funzionali, come ad esempio quale metodo sia più versatile, sia gli aspetti ingegneristici, inquadrando le complessità degli algoritmi utilizzati e i tempi d'esecuzione. L'analisi è quindi sia teorica sia pratica. Per analizzare le caratteristiche da un lato sperimentale si è dimostrato necessario lo sviluppo di un simulatore per reti neurali caotiche, che è illustrato nel dettaglio. Infine, i due metodi sono confrontati: su robot simulati, utilizzando un robot reale e per ultimo su dati ricavati da soggetti umani.

1.4 Struttura della tesi

Nel capitolo 2 si introducono i Central Pattern Generator, ovvero un modello di locomozione per robot ispirato alle reti neurali esistenti negli animali.

Nel Capitolo 3 si introduce la soluzione alternativa ai Central Pattern Generator, ovvero le reti neurali caotiche (CNN). Anche esse sono ispirate alla biologia, ma noi ne parleremo in maniera più modellistica. Sono inoltre introdotte alcune euristiche per definire i parametri che caratterizzano questa rete.

Nel Capitolo 4 si parla del costo computazionale dei due metodi presentati, si descrive l'implementazione della libreria per simulare le CNN e le considerazioni effettuate nella progettazione.

Nel Capitolo 5 si presentano alcuni risultati iniziali riguardanti sia i tempi di simulazione, sia l'approssimazione di funzioni periodiche considerando entrambi i metodi.

Il Capitolo 6 presenta invece alcuni risultati riguardanti il confronto tra i Central Pattern Generator e le reti neurali caotiche su di problemi derivanti dalla locomozione. In particolare, si utilizzano i due metodi per approssimare le traiettorie dei giunti di robot, che sono provati, inizialmente in simulazione e poi su di un robot reale. Una parte è inoltre dedicata all'approssimazione di movimenti umani.

Il Capitolo 7 conclude il confronto tra le reti neurali caotiche e i central pattern generator e presenta le conclusioni del lavoro svolto, elencando anche alcuni possibili sviluppi futuri.

Infine, l'appendice A si concentra sui dettagli di basso livello della comunicazione col simulatore scelto.

CAPITOLO 2

Central Pattern Generator

2.1 Introduzione

I Central Pattern Generator (CPG) sono una rete neurale biologica che produce segnali ritmici senza avere in ingresso un feedback sensoriale. Numerosi risultati ottenuti studiando sia organismi semplici sia sistemi nervosi "in vitro" [48, 15] hanno dimostrato le uscite ritmiche di queste reti sono utilizzate per la locomozione, deglutizione, o altre attività oscillatorie.

I dettagli anatomici sono noti solo in pochi casi [20], però si verifica che i CPG sono solitamente piccoli [15] e che i pattern ritmici si possono sviluppare principalmente in due modi: tramite interazioni fra neuroni (detto anche *network-based synchronicity*) oppure all'interno dello stesso neurone (*endogenous oscillator neurons*), come risultato delle interazioni fra diverse correnti.

I CPG inoltre supportano un certo livello di neuro-modulazione, ovvero la rete può adattare l'uscita per rispondere meglio alle necessità dell'organismo. Per esempio, il CPG può produrre una nuotata lenta in assenza di input, ma arrivare a generare una nuotata veloce in presenza di un segnale in ingresso, quando è necessario scappare da un predatore [15].

Da numerosi esperimenti si è verificato che i central pattern generator si trovano in una grande varietà di animali, sia semplici (come le salamandre o le lamprede) sia più complessi, come i mammiferi. Per esempio, in [48] si è dimostrato che, isolando il sistema nervoso di una locusta, esso riesce a produrre la cosiddetta *fictive locomotion*: ovvero un segnale ritmico del tutto simile a quello osservato durante il volo. Questo esperimento è stato effettuato anche con altri animali, ma risultati così chiari sono possibili solo con invertebrati primitivi. L'organizzazione neuronale dei CPG non è nota in dettaglio in animali anche solo leggermente più complicati perché sono coinvolti numerosi neuroni [15].

Un altro esperimento molto importante che dimostra questi concetti è stato effettuato in [37]. In questa prova un gatto privo di corteccia celebrale è stato posto su di un tapis roulant inizialmente fermo. In seguito, il tapis roulant viene messo in azione: il gatto, di conseguenza, inizia a muoversi ed assume una andatura stabile. Quando la velocità del rullo viene aumentata il gatto inizia a camminare più velocemente finché non raggiunge un passo di trotto. In generale, se la velocità del tapis roulant viene cambiata, il gatto adatta la sua andatura per "sincronizzarsi" con il rullo. Questo esperimento dimostra che i circuiti responsabili del moto non sono nel cervello anche in animali più complessi di salamandre, e inoltre queste reti neurali possono rispondere a degli stimoli esterni.

2.2 Modellazione CPG

Il fatto che i CPG siano presenti negli animali li ha resi molto interessanti nel campo della robotica: di conseguenza nel corso degli anni sono stati sviluppati molti diversi modelli di CPG, che sono solitamente raggruppati in tre categorie:

- modelli biofisici: sono i modelli più dettagliati. Vengono solitamente usati per studiare il funzionamento delle singole cellule e in circuiti di dimensioni ridotte;
- modelli connettivi: sono più semplici dei primi, e vengono utilizzati per studiare come lo schema di connessione influenzi i pattern generati e le sincronizzazioni. Anche in questo caso si formalizzano i singoli neuroni, ma con modelli semplificati.
- modelli oscillatori: sono ancora più astratti dei precedenti, e si basano sui modelli matematici degli oscillatori. In questo caso, non si modellano più i neuroni singolarmente, ma l'oscillatore completo. Lo scopo non è studiare come viene generato il ritmo, ma come gli accoppiamenti modificano le dinamiche.

Come già anticipato, l'obiettivo di questo studio non è studiare gli aspetti biologici che permettono ad una rete di sviluppare un'uscita ritmica; ma capire come il comportamento oscillatorio permetta di far muovere gli arti di un robot, quindi il modello utilizzato appartiene alla terza classe, quella dei modelli oscillatori.

I Central Pattern Generator vengono modellati come sistemi dinamici accoppiati. Questa scelta produce alcuni vantaggi tra cui la stabilità, la sincronizzazione, che permette di coordinare i vari gradi di libertà, ed è semplice modulare la traiet-

toria in maniera continua cambiando semplicemente dei parametri. Inoltre, quasi tutti gli esperimenti noti sono tramite simulazione, integrando numericamente sistemi di equazioni differenziali, ma esistono anche alcune implementazioni in elettronica analogica o FPGA [16, 23, 22].

Esistono vari tipi di modelli neuronali per i CPG, basati su diversi oscillatori: i due modelli più usati per la locomozione di robot sono il modello di Matsuoka o di Hopf. Questo lavoro utilizza una piccola variante del modello in [36], che implementa un oscillatore di Hopf.

Il problema principale che si nota con i CPG è che non esiste alcuna metodologia per ottenere un sistema che riproduca il pattern voluto ([17]). Questa limitazione è piuttosto importante, e la maggior parte delle soluzioni esistenti in letteratura che utilizzano i CPG si limitano ad aggirare questo problema, utilizzando algoritmi di ottimizzazione di carattere generale, come gli algoritmi genetici. In questo lavoro non approfondirò il modello di Matsuoka, siccome esso soffre dei problemi descritti. Si nota che è poco pratico: dato che il "training" utilizza metodi di ottimizzazione esterni la simulazione risulta piuttosto complessa e dispendiosa.

I CPG sono stati spesso applicati alla locomozione di robot, ma la maggior parte degli studi si riferiscono al movimento di robot dotati di 4 o più zampe. Per esempio, nella tesi di laurea magistrale di Mattia Pirotti ([32]) si è utilizzato un CPG per realizzare le andature di un robot quadrupede sviluppato al Politecnico di Milano. Anche il nuoto è stato spesso oggetto di simulazioni, con salamandre o lamprede robot (un esempio è riportato in [17]). Numerosi esperimenti con i CPG sono stati realizzati sia dall'Hirose-Fukushima Robotics Lab (focalizzati principalmente sull'integrazione di sensori) sia all'ETH, da Ijspeert e Righetti, che invece studiano l'alternanza delle andature. In pochi casi (per esempio in [3], [29] e [7]) sono stati sviluppati alcuni modelli per robot bipedi, ma con un numero molto limitato di gradi di libertà (solitamente 6, 2 per gamba e 2 nell'anca).

2.3 Oscillatore adattivo di Hopf e PCPG

Questo modello è la versione base illustrata da Righetti in [36]. È denominato Oscillatore a Frequenza Adattiva (*Adaptive Frequency Oscillator, AFO*). Questa famiglia di oscillatori sfrutta le capacità di sincronizzazione tipiche dei sistemi dinamici accoppiati, ma utilizzano una ulteriore equazione per fare in modo che

la frequenza rimanga codificata nel sistema:

$$\begin{aligned}
\dot{x} &= \left(\mu - r^2\right) x - \omega y + \epsilon \, \mathrm{F}(t) \\
\dot{y} &= \left(\mu - r^2\right) y + \omega x \\
\dot{\omega} &= -\epsilon \, \mathrm{F}(t) \frac{y}{r} \\
r &= \sqrt{x^2 + y^2}
\end{aligned}$$

Dove μ ed ϵ sono costanti e $F(t)$ rappresenta il segnale che si vuole imparare. In dettaglio, l'ampiezza dell'oscillatore è data da $\sqrt{\mu}$ ed ϵ è un parametro che permette di variare la velocità di convergenza. Si può verificare che questo oscillatore riesce ad imparare una qualunque frequenza.

2.3.1 PROGRAMMABLE CENTRAL PATTERN GENERATOR

Visto che un AFO riesce ad approssimare un segnale composto da una sola armonica, l'idea di base per realizzare segnali arbitrari consiste nell'utilizzare un insieme di tanti oscillatori adattivi per riprodurre un segnale periodico. In questa sezione presento la rete neurale adatta a codificare una sola dimensione in uscita. L'estensione multidimensionale è naturale: consiste infatti nell'utilizzare multipli CPG accoppiati e sarà presentata in dettaglio più avanti. Anche questa soluzione è stata sviluppata da Righetti, e viene denominata *Programmable Central Pattern Generator, PCPG*. Questa regola è piuttosto interessante dal punto di vista teorico perché permette di estrarre le frequenze di un segnale in ingresso senza alcun algoritmo di signal processing esplicito. Inoltre funziona piuttosto bene anche in casi con rumore.

Un Programmable Central Pattern Generator è caratterizzato dalle seguenti equazioni, simili a quelle dell'AFO, a cui sono stati aggiunti due termini, uno per imparare le ampiezze e uno per la differenza di fase tra l'oscillatore k e il primo.

$$\begin{aligned}
\dot{x}_0 &= \gamma \left(\mu - r_i^2\right) x - \omega_i y + \epsilon F(t) \\
\dot{x}_i &= \gamma \left(\mu - r_i^2\right) x - \omega_i y + \epsilon F(t) + \tau \sin\left(\frac{\omega_i}{\omega_0}\theta_0 - \phi_i\right) \qquad \forall \, i > 0 \\
\dot{y}_i &= \gamma \left(\mu - r_i^2\right) y + \omega_i x \\
\dot{\omega}_i &= -\epsilon F(t) \frac{y_i}{r_i}
\end{aligned}$$

$$\dot{\alpha}_i = \eta x_i F(t)$$
$$\dot{\phi}_0 = 0$$
$$\dot{\phi}_i = \sin\left(\frac{\omega_i}{\omega_0}\theta_0 - \theta_i - \phi_i\right) \qquad \forall\, i > 0$$

con

$$\theta_i = \operatorname{sgn} x_i \arccos\left(-\frac{y_i}{r_i}\right)$$
$$Q_l(t) = \sum_{i=0}^{N} \alpha_i x_i(t)$$
$$F(t) = P(t) - Q_l(t)$$
$$r_i = \sqrt{x_i^2 + y_i^2}$$

L'output del CPG è $Q_l(t)$, che è la combinazione lineare dell'output di ogni oscillatore. $F(t)$ rappresenta il feedback negativo: la parte del segnale $P(t)$ che la rete deve ancora imparare. α_i è l'ampiezza associata alla frequenza ω_i dell'oscillatore i.

L'equazione dell'evoluzione di α modifica effettivamente il valore solo quando il valore ω associato raggiunge una frequenza presente in $F(t)$.

Rispetto alla versione adattiva semplice, descritta sopra, nel PCPG è stato aggiunto un termine di accoppiamento alle x, $\tau \sin\left(\frac{\omega_i}{\omega_0}\theta_i - \phi_i\right)$. Questo termine serve per nascondere l'effetto delle frequenze non imparate perfettamente. È importante notare che questo termine non va aggiunto al primo oscillatore del CPG.

In questo modello ci sono 5 parametri che si possono scegliere:

- il numero degli oscillatori N
- la forza di accoppiamento τ
- la costante γ
- i due tempi di rilassamento ϵ per le frequenze e η per i pesi.

Il numero degli oscillatori è semplicemente il massimo numero di frequenze diverse che è possibile imparare: in generale, quindi, più è alto, più ampio sarà l'insieme di segnali che è possibile imparare. Ovviamente si deve anche considerare che un N troppo alto influenza in maniera negativa la velocità di simulazione.

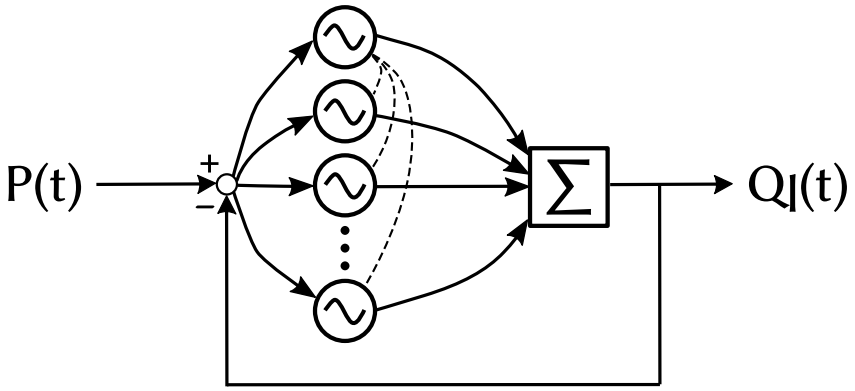


Figura 2.1: Schema dei collegamenti tra gli oscillatori in un CPG. Ogni oscillatore riceve lo stesso ingresso, $F(t) = P(t) - Q_I(t)$. Tutti gli oscillatori tranne il primo hanno anche un termine di accoppiamento in ingresso.

Il fattore τ serve per mantenere una corretta differenza di fase tra l'oscillatore i e l'oscillatore 0. Valori prossimi allo zero significano un accoppiamento basso fra le componenti: questo implica che se la prima componente è leggermente più veloce (o più lenta) del necessario, sul lungo periodo lo scostamento fra la prima e la seconda componente sarà sempre più alto. Un valore alto di τ, invece, riesce a contrastare questa deriva e permette di ottenere segnali più sincronizzati. D'altra parte, però, valori troppo alti tendono a produrre sistemi instabili.

Il parametro γ permette di controllare la velocità di recupero dopo le perturbazioni. Questo parametro non influenza molto la qualità delle soluzioni trovate, quindi viene mantenuto ad 8, ovvero il valore riportato da [36].

Gli ultimi due parametri, ϵ e η, sono delle costanti per il learning. Valori più bassi permettono di avere convergenze più precise, ma anche più lente. Anche per questi due parametri, da prove sperimentali si nota che i valori proposti in [36], ovvero $\eta = 0.5, \epsilon = 0.9$, sono validi per la maggior parte dei problemi.

C'è una piccola differenza rispetto al modello proposto da Righetti per quanto riguarda il termine di accoppiamento, infatti in [36] si suggerisce di usare $\tau \sin(\theta_i - \phi_i)$, quindi senza il fattore di correzione per le velocità. Entrambe le versioni producono un oscillatore funzionante, infatti il valore effettivo dell'accoppiamento non è critico.

2.4 Caratteristiche della rete

Ogni oscillatore riceve lo stesso segnale in ingresso, ovvero la differenza tra il segnale imparato e il segnale da approssimare. Siccome si usa un feedback negativo, questa differenza arriva a zero quando le componenti in frequenza vengono imparate. Il singolo oscillatore tende ad adattarsi alla componente a cui è più vicino, e la dimensione del "bacino d'attrazione" dipende dalla potenza della sinusoide. Quindi, per imparare accuratamente il segnale, si devono avere abbastanza oscillatori nell'anello per eliminare la rispettiva componente in $F(t)$.

Se ci sono meno oscillatori del necessario il sistema impara solo le frequenze associate ai moduli più elevati. Viceversa, se ci sono troppi oscillatori, si possono verificare due casi: gli oscillatori non necessari convergono ad $\alpha_k = 0$, oppure più oscillatori raggiungono la stessa frequenza e la somma dei rispettivi α sarà uguale all'ampiezza della componente.

Questo oscillatore possiede alcune proprietà di stabilità che sono inerenti all'oscillatore di Hopf, e pertanto produce traiettorie stabili rispetto alle perturbazioni. Questa caratteristica potrebbe essere utile per integrare ingressi aggiuntivi.

Una altra caratteristica importante è che permette una modulazione semplice delle ampiezze e delle frequenze che compongono la traiettoria, semplicemente modificando i vettori ω e α. Inoltre, per le proprietà delle soluzioni ad equazioni differenziali (si ha $x, y \in C^1([a, b])$), la modifica di ω produrrà sempre una traiettoria continua. La modifica del valore di α potrà presentare discontinuità di salto, ma si può ottenere lo stesso effetto di amplificazione mantenendo la continuità modificando il valore di μ.

Questo modello è essenzialmente equivalente ad una scomposizione in serie di Fourier "dinamica", pertanto è possibile imparare un qualunque segnale periodico.

2.5 Proprietà

In questa sezione presento un semplice esperimento numerico per verificare le proprietà del sistema illustrato sopra. Il segnale che si vuole approssimare è:

$$f(t) = \sin 6t - \frac{1}{2}\cos 12t + \frac{7}{5}\sin\left(18t + \frac{\pi}{3}\right) \tag{2.1}$$

La rete che utilizzata è composta da 4 oscillatori. La figura 2.2 presenta i risultati.

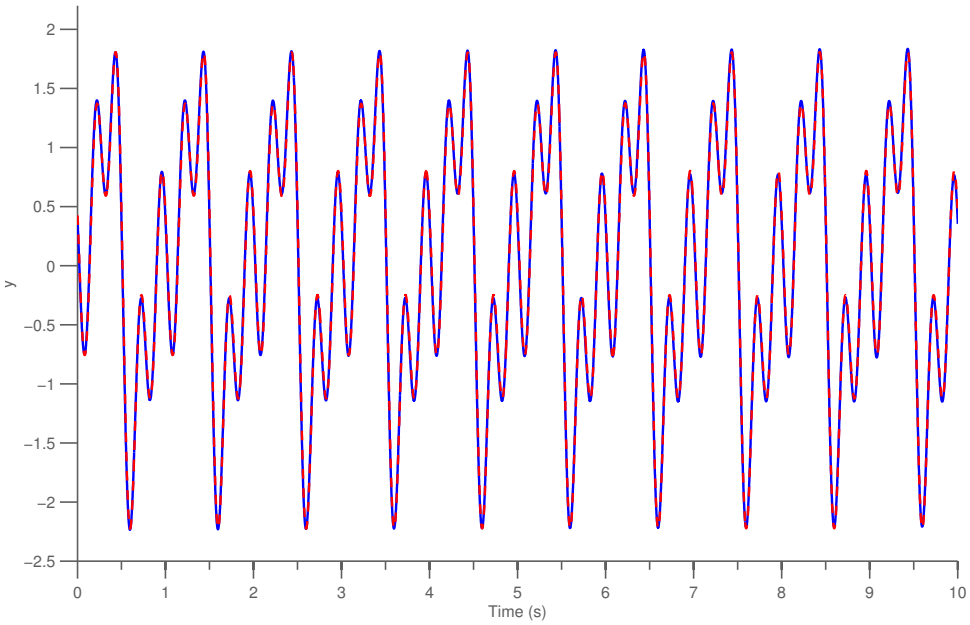


Figura 2.2: Risultati del learning PCPG. In blu è rappresentato l'output del PCPG, in rosso l'uscita desiderata. Come si vede, il segnale viene riprodotto in maniera molto precisa.

In particolare, si può vedere in figura 2.3, come vengono adattatate rispettivamente le frequenze e i pesi. Come anticipato, ω converge al valore corretto della frequenza, e α viene adattato in seguito. Inoltre, il terzo oscillatore (rosso nel grafico) ha come valore associato $\alpha_3 = 0, \omega_3 = 15$, valori a cui però converge solo dopo che tutti gli altri oscillatori hanno raggiunto un valore corretto.

In figura 2.4 si mostra invece come reagisce il sistema ad un cambio di frequenze e ampiezze. In entrambi i casi si nota che il cambio di parametri produce un cambiamento della traiettoria in modo continuo.

I valori scelti per la simulazione sono i seguenti: $N = 4$, $\gamma = 8$, $\mu = 1$, $\tau = 1$, $\epsilon = 0.9$ ed $\eta = 0.5$. Come valori iniziali per l'integrazione: di $x(0) = [1, 1, 1, 1]$, $y(0) = [0, 0, 0, 0]$, $\alpha(0) = [0, 0, 0, 0]$, $\phi = [0, 0, 0, 0]$ e $\omega = [5, 10, 15, 20]$. Il sistema è stato simulato per il training per 500 secondi con uno step di integrazione pari a $\delta = 1$ ms.

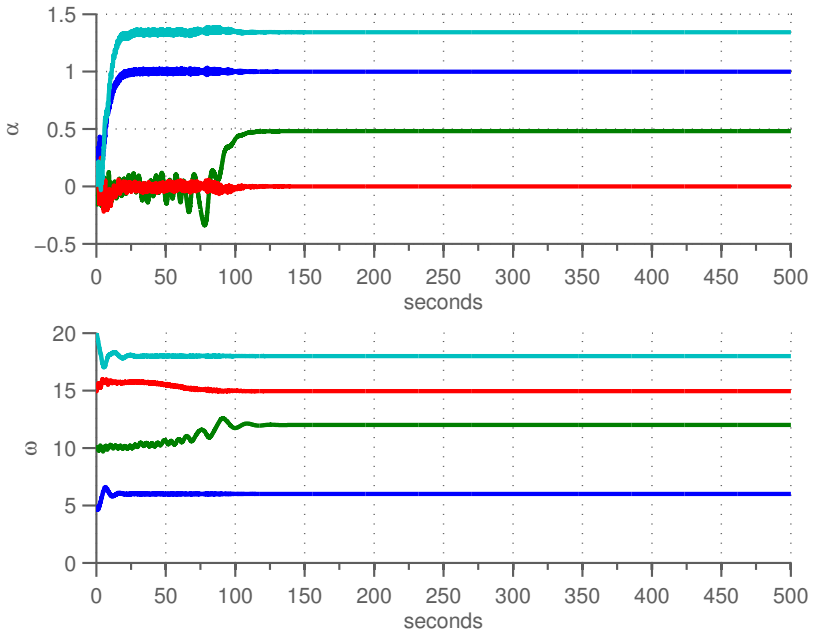


Figura 2.3: Evoluzione dei valori di α (grafico superiore) e ω (grafico inferiore) durante il training.

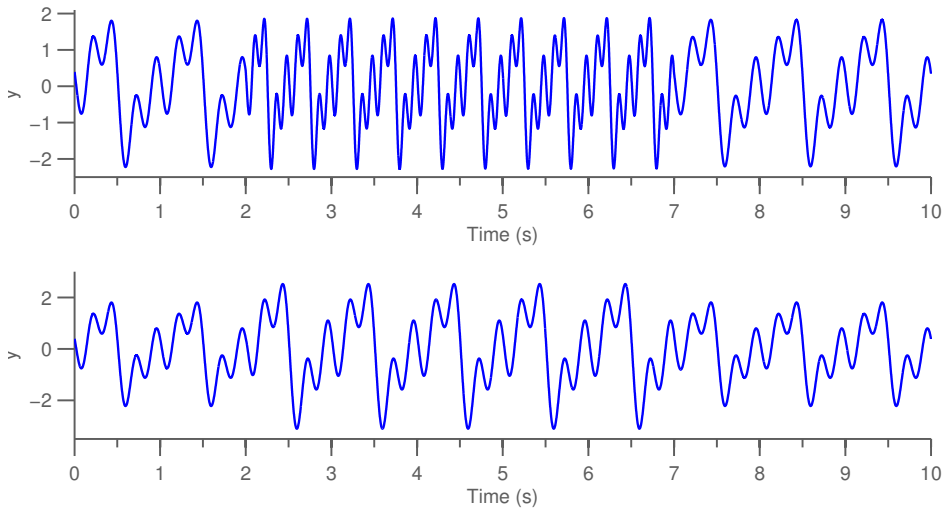


Figura 2.4: Modulazione dei valori di μ (grafico inferiore) e ω (superiore). Al tempo $t = 2$, μ viene raddoppiato (quindi l'amplificazione diventa $\sqrt{2}$) e ω dimezzato. Al tempo $t = 7$ entrambi ritornano al loro valore originale.

2.6 Estensione a M dimensioni

Nella maggior parte dei casi è necessario realizzare più di una uscita. Per esempio, nel caso di controllori per robot si devono avere tanti output quanti sono i gradi di libertà che si vogliono controllare. L'estensione è abbastanza semplice: si usa un CPG diverso per ogni grado di libertà da controllare, e ciascuno di essi viene coordinato in fase tramite un fattore di accoppiamento, simile a quanto fatto prima.

In dettaglio, si deve modificare il primo oscillatore di ogni CPG in questo modo:

$$\dot{x}_{0,k} = \gamma \left(\mu - r_{0,k}^2\right) x_{0,k} - \omega_{0,k}\, y_{0,k} + \tau \sin\left(\theta_{0,k-1} - \theta_{0,i} - \phi_{0,i}\right)$$
$$\dot{\phi}_{0,k} = \sin\left(\theta_{0,k-1} - \theta_{0,k} - \phi_{0,k}\right)$$

Dove con la notazione $x_{i,j}$ si rappresenta lo stato x_i dell'oscillatore j (e analogamente per le altre variabili). $\theta_{0,k-1}$ rappresenta il valore di θ_0 del CPG collegato in ingresso al k-esimo CPG. Il primo oscillatore della catena (non collegato ad alcun altro CPG "in ingresso") continua ad utilizzare l'equazione originale.

Anche in questo caso il coefficiente τ serve per conservare la differenza di fase tra i due CPG collegati. Si nota che non è prescritta una topologia, si possono collegare due CPG in modo pressoché qualunque. Per una applicazione come la locomozione bipede la topologia più naturale da sfruttare è quella riportata in figura 2.5: si collegano "in serie" i vari gradi di libertà di uno stesso arto, e le due gambe sono collegate in modo da mantenere la differenza di fase necessaria tra di esse.

2.7 Altri modelli simili

L'idea del PCPG è stata usata in versioni piuttosto simili anche da altri autori. In particolare, è interessante la versione proposta da Nakanishi et al. in [30] e poi elaborata da Gams e Ijspert ([10]) che utilizzano un modello equivalente al PCPG di Righetti, ma introducono una ulteriore componente di filtraggio dopo il CPG per riprodurre meglio il segnale.

Il vantaggio principale di questo approccio è che spesso bastano meno oscillatori del necessario, in quanto poi il filtro (anch'esso sottoposto ad un processo di training) riesce a migliorare l'output. Ovviamente, però, avendo un ulteriore

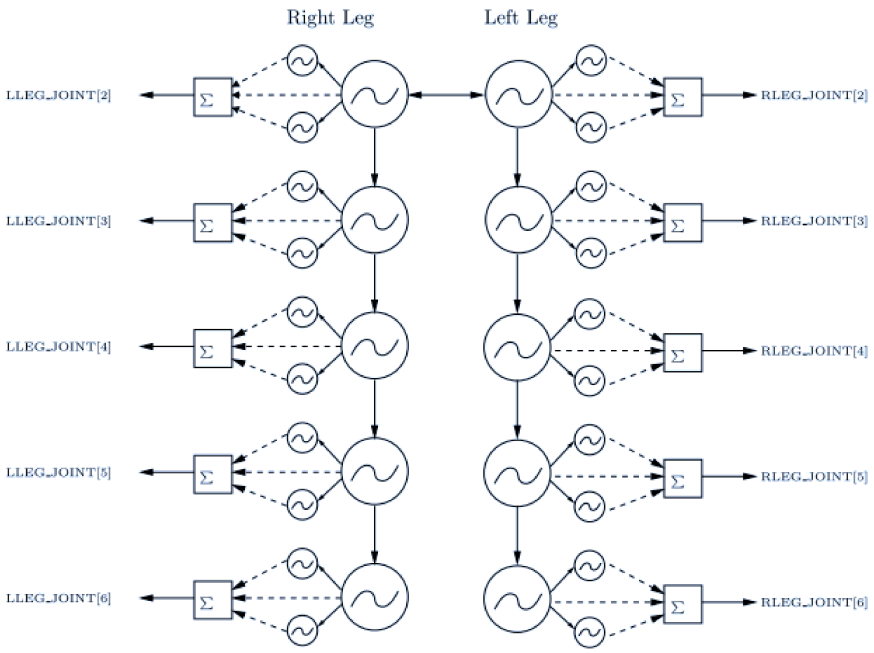


Figura 2.5: Possibile topologia di connessione tra CPG. Immagine tratta da [36].

blocco di cui fare il training, la velocità del procedimento tipicamente ne risente in maniera negativa.

Questa soluzione è stata usata in due esperimenti: il primo riguarda la locomozione bipede, ma con solo 4 gradi di libertà (2 nell'anca, 1 nel ginocchio destro e 1 nel ginocchio sinistro), mentre il secondo si riferiva al controllo degli 8 gradi di libertà delle braccia di un robot simile ad Asti.

2.8 Fourier CPG

Il principale difetto dei PCPG di Righetti è la convergenza lenta. Per risolvere questo problema è stato sviluppato un algoritmo offline che permette di trovare i parametri di un CPG di Hopf in maniera molto più veloce. Si basa sulla trasformata di Fourier veloce per produrre lo stesso risultato dei PCPG.

L'algoritmo è molto semplice, ma sono possibili alcune varianti per rendere il training più "autonomo": per esempio, si può fare in modo che l'algoritmo scelga il numero ottimale di oscillatori in base ad una soglia sull'errore.

Per utilizzare l'oscillatore di Hopf, serve ancora calcolare lo stato iniziale di

Algorithm 1 Impara i parametri di un PCPG tramite analisi di Fourier

function FOURIERCPG-TRAIN(signal, dt, N)
  $L \leftarrow \text{length}(\text{signal})$
  $Y, \text{freq} \leftarrow \text{fft}(\text{signal})/L$
  $\text{spectrum} \leftarrow Y(0 : L/2)$ ▷ Considera la parte sinistra della fft
  $A \leftarrow 2|\text{spectrum}|$ ▷ Calcola il modulo dello spettro
  for $k \leftarrow [0, 1, \ldots, N)$ **do**
    $i \leftarrow \arg\max A$ ▷ L'indice a cui corrisponde l'ampiezza massima
    $\alpha_k \leftarrow A[i]$
    annulla A in un intorno di i
    $\theta_k \leftarrow -\text{atan2}(\text{Im}\{Y(i)\}, \text{Re}\{Y(i)\})$ ▷ Fase nel punto
    $\omega_k = 2\pi\,\text{freq}[i]$
  return α, ω, θ

x, y e ϕ. Per i primi due si può usare la definizione: $x = \cos(\theta), y = \sin(\theta)$. Per calcolare ϕ si deve invece notare che esso rappresenta una differenza di fase (istantanea) fra il primo oscillatore e gli altri.

Formalmente, la regola di calcolo di ϕ è la seguente:

$$\begin{cases} \phi_{0,j} = \theta_{0,j} - \theta_{0,j-1} \\ \phi_{i,j} = \theta_{i,j} - \theta_{0,j} \end{cases}$$

Il primo oscillatore di ogni CPG ha ϕ uguale alla differenza tra le fasi del primo oscillatore del CPG precedente e la sua. Inoltre, ogni oscillatore nello stesso CPG ha come valore di ϕ la differenza tra la propria fase e la fase del primo oscillatore.

2.9 Conclusioni

In questa sezione è stato presentato un modello di CPG che permette di imparare traiettorie periodiche arbitrarie. Possiede diversi vantaggi, alcuni condivisi tra tutti i modelli di CPG:

- hanno una profonda base biologica;
- sono a tutti gli effetti distribuiti: ad ogni grado di libertà corrisponde un CPG diverso, questo li rende particolarmente appetibili per robot modulari, sistemi paralleli o in generale microcontrollori;

- la modulazione del movimento è relativamente semplice e bastano segnali di alto livello;
- possono integrare il feedback sensoriale

e altri invece relativi solo al modello illustrato, come la possibilità di fare online learning. Però si devono anche elencare alcuni problemi: i modelli molto generali implicano la difficoltà nell'avere una metodologia di progetto valida ed efficiente. Utilizzare molti oscillatori rende i calcoli onerosi, e altri svantaggi risiedono nelle equazioni differenziali: sono numericamente complesse, quindi non sempre il metodo di Eulero in avanti produce buoni risultati, e sono spesso instabili (*stiff*[1]), quindi si deve usare un tempo di integrazione molto piccolo o dei solver piuttosto complessi: in entrambi i casi la simulazione è molto rallentata. In aggiunta, il tempo di learning per questo algoritmo è elevato, specialmente se le condizioni iniziali per l'integrazione delle equazioni differenziali non sono ottimali (per esempio, se i valori iniziali di ω sono molto diversi dai valori reali). Questi ultimi problemi vengono risolti dall'algoritmo FourierCPG, che tenta di rendere i PCPG più pratici.

[1] Una equazione differenziale è "stiff" quando i metodi numerici per risolverla sono instabili a meno che il passo di integrazione sia molto piccolo. Non esiste una definizione formale di *stiffness* [34].

CAPITOLO 3

Reti neurali caotiche

3.1 Introduzione

Con reti neurali caotiche si intende una classe di sistemi dinamici la cui evoluzione mostra una dipendenza dalle condizioni iniziali. Questo fenomeno si verifica anche se il sistema è deterministico, senza il coinvolgimento di fattori stocastici. È ormai noto che una componente caotica è benefica, anche in sistemi "naturali", infatti permette l'esplorazione di un intervallo più ampio di condizioni, pur mantenendo un certo grado di memoria e predicibilità ([42]).

Ci concentreremo su una classe di reti neurali che esibisce comportamenti caotici: le reti neurali ricorrenti (RNN). Le RNN sono una naturale estensione delle reti neurali di tipo *feed-forward* in cui sono presenti delle interconnessioni fra i neuroni che formano dei cicli. Formalmente, le RNN implementano dei sistemi dinamici, al contrario delle reti neurali classiche, che forniscono solo un mapping funzionale. Nonostante la superiore potenza computazionale delle RNN, esse non vengono usate molto perché l'addestramento risulta piuttosto complicato. Esistono vari algoritmi per il training, ispirati alla *backpropagation* (per esempio, la *backpropagation through time*), che però sono piuttosto complessi computazionalmente, e tendono a convergere a minimi locali (per una analisi di questi metodi e le loro caratteristiche, si può vedere [1], [44] o [13])

Recentemente, per risolvere questo problema, sono state sviluppate due idee: le *Echo State Network* in [19], e le *Liquid State Machine*, da Maass ([28]). Nonostante usino un formalismo leggermente diverso e modelli neuronali differenti, si basano su alcuni concetti fondamentali molto simili, e infatti ora vengono spesso unificate sotto il nome di *Reservoir Computing* (RC) (questa terminologia è introdotta in [46]). La caratteristica principale del RC è la separazione della rete ricorrente in due parti: la parte detta *reservoir*, che può contenere connessioni ricorrenti e la parte che produce l'output, il *readout*. Il reservoir è un sistema dina-

mico fissato, generato casualmente e molto grande (configurazioni da 500 o 1000 neuroni sono generalmente usate). Il readout invece è un modello semplice, solitamente lineare. Questa configurazione permette di considerare il reservoir con una black-box e di fare il training del solo readout. Il modello di neurone per il readout è scelto per fare in modo che l'addestramento sia veloce ed efficace, senza i problemi di cui soffrono i metodi per il training delle RNN generiche.

Una interessante interpretazione del reservoir computing è fornita dalla teoria dei segnali: si può vedere il reservoir come un insieme di funzioni complesse che fungano da base funzionale per riprodurre un segnale qualunque, tramite una combinazione lineare calcolata dal readout.

La differenza principale tra LSM e ESN risiede nel tipo di reservoir utilizzato: nelle Liquid State Machine viene solitamente usato un modello di neurone binario di tipo *Leaky Integrate and Fire* (*LIF*), mentre per le ESN si preferisce neuroni analogici, per esempio neuroni sigmoidali o Leaky Integrator (LI). I neuroni LIF sono biologicamente più plausibili, ma la simulazione risulta più complessa, quindi le applicazioni sono di un tipo più teorico (anche se sono state usate con successo per alcuni task di classificazione, per esempio in [47]).

Le reti caotiche (CNN) qui considerate si ispirano come modello architetturale al reservoir computing, infatti si sfrutta la divisione tra reservoir e readout, ma esiste una differenza di tipo concettuale: nel reservoir computing si usano reti inattive prima del training, le cui attività neuronali, eccitate da segnali in input, si annullano dopo un transitorio (chiamata *Echo State Property*). Si opera ovvero in un regime non caotico, mentre nel modello qui considerato il caos viene sfruttato durante il training per trovare il punto operativo migliore.

3.2 Utilizzi

Le reti neurali sono spesso usate in robotica per scopi come il path planning o la visione artificiale. Anche il reservoir computing è stato usato spesso in ambito robotico, per esempio in [2] viene usata una Echo State Network per effettuare la localizzazione di robot integrando le informazioni dei sensori. In [33], gli autori suggeriscono l'utilizzo di una Echo State Network per muovere un robot dotato di ruote in base all'ambiente circostante. Per quanto riguarda la locomozione, invece, le CNN sono state utilizzate in [45] per generare la camminata di un essere umano.

Un altro progetto che si può nominare è attualmente in corso, il cui principale referente è il dottor Folgheraiter, e riguarda lo sviluppo di un umanoide per

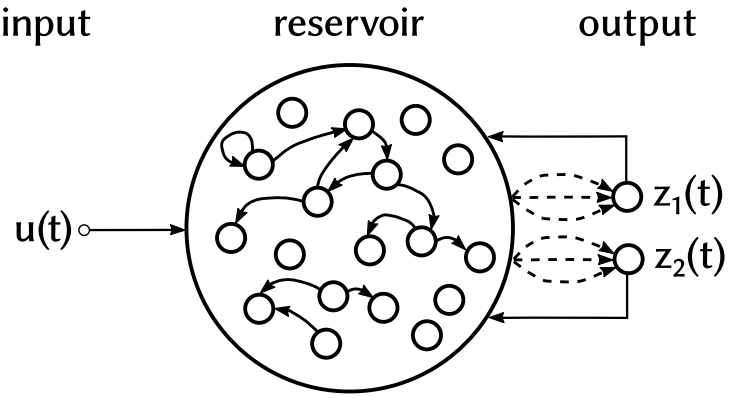


Figura 3.1: Una rete neurale caotica con 1 input e 2 output. Nel reservoir computing è si addestrano solo le connessioni tra il reservoir e il readout (tratteggiate nella figura). Le uscite sono poi retroazionate nel reservoir. Il reservoir è caratterizzato da un numero grande di neuroni, connessi fra di loro in modo casuale e sparso.

applicazioni in ambienti pubblici. Correntemente il progetto è in una fase ancora abbastanza iniziale: un primo prototipo di una gamba è stato sviluppato e il robot completo è solo simulato. Il controllore per questo robot è una rete neurale caotica ([8, 9]), implementato tramite la libreria disegnata in questo lavoro di tesi. Il team del professor Folgheraiter ha anche implementato una versione della libreria in Python, per la piattaforma Raspberry Pi2. Tra gli obiettivi futuri di questo progetto rientra l'aggiunta di feedback sensoriali per modificare le traiettorie generate dalla CNN; si vuole per esempio includere la possibilità di evitare gli ostacoli rilevati tramite i sensori.

Inoltre non mancano usi più tipici, ispirati al machine learning. In alcuni casi il reservoir computing viene usato per la predizione di serie temporali, per esempio in [18] o [49]. Questa soluzione è stata anche applicata alla classificazione: una interessante applicazione del reservoir computing è il riconoscimento vocale. In alcuni studi ([46, 39]), infatti, gli autori sono stati in grado di raggiungere le performance dei software considerati "stato dell'arte", arrivando anche ad una accuratezza del 97%.

3.3 Modello di rete utilizzato

Come già anticipato, nel campo del reservoir computing sono utilizzati svariati modelli di neurone. La trattazione seguente si concentra su un modello definito

dalle seguenti equazioni:

$$\begin{aligned} \tau \dot{x}_i(t) &= -x_i(t) + \lambda \sum_{j=1}^{N} R_{ij} r_j(t) + \sum_{j=1}^{M} W_{ij}^{fb} z_j(t) + \sum_{j=1}^{L} W_{ij}^{in} u_j(t) \\ r_i(t) &= \tanh(x_i(t)) + \xi_i(t) \\ z_j(t) &= \sum_{i=1}^{N} W_{ij} r_i(t) + \xi_j(t) \end{aligned} \tag{3.1}$$

Dove:

- N è la dimensione del reservoir
- M è il numero delle uscite
- L è il numero degli ingressi
- τ è un parametro di temporizzazione, chiamato tempo di membrana (*membrane time*)
- λ è un parametro che indica la caoticità del sistema
- $\xi(t)$ sono dei termini di rumore

Le $x_i(t)$ sono dette attivazioni e $r_i(t)$ *firing rates*. Il neurone i riceve ingressi dagli altri neuroni con rate $r_k(t)$ tramite sinapsi con pesi R_{ik}. L'ingresso viene dato dalle M sinapsi W^{in}, che collegano ciascun ingresso ad ogni neurone nel reservoir. Infine, $z(t)$ è il readout, e fornisce una proiezione dello stato interno. L'unità di uscita è connessa a ciascun neurone del reservoir tramite i pesi W, e viene retroazionata nella rete con le sinapsi di feedback W^{fb}.

Per comodità d'espressione si può riscrivere la forma sopra in modo equivalente facendo ricorso all'espressione matriciale, utilizzando la naturale convenzione che l'applicazione di una funzione ad un vettore produce un vettore che ha come elementi il risultato dell'applicazione della funzione elemento per elemento (quindi, $f([\, v_0 \; v_1 \; \dots \,]) = [\, f(v_0) \; f(v_1) \; \dots \,]$):

$$\begin{aligned} \tau \dot{x}(t) &= -x(t) + \lambda R r(t) + W^{fb} z(t) + W^{in} u(t) \\ r(t) &= \tanh(x(t)) + \xi(t) \\ z(t) &= W^{\intercal} r(t) + \xi(t) \end{aligned}$$

Inoltre, in certe occasioni ed ovviamente nelle simulazioni, sarà utilizzata la ap-

prossimazione a differenze finite per analizzare il comportamento della rete:

$$x(t+\delta) = \left(1 - \frac{\delta}{\tau}\right) x(t) + \frac{\delta}{\tau}\left(\lambda R r(t) + W^{fb} z(t) + W^{in} u(t)\right)$$

Dove δ è il passo di integrazione. Nelle sezioni seguenti mi riferirò a questo modello come modello 3.1.

Un reservoir appartenente a questa famiglia è definito totalmente dalla tupla $(\tau, \lambda, R, W^{fb}, W^{in})$. Di questi, solo i primi 2 (τ e λ) possono essere modificati direttamente, mentre i rimanenti sono generati in maniera casuale, dipendente da dei *meta-parametri* (descritti in sezione 3.3.1). Anche lo stato iniziale del reservoir $x(0)$ è generato casualmente, solitamente estratto da una distribuzione normale con varianza 0.5 e media 0.

Il parametro di discretizzazione δ è grossomodo fissato dal problema. Infatti esso rappresenta semplicemente l'intervallo tra un dato e l'altro, che quindi è anche un valore che cambia la frequenza massima rappresentabile dalla rete. Esiste una formula per calcolare la frequenza di taglio del filtro, che è derivata nella sezione 3.6. In maniera dipendente dal problema, quindi, potrebbe essere necessario fare *oversampling* per fare in modo che la rete possa rappresentare tutte le frequenze del segnale. In casi più rari si può utilizzare il *subsampling*, che ha anche il pregio di abbreviare i tempi di computazione, tenendo però presente che un δ troppo grande causa un errore di discretizzazione significativo.

In generale, inoltre, le equazioni del reservoir saranno anche rappresentate tramite il simbolo $\mathcal{R}$. Quindi, in maniera compatta si indicherà l'evoluzione temporale del sistema con:

$$y \leftarrow \mathcal{R}(z, u)$$
$$z = W^{\mathsf{T}} y(t)$$

Nelle prossime sezioni descriverò brevemente altri modelli di neurone solitamente utilizzati in questo ambito, e elencherò i motivi che dovrebbero essere tenuti in considerazione quando si effettua la scelta del modello.

3.3.1 LA TOPOLOGIA

La topologia della rete è generata casualmente. Le distribuzioni scelte per i collegamenti tipicamente sono:

- R: è una matrice con parametro di sparsità p. I valori diversi da zero sono estratti da una distribuzione normale, con media 0 e varianza $\frac{1}{pN}$ (da cui segue che $\rho(R) \approx 1$ [11])[1]
- W^{in} e W^{fb} sono invece uniformemente distribuiti in $[-1, 1]$.

Da numerosi esperimenti (per esempio, [19]), vari autori notano che la rete è abbastanza robusta rispetto alla scelta di questi parametri, e distribuzioni diverse funzionano spesso quasi altrettanto bene.

Un fattore invece più importante è dato dalla sparsità della matrice R. Infatti, la scelta del valore p influenza sia l'errore ottenuto sia le performance (in termini di tempo d'esecuzione) del sistema. Matrici più sparse tendono a produrre reti con errori minori, e inoltre, come motiverò nella sezione 4.3.1, una matrice più sparsa implica migliori performance della simulazione.

Nella letteratura sono state proposte diverse soluzioni più sofisticate (per una analisi si può vedere [5]) per la generazione della topologia del reservoir, in modo da sfruttare meglio le informazioni note sull'obiettivo da approssimare. Rispetto all'inizializzazione standard queste procedure mostrano miglioramenti ridotti (come riportato anche in [26]); in questa trattazione, quindi, non discuteremo dei metodi alternativi.

3.3.2 IL TEMPO DI MEMBRANA τ

Dimensionalmente τ è un tempo ed è collegato alla memoria del neurone. Inoltre, dà un limite superiore alla frequenza del segnale in uscita. Più in dettaglio, al crescere di τ diminuisce la frequenza del segnale in uscita. Di conseguenza, τ deve essere impostato per seguire la velocità del segnale desiderato in uscita.

3.3.3 FATTORE DI CAOS λ

Il fattore λ serve invece per scalare i pesi delle connessioni ricorrenti. Si verifica facilmente che reti con $\lambda < 1$ sono inattive prima dell'addestramento: l'output di ciascun neurone, dopo un transitorio iniziale, diventa costante e pari a 0. Un comportamento più interessante si ottiene invece per $\lambda > 1$: infatti la rete esibisce attività caotica spontanea, che diventa più irregolare all'aumentare del parametro [40]. In alcuni modelli di neurone il parametro λ è "nascosto" all'interno della matrice R; in questo caso $\lambda \approx \rho(R)$. Si tratta di una semplice differenza di notazione

[1] con $\rho(X)$ si denota il raggio spettrale — il massimo tra i moduli degli autovalori — di X

in quanto nel modello spiegato finora il raggio spettrale di R è 1, e λ permette di scalarlo, mentre altri autori preferiscono rendere implicito il parametro.

3.4 Feedback

Nonostante il reservoir sia fissato, per la maggior parte dei task che considereremo il readout è retroazionato nel reservoir. La presenza del feedback aggiunge potenza alla rete, rendendola equivalente ad una rete ricorrente usuale, che può quindi approssimare qualunque sistema dinamico, come dimostrato in [27].

Si può verificare che la presenza del feedback permette di modificare le connessioni interne della rete semplicemente addestrando il readout. Infatti, considerando senza perdere di generalità la rete senza input:

$$\dot{x}(t) = -x(t) + Rr(t) + W^{fb}z(t) \tag{3.2}$$

$$z(t) = W^{\intercal}r(t) \tag{3.3}$$

Si può riscrivere, sostituendo $z(t)$ nell'equazione 3.2, nella forma:

$$\begin{aligned}\dot{x}(t) &= -x(t) + \lambda Rr(t) + W^{fb}W^{\intercal}r(t) \\ &= -x(t) + \underbrace{(\lambda R + W^{fb}W^{\intercal})}_{W^{eff}} r(t)\end{aligned}$$

Da qui segue direttamente che quando il metodo di training modifica i pesi in W, modifica anche la topologia "effettiva" della rete ricorrente.

Nonostante con il reservoir computing sia possibile calcolare uscite non banali in funzione degli ingressi alla rete, il feedback rende la rete autonoma: è possibile generare un segnale in uscita anche senza avere un ingresso, ottenendo di fatto un generatore di pattern. Questo è il principale motivo per cui nel seguito di questa tesi tratteremo esclusivamente questo tipo di reti. Il feedback ha uno scopo duale: da una parte serve per controllare la caoticità del sistema e trasformare il regime caotico iniziale nell'andamento desiderato dopo il training ma può anche produrre problemi di stabilità. Infatti, come è riportato in [35], si possono avere piccole variazioni nell'output che vengono amplificate ad ogni passo, producendo così deviazioni sempre più grandi rispetto al segnale obiettivo. Per questo motivo nel campo del reservoir computing classico numerosi autori (per esempio [25]) suggeriscono di evitare il feedback, se il task da imparare lo permette.

3.5 Limitatezza

Una proprietà desiderabile per un sistema dinamico, specialmente se lo scopo è la simulazione su un calcolatore, è la limitatezza della traiettoria. Questo vuol dire che la traiettoria di un sistema con un ingresso limitato deve sempre rimanere in un sottoinsieme compatto $A \subset \mathbb{R}^M$.

Nel modello 3.1 la compattezza è garantita a patto di scegliere un passo di integrazione sufficientemente piccolo; infatti:

$$x(t+\delta) = \left(1 - \frac{\delta}{\tau}\right) x(t) + \frac{\delta}{\tau}\left(W^{eff} r(t) + W^{in} u(t)\right)$$

La seconda parte dell'equazione è sicuramente limitata: $u(t)$ lo è per ipotesi, e $r(t)$ è il risultato dell'applicazione di una funzione sigmoidale allo stato precedente, quindi è compresa in $[-1, 1]$, e la moltiplicazione per le rispettive matrici introduce solo un fattore di scala. Per avere che la prima parte sia limitata si deve avere $|1 - \frac{\delta}{\tau}| < 1$.

$$\begin{cases} 1 - \frac{\delta}{\tau} < 1 \\ 1 - \frac{\delta}{\tau} > -1 \end{cases} \qquad \begin{cases} \frac{\delta}{\tau} > 0 \\ \frac{\delta}{\tau} < 2 \end{cases}$$

Il primo vincolo è sempre verificato perché sia δ sia τ sono tempi, quindi positivi. Il secondo, invece, è più importante e dovrebbe essere controllato, ma si può anche evidenziare il fatto che fisicamente $\frac{\delta}{\tau}$ rappresenta il "flusso perso" (*leak rate*), quindi solitamente dovrebbe essere minore di 1, altrimenti il neurone perderebbe più "corrente" rispetto a quella che entra.

3.6 Frequenza di taglio

In questa sezione derivo la formula per calcolare la frequenza di taglio del neurone illustrato precedentemente. La formula che si ottiene è utile per determinare il valore di δ o τ più adatto al problema che si sta considerando.

È importante notare che questa formula produce una approssimazione, in quanto si considera l'ingresso del filtro come un ingresso autonomo, ma in realtà è composto da un ingresso ed una funzione non lineare dello stato stesso. Questa approssimazione, che è necessaria per ottenere una formula chiusa, è comunque

valida dal punto di vista empirico.

$$x(t+\delta) = \left(1 - \frac{\delta}{\tau}\right) x(t) + \frac{\delta}{\tau}\left(\lambda R r(t) + W^{fb} z(t) + W^{in} u(t)\right)$$

Per abbreviare la notazione sia $\gamma = \frac{\delta}{\tau}$, $y(t) = \lambda R r(t) + W^{fb} z(t) + W^{in} u(t)$.

$$x(t+\delta) = (1-\gamma)\, x(t) + \gamma y(t)$$

Applicando la trasformata Z si ottiene:

$$\begin{aligned} X(z) &= (1-\gamma)\, z^{-1} X(z) + \gamma Y(z) \\ X(z)\left(1 - (1-\gamma)\, z^{-1}\right) &= \gamma Y(z) \\ H(z) = \frac{X(z)}{Y(z)} &= \frac{\gamma}{1-(1-\gamma)\, z^{-1}} \end{aligned}$$

Per calcolare la frequenza di taglio si deve risolvere l'equazione $|H(e^{j\omega})| = \frac{1}{\sqrt{2}}$. Il modulo della risposta in frequenza è:

$$\begin{aligned} H(e^{j\omega}) &= \frac{\gamma}{1-(1-\gamma)\, e^{-j\omega}} \\ |H(e^{j\omega})| &= \frac{\gamma}{\sqrt{1 - 2\,(1-\gamma)\cos\omega + (1-\gamma)^2}} \end{aligned}$$

Elevando al quadrato si arriva:

$$\begin{aligned} \frac{\gamma^2}{1 - 2\,(1-\gamma)\cos\omega + (1-\gamma^2)} &= \frac{1}{2} \\ 2\gamma^2 &= 1 - (2-2\gamma)\cos\omega + 1 + \gamma^2 - 2\gamma \\ (2-2\gamma)\cos\omega &= 2 - \gamma^2 - 2\gamma \\ \cos\omega &= \frac{2 - 2\gamma - \gamma^2}{2-2\gamma} \end{aligned}$$

Da cui segue che[2]:

$$\omega = 2\pi f \delta = \arccos \frac{2 - 2\gamma - \gamma^2}{2-2\gamma}$$

[2]In questa formula è necessario δ poiché è stata utilizzata la versione discreta, quindi implicitamente la frequenza normalizzata. Viceversa, per indicare le frequenze dei segnali di ingresso ci si riferisce alla frequenza fisica del segnale, per cui δ fornisce la correzione necessaria.

Il valore massimo di frequenza rappresentabile, in definitiva, è influenzato dal passo di integrazione δ: è ovviamente impossibile rappresentare segnali con una frequenza maggiore rispetto alla frequenza di aggiornamento della rete; ma anche dal rapporto $\gamma = \frac{\delta}{\tau}$ che quantifica la memoria del neurone.

3.7 Addestramento

Un algoritmo di training è una procedura che cerca la corrispondenza ottimale tra l'uscita del reservoir e l'uscita desiderata. Il segnale obiettivo è solitamente una funzione sia del tempo sia degli ingressi alla rete, denotata con $f(t,u)$. L'addestramento consiste nel trovare i pesi W che minimizzano la funzione obiettivo:

$$E_{tot}(T) = \sum_{k=0}^{T} \left(z(k,u) - f(k,u)\right)^2 = \sum_{k=0}^{T} \left(W^{\intercal}\mathcal{R}_k(u) - f(k,u)\right)^2$$

La procedura di training, quindi, deve sfruttare l'andamento caotico del sistema non ancora allenato per trovare un punto in cui la funzione obiettivo ha un minimo, ovvero dove la rete produce il segnale desiderato. Si può infatti verificare che per valori di λ troppo piccoli, il training non ha successo, ma con valori di λ troppo grandi il feedback allenato non riesce a sopprimere il comportamento caotico, quindi la procedura di learning non converge. Esiste invece un intervallo di valori per λ per cui l'addestramento funziona, producendo un errore di approssimazione che decresce al crescere di λ. Questo intervallo è dipendente dal tipo di algoritmo scelto per il training.

Si inizia ora la descrizione di alcuni metodi di training che si possono utilizzare nel reservoir computing.

3.7.1 REGRESSIONE

Questo metodo è il primo tra quelli sviluppati. È la soluzione diretta del problema di minimizzazione descritto sopra. Si tratta di un algoritmo di addestramento di tipo batch, ma può essere esteso per funzionare in modalità online.

1. simulare la rete da 0 a t_0, dove t_0 è scelto in modo da ignorare i transitori iniziali.
2. simulare la rete per T passi, salvando l'uscita di ogni neurone del reservoir in una matrice S, di dimensioni $N \times T$. Contemporaneamente, salvare l'out-

put desiderato in un vettore riga f (o, se si hanno M uscite, in una matrice $M \times T$).

3. Calcolare, tramite i minimi quadrati, i pesi W che soddisfano l'equazione:

$$W = \arg\min_{w} \|w^{\mathsf{T}} S - f\|^2$$

In pratica questo problema può essere risolto dalla pseudo-inversa di Moore-Penrose: $W^{\mathsf{T}} = fS^{\dagger}$.

Considerando invece reti con feedback, si deve utilizzare il *teacher forcing*. Durante il training, si deve spezzare l'anello di feedback e sostituire $z(t)$ con l'output desiderato. Formalmente: $y \leftarrow \mathcal{R}(f(t), u)$. Quindi l'unità di readout è "forzata" con il valore obiettivo, e questo valore è distribuito nel reservoir tramite le sinapsi di retroazione.

Utilizzando il teacher forcing, l'algoritmo descritto sopra funziona correttamente, ma introduce problemi di stabilità. Infatti, l'algoritmo porterà alla riduzione dell'errore di training verso 0, senza alcun vincolo sulla dimensione dei parametri. Quindi aumentare molto un particolare W_j per avere un errore di training leggermente minore è un trade-off accettabile. Quando $W_j \gg 1$ significa che l'uscita del neurone j è molto amplificata per produrre la soluzione voluta. Di conseguenza se durante la fase di testing la traiettoria del neurone j devia leggermente (per esempio, per questioni numeriche) rispetto a quella su cui si è effettuato il training, si ottiene uno $z(t)$ molto differente da $f(t)$, e questo effetto è amplificato ad ogni passo di integrazione.

La soluzione proposta per questo problema è di aggiungere del rumore bianco ad ogni neurone nella rete durante il training, e poi applicare lo stesso algoritmo di sopra. Una seconda soluzione, invece, consiste nell'utilizzare la regressione di Tichonov, ovvero aggiungere un fattore di regolarizzazione ai termini della funzione obiettivo (è noto dalla letteratura sul *machine learning* che queste due soluzioni sono grossomodo equivalenti [4]). La funzione obiettivo diventa:

$$W = \arg\min_{w} \|w^{\mathsf{T}} S - f\|^2 + \beta \|w\|^2$$

che può essere risolta:

$$W^{\mathsf{T}} = f(t)\, S^{\mathsf{T}} \left(SS^{\mathsf{T}} + \beta I\right)^{-1}$$

La scelta del parametro β è solitamente effettuata con un algoritmo di tipo *line search* per minimizzare l'errore di test. Se $\beta = 0$ è anche chiamata soluzione di Wiener-Hopf[3].

Versione online

Esistono alcune modifiche di questo algoritmo per il funzionamento online, come ad esempio *Recursive Least Squares* (RLS) o *FORCE Learning* [45] che si basa su di una stima iterativa della matrice $(SS^\intercal + \beta I)^{-1}$ per trovare la soluzione di Tichonov al problema di minimizzazione. In particolare, l'algoritmo FORCE può essere schematizzato come segue:

1. Simulare la rete per un passo
2. Calcolare $e(t) = z(t) - f(t)$.
3. Stimare la matrice di correlazione del passo attuale tramite la formula:

$$P(t) = P(t-1) - \frac{P(t-1)r(t)r(t)^\intercal P(t-1)}{1 + r(t)^\intercal P(t-1)r(t)}$$

   con $P(0) = \frac{1}{\beta} I_N$.
4. Aggiornare i pesi con $W(t) = W(t-1) - e(t)P(t)r(t)$

Naturalmente, visto che il training con regressione è un metodo di learning supervisionato, presuppone che nel cervello di un organismo esista una componente (sia esso un neurone o una rete) che sia in grado di fornire il segnale da imparare. Questo ovviamente non spiega come la computazione possa "emergere" in una fase iniziale. Per questo motivo l'algoritmo di training presentato nella sezione successiva risulta essere molto interessante.

3.7.2 REGOLA EH

Questa regola è stata introdotta in [14], e benché sia poco potente, è di notevole interesse teorico perché ha una motivazione biologica, visto che sfrutta un meccanismo basato su una ricompensa. Più in dettaglio, al contrario della maggior parte delle altre regole di aggiornamento pesi, non usa informazioni globali sul segnale da approssimare, ma un singolo bit di informazione (la ricompensa), che indica se le performance attuali sono migliorate rispetto alla storia precedente.

[3] se inoltre la matrice S ha rango massimo la soluzione è uguale a quella calcolata tramite pseudo-inversa

In maniera più formale, si possono definire le performance correnti considerando tutti i readout:

$$P(t) = \sum_{i=1}^{M} (z_i(t) - f_i(t))^2$$

Dove $f_i(t)$ è l'output desiderato per il neurone di readout i. Utilizzo la convenzione che $\bar{X}(t) = \alpha \bar{X}(t-1) + (1-\alpha)X(t)$ rappresenta una versione filtrata (con un filtro passa-basso) del segnale.

$$M(t) = \begin{cases} 1 & \text{se } P(t) < \bar{P}(t) \\ 0 & \text{se } P(t) \geq \bar{P}(t) \end{cases}$$

Come già anticipato, $M(t)$ è un segnale che assume valore 1 se le performance attuali sono migliorate rispetto ai valori assunti in precedenza. La modifica dei pesi al neurone j dell'uscita i al tempo t è quindi data da:

$$W_{ij}(t) = W_{ij}(t-1) + \eta(t)\,[z_j(t) - \bar{z}_j(t)]\,M(t)\,r_i(t) \quad \forall\, 0 \leq i < N,\ 0 \leq j < M$$

Dove $\eta(t)$ è il learning rate, che può essere tenuto costante oppure diminuito nel tempo. In [14] si utilizza un learning rate che diminuisce nel tempo con la formula:

$$\eta(t) = \frac{\eta_0}{1 + \frac{t}{T}} \qquad \eta_0, T \text{ costanti opportune}$$

L'algoritmo che si utilizza per fare il training di una rete neurale caotica inizializzata con la procedura standard tramite questa regola è riportato sotto (algoritmo 2). La versione descritta, per semplicità, fa l'ipotesi che la rete abbia una singola uscita. Modificarlo per permettere il training di reti con M uscite è molto semplice, infatti basta considerare w come una matrice $N \times M$, e aggiornare le colonne separatamente.

Analizzando l'algoritmo si può notare che:

- I pesi del readout sono inizialmente tutti 0. Questo è necessario per evitare il feedback renda troppo caotica la dinamica del reservoir.
- La quantità $z_j(t) - \bar{z}_j(t)$ è una stima del rumore $\xi(t)$ sul readout j al tempo t, quindi il rumore è necessario per il funzionamento dell'algoritmo. In fase di testing, i rumori possono essere tolti sia dal readout sia dal reservoir. Solitamente nei nostri test viene eliminato il rumore sul readout, ma rimane quello sulle uscite del reservoir.

Algorithm 2 Addestra una rete caotica tramite la regola EH

function EH-TRAIN($\mathcal{R}$, time, η_0, $\alpha = 0.8$, $T = 20$)
  $w \leftarrow \mathbf{0}$
  $z(0) \leftarrow$ random initialization
  for $k \leftarrow [\delta, 2\delta, \ldots, \text{time})$ **do**
    $r \leftarrow \mathcal{R}(z(k-1), u(k))$ ▷ Simula la rete per un passo
    $z(k) \leftarrow w^\intercal r + \xi(k)$
    $p \leftarrow (z(k) - \bar{z})^2$ ▷ Calcola le performance attuali
    $\bar{p} \leftarrow \alpha\bar{p} + (1-\alpha)p$
    $\bar{z} \leftarrow \alpha\bar{z} + (1-\alpha)z(k)$
    if $p < \bar{p}$ **then** ▷ Le performance sono migliorate?
      $\eta \leftarrow \eta_0 / \left(1 - \frac{k}{T}\right)$
      for $i \leftarrow [0 \ldots N)$ **do**
        $w_i \leftarrow w_i + \eta(z - \bar{z})r_i$
  return w

- Il metodo non fa particolari assunzioni sul modello di neurone utilizzato. Nonostante sia stato presentato per il modello 3.1, può essere utilizzato anche in altri casi. L'unica cosa necessaria è la presenza del feedback.
- La quantità di informazione trasmessa per il training è minimale. Si può congetturare che se si diminuisse ancora il contenuto di informazione allora l'addestramento non sarebbe possibile.
- Il valore di $\alpha = 1 - \frac{\delta}{t_{avg}}$ è un ulteriore parametro da impostare. In [14] si suggerisce un $t_{avg} = 5\delta$, quindi $\alpha = 0.8$. Il valore rappresenta una ulteriore operazione di filtraggio, quindi in generale dipende dal segnale.
- In modo qualitativo, il sistema deve essere abbastanza caotico nelle fasi iniziali, altrimenti l'uscita del reservoir si annullerebbe prima che il learning inizi a modificare i pesi W.

3.8 Altri modelli di neurone discussi

Nel reservoir computing non si ha alcun vincolo sul tipo di neurone che popola il reservoir, quindi molte varianti sono possibili. Ne discuto solo 2, di cui una molto teorica e una che, al contrario, è spesso usata.

Altri modelli abbastanza comuni sono i cosiddetti *Leaky Integrate and Fire*, che sono essenzialmente dei normali Leaky Integrator (come il modello 3.1) con una funzione di attivazione binaria e che possono assumere valore "alto" solo un numero limitato di volte per periodo di tempo. Questo tipo di neurone è più utile in casi di classificazione, quindi non molto interessante ai nostri scopi.

3.8.1 NEURONI LINEARI

Questo modello di neurone è molto semplice e non presenta comportamenti caotici. È caratterizzato dall'equazione:

$$x(k+1) = Rx(k) + W^{fb}z(k) + W^{in}u(k)$$

Viene usato principalmente perché permette di raggiungere risultati teorici, ma l'uso pratico è limitato. Ammette una soluzione in forma chiusa:

$$x(k) = \left(W^{eff}\right)^k x_0 + \sum_{i=0}^{k-1} \left(W^{eff}\right)^{k-i-1} W^{in}u(i)$$

Questo modello è, per esempio, usato in [12]: si tratta di un lavoro essenzialmente teorico in cui gli autori sviluppano una formula chiusa per predire le performance di neuroni lineari su di un task qualunque. Naturalmente questo risultato teorico è ottenibile in quanto il neurone è molto semplice e come tale manca di applicazioni pratiche.

3.8.2 VARIANTE LEAKY INTEGRATOR

Questo modello appartiene alla categoria dei Leaky Integrator, come il modello 3.1, ed è il più usato nel campo delle Echo State Network. L'equazione caratteristica è molto simile a quella di 3.1:

$$\gamma\dot{x}(t) = -\alpha x(t) + \tanh\left(Rx(t) + W^{fb}z(t) + W^{in}u(t)\right)$$

Oppure, in tempo discreto:

$$x(k+1) = \left(1 - \frac{\alpha\delta}{\gamma}\right)x(k) + \frac{\delta}{\gamma}\tanh\left(Rx(k) + W^{fb}z(k) + W^{in}u(k)\right) \tag{3.4}$$

Tramite alcune trasformazioni algebriche si può trovare una corrispondenza tra i due modelli. Moltiplicando entrambi i membri di 3.4 per R, si ottiene:

$$Rx(k+1) = \left(1 - \frac{\alpha\delta}{\gamma}\right) Rx(k) + \frac{\delta}{\gamma} R \tanh\left(Rx(k) + W^{fb} z(k) + W^{in} u(k)\right)$$

Si introduce un nuovo vettore di stato $\bar{x}(k) = Rx(k) + W^{fb} z(k) + W^{in} u(k)$:

$$\begin{aligned}\bar{x}(k+1) - W^{fb} z(k+1) - W^{in} u(k+1) = \\ \left(1 - \frac{\alpha\delta}{\gamma}\right)\left(\bar{x}(k) - W^{fb} z(k) - W^{in} u(k)\right) + \frac{\delta}{\gamma} R \tanh\left(\bar{x}(k)\right)\end{aligned}$$

$$\begin{aligned}\bar{x}(k+1) = \left(1 - \frac{\alpha\delta}{\gamma}\right)\bar{x}(k) + \frac{\delta}{\gamma} R \tanh\left(\bar{x}(k)\right) + \\ W^{fb}\left(\frac{\delta}{\gamma} z(k+1) - \left(1 - \frac{\delta\alpha}{\gamma}\right) z(k)\right) + \\ W^{in}\left(\frac{\delta}{\gamma} u(k+1) - \left(1 - \frac{\delta\alpha}{\gamma}\right) u(k)\right)\end{aligned}$$

Introducendo $\bar{u}(k) = \frac{\gamma}{\delta}\left(\frac{\delta}{\gamma} u(k+1) - \left(1 - \frac{\delta\alpha}{\gamma} u(k)\right)\right)$ e definendo analogamente $\bar{z}(k)$ si può riscrivere:

$$\bar{x}(k+1) = \left(1 - \frac{\alpha\delta}{\gamma}\right)\bar{x}(k) + \frac{\delta}{\gamma}\left(R \tanh\left(\bar{x}(k)\right) + W^{fb}\bar{z}(k) + W^{in}\bar{u}(k)\right)$$

Si vede facilmente che l'equazione sopra è molto simile all'equazione di aggiornamento per lo stato che si aveva in 3.1, a parte il parametro α. I due sistemi non sono comunque equivalenti, anche se si avesse $\alpha = 1$: ci sono infatti anche delle operazioni di filtraggio su input, feedback e stato interno. Questo modello ha due parametri che variano la caoticità: α e il raggio spettrale della matrice R (che è analogo al fattore λ del modello 3.1).

Questo modello è molto usato poiché le Echo State Netwoks sono state introdotte con esso; la maggior parte dei lavori che li utilizzano (per esempio [18] o [49]) si riferiscono a dei task di predizione di serie temporali.

3.9 Misura delle performance

È già stato evidenziato, seppur in maniera informale, che affinché la rete riesca ad imparare un segnale sia importante che essa dimostri una variabilità di dinamiche. In generale è complicato stabilire se una rete ricorrente potrà imparare o meno un certo pattern. In [18] si introduce il concetto di reservoir con "dinamiche ricche", che però non è stato quantificato. Come già anticipato, una rete caotica possiede questa proprietà, in senso qualitativo.

Un metodo sicuramente funzionante per valutare un reservoir è fare il training e misurare successivamente le performance, su un insieme di validazione, come da procedura usuale nel machine learning. Altri autori ([31]) propongono misure, basate sull'entropia, che stimano la diversità delle dinamiche presenti nel reservoir prima del training. Dalle nostre prove, comunque, questi metodi si sono dimostrati subottimali.

3.9.1 ESPONENTE DI LYAPUNOV

Siccome l'evoluzione di un sistema dinamico caotico mostra dipendenze dalle condizioni iniziali, piccole differenze di valori iniziali per due sistemi altrimenti identici devono persistere (od amplificarsi). Viceversa, se il sistema è ordinato, ci si aspetta che due condizioni iniziali leggermente differenti non introducano grosse differenze nell'evoluzione degli stati e che gli effetti della diversa inizializzazione tendano ad azzerarsi.

L'esponente di Lyapunov è il metodo principale per caratterizzare la presenza di evoluzioni caotiche in un sistema dinamico e cattura in modo formale questa intuizione.

Sia $\mathrm{d}(k) = \|x_2(k) - x_1(k)\|$ la distanza tra gli stati del sistema originale e del sistema perturbato al tempo k. Considerando il sistema linearizzato, la distanza si può approssimare con:

$$\mathrm{d}(k) = e^{\lambda k}\,\mathrm{d}(0)$$

L'esponente di Lyapunov λ è quindi definito nel modo seguente:

$$\lambda = \lim_{k\to\infty} \frac{1}{k} \log \frac{\mathrm{d}(k)}{\mathrm{d}(0)}$$

Se $\lambda > 0$ il sistema è caotico, altrimenti è ordinato (quando $\lambda \approx 0$ si è all'"orlo del caos"). Siccome questa è una quantità asintotica deve essere stimata. Per farlo

in modo numericamente stabile si può utilizzare l'algoritmo descritto in [41].

Algorithm 3 Calcolo dell'esponente di Lyapunov

function APPROXIMATE-LYAPUNOV($\mathcal{R}$)
  Simula $\mathcal{R}$ per K passi per eliminare i transitori iniziali
  $r_0 \leftarrow$ output corrente del $\mathcal{R}$
  $r_1 \leftarrow$ output corrente del $\mathcal{R}$
  $r_1 \leftarrow r_1 + d_0$ ▷ Aggiungi ad r_1 una piccola perturbazione
  for $i \leftarrow [0 \dots N)$ **do**
    $r_0 \leftarrow \mathcal{R}(\cdots)$ ▷ Simula un passo di entrambe le reti
    $r_1 \leftarrow \mathcal{R}(\cdots)$
    $d \leftarrow \|r_1 - r_0\|$ ▷ Calcola la divergenza tra gli stati
    $\lambda_i \leftarrow \log(d/d_0)$
    $r_1 \leftarrow (d_0/d)(r_1 - r_0)$ ▷ Normalizza per evitare overflow
  return media di λ_i

I punti importanti di questo algoritmo sono i seguenti:

- Il valore minimo di K necessario per eliminare i transitori. Valori di K di circa 1000 sono sufficienti, ma non è una scelta critica per la precisione dell'algoritmo.
- Il valore di d_0, che deve essere piccolo, ma sufficientemente grande affinché la sua influenza sia misurabile con la precisione numerica di un calcolatore. In [41] si consiglia un valore nell'ordine di $\sqrt{\epsilon} \approx 10^{-8}$ (lavorando in *double precision*).
- La condizione per terminare le iterazioni non è molto importante: si può scegliere se fare un numero fisso di iterazioni oppure se terminare quando la $\lambda_{i+1} \approx \lambda_i$ al livello di precisione richiesto.
- Il passo di normalizzazione serve per fare in modo che al tempo k, x_1 abbia ancora distanza d_0 da x_0, preservando la direzione.

Per valutare questa quantità si può considerare sia la rete con teacher-forcing sia senza. Senza teacher-forcing (con un readout casuale) quantifica le dinamiche del sistema, con teacher-forcing, invece, è più indicativa della resistenza della rete al rumore.

3.9.2 MISURA P

Questa misura è stata ideata basandosi su una possibile formalizzazione della nozione intuitiva di "dinamiche ricche". In particolare, essa si basa sulla *Analisi delle Componenti Principali* ed è definita come una quantità che misuri la distribuzione delle componenti principali in cui consiste l'evoluzione del reservoir.

Si può calcolare la matrice di covarianza $\Sigma = \{\sigma_{ij}\}$ di $r(t)$:

$$\sigma_{ij} = \frac{1}{N} \sum_{n=1}^{N} (r_i(n) - \mu_i)(r_j(n) - \mu_j)$$

L'autovettore corrispondente all'autovalore più alto di questa matrice è la componente principale della traiettoria. Ciascun autovalore contribuisce in maniera proporzionale alla varianza totale, pertanto si possono normalizzare in modo tale che la somma sia 1.

$$\lambda'_i = \frac{\lambda_i}{L} \qquad \text{dove } L = \sum_{i=0}^{N} \lambda_i$$

$$\rho = -\frac{1}{\log N} \sum_{i=0}^{N} \lambda'_i \log \lambda'_i$$

Dalla definizione seguono le proprietà:

- ρ è invariante rispetto al fattore di scala dei dati: l'effetto è cancellato dalla normalizzazione degli autovalori.
- ρ varia tra 0 e 1. Si può verificare facilmente (siccome è l'entropia della distribuzione degli autovalori normalizzati) che il vale 1 per $\Lambda' = \{\frac{1}{N}, \ldots, \frac{1}{N}\}$, mentre si ottiene il minimo quando un solo autovalore vale 1.

Un valore grande di ρ implica una piccola correlazione fra le singole uscite del reservoir, quindi un sistema con dinamiche molto ricche, al contrario, al diminuire di ρ si ottiene un sistema con poca variabilità.

Ci si può aspettare, quindi, che esista un intervallo di valori di ρ per cui il training abbia successo con performance grossomodo decrescenti al crescere di ρ. Per valori troppo piccoli, non ci sono abbastanza dinamiche da sfruttare per il training; viceversa, per valori troppo grandi le dinamiche sono difficilmente controllabili dalla procedura di training.

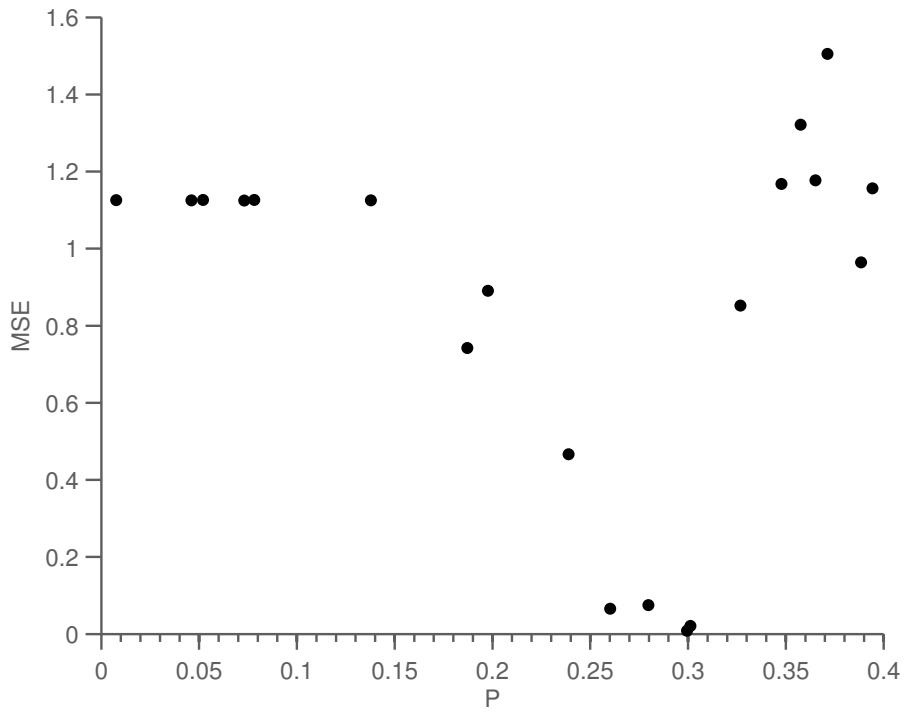


Figura 3.2: Errore di riproduzione del segnale in funzione della misura p sul problema in sezione 5.2. Il training è effettuato con la regola EH, con la configurazione in sezione 5.2.2. Le reti sono ottenute facendo variare λ tra 0 e 3.

3.10 Note sulla scelta dei parametri nel RC

Come è già stato visto, il reservoir computing ha diversi parametri da configurare. Si nota che non si può capire facilmente dal risultato quale sia il parametro che necessita di modifiche per migliorare il funzionamento della rete e inoltre ciascuno di essi influenza le performance in maniera sensibile. Per semplificare la progettazione sono state trovate in maniera empirica delle regole che aiutino ad identificare dei punti di partenza adatti per questi valori. Per i parametri mi riferisco al modello 3.1, considerazioni del tutto simili si possono fare per i parametri analoghi della variante presentata nella sezione 3.8.2.

Il modo più pragmatico per misurare l'errore di una rete neurale è di fare il training e misurare l'errore. Ovviamente questo non è sempre possibile, perché potrebbe per esempio usare troppo tempo. In questi casi si può ricorrere alle misure elencate in precedenza, la P o l'esponente di Lyapunov, che empiricamente si nota che riescono a fornire delle idee sulla potenza espressiva della rete.

3.10.1 DIMENSIONE DEL RESERVOIR

Questo parametro è abbastanza importante, perché influenza anche il tempo di simulazione della rete. In generale si consiglia di considerare il reservoir più grande possibile. Valori buoni iniziali sono attorno ai 700 neuroni, che riescono a garantire ottimi tempi di simulazione.

3.10.2 SPARSITÀ

La sparsità è forse ancora più importante della dimensione per quanto riguarda le performance della simulazione: influenza infatti la memoria utilizzata, e ovviamente l'accesso a memoria è una operazione piuttosto lenta. Nelle pubblicazioni iniziali riguardanti le Echo State Network si raccomanda una connettività molto sparsa, ed empiricamente si nota che matrici sparse tendono a produrre risultati leggermente migliori. Il fattore di sparsità solitamente scelto è il 10%; valori fino al 20% producono buoni risultati. Coefficienti di sparsità più alti solitamente non servono a molto e rallentano significativamente la simulazione.

Una estensione molto semplice che si può effettuare per quanto riguarda la scelta di questo parametro è la seguente: connettere ogni nodo ad un numero fisso di altri nodi, in maniera indipendente dalla dimensione del reservoir (quindi il costo non è più quadratico ma lineare, come si vede in sezione 4.3.1). Per esempio si può scegliere di connettere ogni nodo a 10 altri nodi.

3.10.3 FATTORE DI CAOS

La scelta di λ in 3.1 è abbastanza importante. In generale può essere visto come un fattore che permette di rendere l'output più vario. In semplici esperimenti si nota che alzare questo parametro è un modo poco dispendioso di rendere più potente la rete, senza dover aggiungere neuroni. Ovviamente si vede anche che esiste un valore, che, se superato, peggiora le performance (che dipende sia dal problema in considerazione sia dal valore degli altri parametri, quindi è difficile da stimare).

In un task senza input un buon valore iniziale è 1.5 e può essere modificato in alcuni casi (vedi figura 3.3):

- se è troppo basso si può vedere, durante il training, che il sistema non riesce a riprodurre il segnale, nemmeno nei primi istanti di tempo del training;

- se invece è troppo alto, solitamente la rete, durante l'addestramento, riesce a seguire il segnale obiettivo per diversi periodi, ma ad un certo punto l'addestramento non riesce più a gestire la caoticità del sistema e di conseguenza il segnale in uscita non segue più il segnale da approssimare.

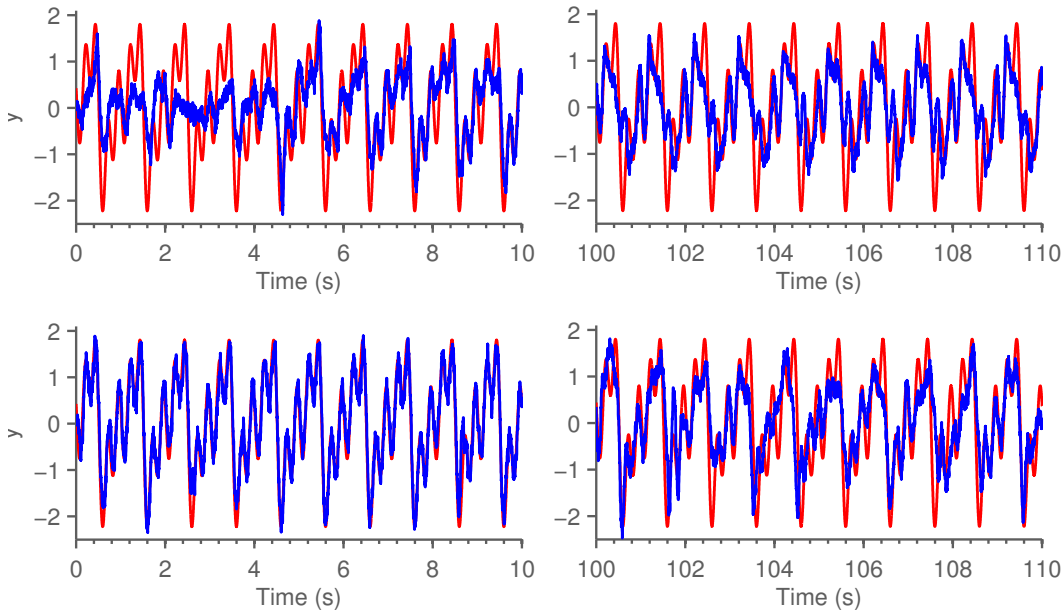


Figura 3.3: Effetto della scelta di λ sullo stessa rete con lo stesso segnale da approssimare (problema illustrato in sezione 5.2, configurazione come in 5.2.2). In rosso il segnale obiettivo, in blu l'uscita della rete durante il training. Nella parte superiore: l'effetto di λ troppo basso ($\lambda = 1.2$) si nota già nei primi secondi, le dinamiche non sono abbastanza ricche per il training. Nella parte inferiore: λ troppo alto ($\lambda = 1.9$). All'inizio il segnale è riprodotto correttamente, ma dopo il readout allenato non riesce più a tenere sotto controllo il caos del sistema, che causa una divergenza dal segnale da approssimare.

In un task con input invece, si può scegliere un valore più basso (per esempio, 1.2) se l'uscita desiderata ha un andamento che "segue" l'ingresso (se per esempio si tratta di un task in cui sia l'input sia l'output devono assumere valori logici). Se invece l'input permette di scegliere tra uscite qualitativamente diverse, è meglio che λ sia grande (per esempio 1.7).

3.10.4 TEMPO DI MEMBRANA

Il parametro τ è già stato oggetto di una analisi in precedenza, si è ovvero calcolata la frequenza di taglio del neurone, che è appunto collegata al leak ra-

te. Quindi, una volta nota la frequenza massima del segnale da imparare, si può calcolare un valore di τ appropriato. Ovviamente questo non è sempre possibile, anche perché come è già stato evidenziato, si tratta di una approssimazione. Naturalmente l'intuizione che τ grande implica un segnale più lento e viceversa per τ piccolo è sempre valida, e fornisce spesso un valore iniziale valido. Una volta impostato il valore iniziale, si può effettuare un regolazione più fine guardando come risponde la rete durante il training rispetto al segnale obiettivo.

3.10.5 MODULO DEI DATI

Un procedimento che si adotta spesso quando si lavora con le reti neurali è la normalizzazione dei dati, portarli ovvero in uno stesso intervallo di riferimento. Questo passo è necessario anche nelle CNN; inoltre è anche utile che l'input e le uscite abbiano modulo paragonabile: se l'uscita fosse molto più alta dell'input, quando essa viene reinserita nella rete come feedback, tenderebbe a nascondere il nuovo valore dell'ingresso. Inoltre, siccome la regola EH usa del rumore di esplorazione (uniformemente distribuito) di ampiezza 0.5 ([14]), si deve avere un segnale di modulo abbastanza grande, altrimenti il rumore sovrasta il segnale da imparare.

Nelle prove effettuate si considerano i segnali da approssimare normalizzati nell'intervallo $[-2, 2]$. Questo intervallo è stato scelto dopo alcune prove sperimentali siccome produceva risultati leggermente migliori rispetto ad altre possibilità.

CAPITOLO 4

Analisi ed implementazione

4.1 Introduzione

Come già anticipato, lo scopo di questa tesi è confrontare, sia da un lato funzionale, sia dal punto di vista dell'efficienza i Central Pattern Generator e le reti neurali caotiche. Inizio questa sezione, quindi, con una analisi della complessità computazionale di ambedue i modelli presentati.

Per confrontare invece le funzionalità si deve realizzare un controllore per la locomozione, ovvero un sistema che riesca a generare le traiettorie volute. Ho quindi dettagliato, nel prosieguo della sezione, l'interfaccia fornita dalla mia implementazione di simulatore per reti neurali caotiche.

Questa libreria fornisce tutte le funzionalità necessarie per simulare una rete neurale caotica ed è stata sviluppata per essere facilmente estendibile in caso di necessità (per esempio, se si volesse modificare il tipo di neuroni o gli algoritmi di training). Le motivazioni che hanno reso necessaria una implementazione totalmente nuova sono discusse nella sezione 4.6.

4.2 Complessità computazionale CPG

Tutti i modelli di CPG oggetto di studi per la robotica sono espressi come sistemi dinamici in tempo continuo, quindi l'analisi della complessità computazionale presentata qui è solitamente valida anche per essi, a meno delle costanti.

Nel nostro caso, ogni oscillatore è dotato di 5 variabili di stato: $x, y, \omega, \alpha, \phi$. Ognuna di esse è scalare, quindi tutte le operazioni per effettuare l'aggiornamento di questi valori sono a complessità costante; di conseguenza anche l'intero oscillatore ha complessità $\Theta(1)$ per ogni passo di simulazione.

Siccome per generare un grado di libertà sono necessari N oscillatori, la complessità computazionale per calcolare una uscita è $\Theta(N)$, anche in questo caso, ad ogni step di integrazione. In aggiunta si generano M uscite, che quindi totalizzano $\Theta(NM)$. In definitiva, simulare una rete di M Central Pattern Generators ciascuno formato da N oscillatori per T istanti di tempo ha complessità $\Theta(NMT)$.

Durante la fase di testing, non è più necessario aggiornare i valori di ω ed α, quindi le derivate si annullano. Le costanti diventano più basse, ma la complessità in notazione Θ-grande rimane uguale.

Fourier CPG

L'algoritmo FourierCPG necessita di un vettore di dati in training di dimensione T e del numero N di oscillatori. Per calcolare i parametri del Central Pattern Generator, esso effettua una trasformata di Fourier veloce, che ha complessità $\Theta(T \log T)$. Poi, per ogni oscillatore, estrae il massimo delle ampiezze nella trasformata (quindi complessità $\Theta(T)$). L'algoritmo è molto efficiente, infatti in totale ha complessità $\Theta(T \log T + NT)$ per ogni CPG (da cui segue $\Theta(MT(\log T + N))$ in totale per M CPG).

4.3 Complessità computazionale CNN

In questa sezione analizzo la complessità asintotica dei vari algoritmi utilizzati nelle reti neurali caotiche. L'analisi è più complessa rispetto a quella dei Central Pattern Generators, quindi è articolata dividendo la varie parti che compongono una rete ricorrente.

Nella letteratura esistono diverse analisi empiriche (per esempio [24] o [43]), ma le analisi teoriche si riferiscono principalmente agli algoritmi di training di reti ricorrenti ([1, 44, 13])

4.3.1 RESERVOIR

Per descrivere l'implementazione del reservoir utilizzo il modello 3.1; altre tipologie di reservoir sono molto simili e le considerazioni si possono ripetere quasi identiche. Per simulare il reservoir si devono implementare le prime due equazioni di 3.1. Analizzando la prima:

- 1 prodotto tra scalare e vettore, che ha come complessità $\Theta(N)$;

- un prodotto tra matrice sparsa e vettore. Questa operazione è lineare nel numero di elementi diversi da zero. Una matrice $N \times N$, con fattore di sparsità p possiede, in media, pN^2 elementi non nulli: l'operazione ha complessità $\Theta\left(pN^2\right) = \Theta\left(N^2\right)$, è ovvero quadratica rispetto alla dimensione del reservoir;
- due prodotti tra matrice e vettore, che hanno complessità pari rispettivamente a $\Theta(NM)$ e $\Theta(NL)$. Il vettore è in un caso l'input al sistema oppure il feedback, quindi sono in entrambi i casi piccoli (e di dimensione costante rispetto al problema, quindi si può considerare l'operazione come lineare).

Dato lo stato, calcolare il firing rate r è una operazione che dipende in maniera lineare dalla complessità del calcolo della funzione di attivazione scelta.

In definitiva, un passo di integrazione del reservoir può essere simulato con una complessità asintotica di $\Theta\left(N^2\right)$. Le costanti, in maniera sperimentale, sono piuttosto piccole, quindi si ha una zona molto ampia di valori di N in cui la crescita del tempo di esecuzione è quasi lineare. Si può vedere un risultato empirico che porta a questa conclusione in figura 4.1.

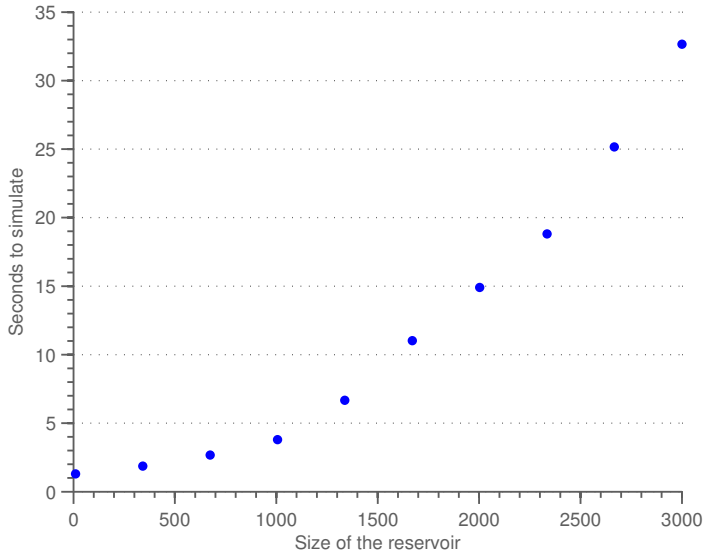


Figura 4.1: Rappresentazione del tempo d'esecuzione di 10 secondi di simulazione (con $\delta = 1$ ms) in funzione della dimensione del reservoir. Si vede che la curva è quadratica, ma la crescita è piuttosto lenta.

Anche la procedura di inizializzazione del reservoir ha un costo abbastanza importante, infatti si deve generare una matrice casuale ($\Theta\left(pN^2\right)$), calcolarne il

raggio spettrale ($\Theta\left(N^2K\right)$, dove K è il numero di iterazioni per il metodo delle potenze) e scalarla (moltiplicazione tra scalare e matrice sparsa, quindi $\Theta\left(N^2\right)$).

4.3.2 READOUT

Il readout, come già visto anche nella descrizione più teorica, è molto semplice, appositamente ideato per essere molto veloce da calcolare. La computazione delle uscite è infatti una operazione $\Theta(NM)$.

4.3.3 ALGORITMI DI TRAINING

Regressione

In questo caso analizzo in maniera dettagliata la versione con il fattore di regolarizzazione.

$$W^\intercal = f(t)\, S^\intercal \left(SS^\intercal + \beta I\right)^{-1}$$

Detto M il numero delle uscite, N il numero di neuroni nel reservoir e T il tempo di training (con $T \gg N$), la matrice S è $N \times T$ e $f(t)$ è una matrice $M \times T$. Formalmente basterebbe avere $T = N$, ma sperimentalmente si nota che è necessario almeno un fattore 10 tra le dimensioni per ottenere buoni risultati.

Per trovare i pesi in questo algoritmo è necessario invertire una matrice, che è una operazione spesso numericamente instabile e poco efficiente. Si può trasformare nell'equivalente sistema $W^\intercal(SS^\intercal + \beta I) = f(t)S^\intercal$, che può essere risolto con algoritmi che non soffrono dei problemi descritti. Le operazioni da effettuare sono:

- La moltiplicazione $SS^\intercal$, con complessità $\Theta\left(N^2T\right)$;
- La somma $SS^\intercal + \beta I$, che ha complessità $\Theta(N)$, siccome I è una matrice diagonale;
- La moltiplicazione $f(t)S^\intercal$, $\Theta(MNT)$;
- La risoluzione del sistema può essere fatta tramite fattorizzazione di Cholesky (o LU), quindi è $\Theta\left(N^3\right)$ [34].

Asintoticamente, siccome $T \gg N$, $\Theta\left(N^2T\right)$ domina $\Theta\left(N^3\right)$. In generale, quindi, questo metodo non è molto efficiente, specialmente per reservoir grandi, ma bastano pochi dati (e quindi pochi passi di simulazione) per ottenere un training con errori piccoli.

La complessità della soluzione calcolata tramite pseudo-inversa è anch'essa $\Theta\left(N^2T\right)$. Infatti, per trovare il vettore dei pesi in questo modo, si deve calcolare la pseudo-inversa di S (per esempio, tramite *singular value decomposition*, SVD), che è una operazione $\Theta\left(N^2T\right)$ ([6]), ed effettuare il prodotto $fS^{\dagger}$, che $\in \Theta(MNT)$.

Anche il FORCE learning ha complessità $\Theta\left(N^2T\right)$, ma è divisa tra le varie iterazioni (ogni iterazione ha complessità $\Theta\left(N^2\right)$). Non analizzo in dettaglio questo algoritmo per brevità, ma si può vedere che consiste in operazioni tra una matrice $N \times N$ e un vettore di dimensione N. La separazione del procedimento nelle varie iterazioni causa un numero di accessi a memoria molto maggiore rispetto alla versione offline, quindi il tempo di esecuzione ne risente in maniera negativa (come si verifica sperimentalmente negli esperimenti nei capitoli 5 e 6).

Regola EH

Questo metodo è computazionalmente piuttosto leggero, infatti utilizza principalmente operazioni tra scalari e tre prodotti tra matrice e vettore. Detto M il numero di uscite (z è un vettore di M elementi), analizzando l'algoritmo 2 si ottiene che ad ogni passo di simulazione servono:

- $\Theta(M)$ operazioni per il calcolo di p
- $\Theta(M)$ per calcolare $\bar{z}$
- Aggiornare i pesi ha complessità $\Theta(MN)$

La complessità è $\Theta(MN)$, senza però considerare le operazioni necessarie per simulare il reservoir (già derivate nella sezione 4.3.1), che totalizzano complessità $\Theta\left(N^2\right)$, che quindi domina $\Theta(MN)$.

Le operazioni restanti sono tutte tra scalari, quindi a complessità costante. In totale, un passo di addestramento tramite la regola EH ha complessità $\Theta\left(N^2\right)$. La complessità di questo metodo non è molto elevata, ma deve essere moltiplicata per il numero di passi di training.

4.4 Considerazioni iniziali per CPG

Prima di iniziare una implementazione dei modelli di Central Pattern Generator, si è cercato di stabilire se esistesse già una libreria per la simulazione di questi modelli. Visto però che la maggior parte delle applicazioni dei CPG è ancora accademica, non è stato trovato alcun pacchetto software o libreria che fornisse le funzionalità necessarie. Questa carenza è anche dovuta alla caratterizzazione dei

modelli utilizzati: infatti una libreria per implementare i CPG sarebbe poco più che un risolutore di equazioni differenziali.

4.5 Implementazione CPG

Viste le considerazioni effettuate, per implementare i CPG si è scelta una soluzione ad hoc: la possibilità di implementare una libreria apposita è stata valutata ma scartata perché sarebbe poco utile, in quanto nessuna parte dell'implementazione sarebbe veramente riutilizzabile in altri casi. Per esempio, cambiando il modello di oscillatore adottato si dovrebbe utilizzare un metodo diverso per trovare i parametri. La libreria sarebbe quindi una collezione di modelli unito all'implementazione di un metodo numerico per evolvere la simulazione.

4.5.1 FOURIER CPG

L'algoritmo per l'addestramento FourierCPG è stato implementato nella funzione `fouriercpg_train(data, N, dt)`, in cui i parametri:

- `data` è il vettore che raccoglie i dati di training;
- `N` rappresenta il numero di oscillatori da utilizzare;
- `dt` è l'intervallo di tempo tra un dato e il successivo.

La funzione ritorna `w`, `alpha` e `phi`, ovvero i parametri necessari al Central Pattern Generator.

4.6 Considerazioni iniziali per CNN

Anche in questo caso, prima di decidere se procedere ad una implementazione ex novo di una libreria per la simulazione delle reti neurali caotiche, si sono analizzate alcune librerie già esistenti per la simulazione di reti neurali. In particolare, si sono trovate due librerie dedicate esclusivamente al reservoir computing, ed un numero più elevato (circa 6, per vari linguaggi di programmazione) per il caso più generale di rete ricorrente.

- CSIM (e la versione successiva, PCSIM), sviluppata da Maass. Non si è potuta utilizzare perché implementa principalmente Liquid Computing. È possibile estenderla, ma l'architettura della libreria che considera la rete come un oggetto globale rende difficile l'utilizzo con MATLAB e causa diversi

bug. Entrambe le versioni sono piuttosto vecchie (rispettivamente 2006 e 2009) e non più mantenute.

- aureservoir, sviluppata da Holzmann. È stata sviluppata nell'ambito delle Echo State Network, quindi anche questa non fornisce direttamente i comportamenti caotici voluti. È inoltre piuttosto complicato aggiungere nuovi modelli di neurone. Anche questa libreria non è più mantenuta e la sua ultima versione risale al 2008.
- rnn, per Lua, basata su Torch7. Al contrario delle prime due non è sviluppata in ambito accademico. Non implementa direttamente le Echo State Network, ma è possibile codificarle sfruttando le RNN. Esistono alcune altre librerie simili a questa, sviluppate per altri linguaggi di programmazione. Visto che si basa su Torch7 è molto efficiente e permette di delegare le computazioni a schede grafiche. Esistono altre librerie simili a questa per diversi linguaggi di programmazione come theano per Python.
- Esistono inoltre molte piccole implementazioni di Echo State Network, pensate principalmente per scopi didattici.

Come si vede, non esistono moltissime librerie per il reservoir computing: la maggior parte delle implementazioni esistenti sono accademiche oppure adhoc per il problema che si vuole risolvere. Siccome non esistono librerie di buona qualità che forniscono le reti neurali caotiche e l'estensione di librerie esistenti per simulare reti neurali sarebbe risultata piuttosto complicata, si è scelto di procedere con l'implementazione di una soluzione da zero.

L'architettura della soluzione sviluppata è stata ispirata principalmente da rnn e librerie simili, che a loro volta si basano su standard de facto della comunità scientifica. Si procede a "blocchi funzionali": vari blocchi implementano i diversi algoritmi necessari, e l'output di uno viene letto come ingresso dei blocchi successivi. Questo idea è inoltre abbastanza in linea con l'idea di reservoir computing, infatti, al contrario delle RNN, i dati fluiscono solo in avanti.

La visione di alto livello dell'implementazione è simile a quella di rnn, ovvero "a strati", che è anche suggerita dalla separazione fra readout e reservoir. Ogni componente è una classe che deve fornire il singolo metodo `simulate()` che calcola l'output dello strato corrente, avanzando quindi di un passo la simulazione.

Nelle prossime sezioni illustrerò le varie componenti implementate nella libreria, fornendo le motivazioni delle scelte effettuate.

4.7 Reservoir

Per implementare il reservoir si è adoperata una scelta piuttosto differente rispetto alle implementazioni elencate sopra. Infatti, queste seguono due strade completamente opposte: aureservoir e rnn non implementano il reservoir separatamente, mentre CSIM implementa i singoli neuroni che lo compongono. Entrambe queste opzioni hanno pregi e difetti: la prima rende le operazioni molto efficienti (infatti non si deve implementare il readout), ma poco generali (non si può cambiare facilmente il tipo di neurone nel reservoir); mentre la seconda è molto lenta (e utilizza più memoria) ma permette di avere neuroni diversi all'interno dello stesso liquido. Nella mia implementazione ho scelto la via intermedia, che cerca di combinare i pregi di entrambe senza incorrere in grossi difetti. Si implementa quindi il concetto di reservoir, che permette di utilizzare la versione matriciale della formula di evoluzione vista in precedenza, in modo che i calcoli siano effettuati in maniera efficiente. Allo stesso tempo, questa scelta permette di modificare la tipologia di neuroni da utilizzare.

Il reservoir implementa l'interfaccia `NodeInterface`, che prescrive di implementare i metodi: `simulate`, `simulateNoInput` e `clone`:

- Il metodo `simulate(input, feedback)` implementa l'equazione caratteristica del reservoir e avanza la simulazione di un passo.
- Anche il metodo `simulateNoInput(feedback)` implementa l'equazione dello stato, ma si deve usare quando il sistema è autonomo.
- `clone()` ritorna una copia del reservoir, con la stessa topologia e lo stesso stato corrente.

Le matrici dei pesi vengono inizializzate con i valori standard (riportati in sezione 3.3.1) nel costruttore, così come lo stato iniziale, e ovviamente non possono essere modificate.

4.7.1 LEAKY INTEGRATOR

Il reservoir per il modello 3.1 viene costruito dalla classe `LeakyReservoirNodes` specificando nel costruttore i parametri:

- dimensione dell'ingresso
- dimensione del feedback
- numero di neuroni

- sparsità
- tempo di membrana τ
- coefficiente caotico λ
- rumore su $r(t)$
- il passo di integrazione

4.7.2 VARIANTE LEAKY INTEGRATOR

La classe che implementa questo modello si chiama `VarLeakyReservoirNodes` e il costruttore accetta i seguenti parametri:

- dimensione dell'ingresso
- dimensione del feedback
- numero di neuroni
- sparsità
- costante α
- tempo di membrana γ
- coefficiente caotico (raggio spettrale di R)
- rumore su $r(t)$
- il passo di integrazione

4.8 FeedbackNet

Questa classe racchiude il reservoir e rappresenta il modello complessivo risultante dall'apprendimento. Il costruttore di `FeedbackNet` accetta i seguenti parametri:

- il reservoir a cui è collegato
- i pesi del readout
- il livello di rumore sull'uscita.

Il metodo `simulate` può essere chiamato in due modi:

- `simulate(input)`: se la rete accetta input, si passa il vettore (colonna) degli input. In questo caso, la rete viene simulata per il numero di passi necessario, passando, ad ogni istante di tempo gli input presi dal vettore.

- `simulate(steps)`: se invece la rete è autonoma, il parametro atteso è uno scalare, che rappresenta il numero di passi da simulare

In entrambi i casi viene restituita l'uscita del neurone di lettura per ogni istante di simulazione.

4.9 Metodi di training

I metodi di training sono implementati anche essi come blocchi che sono collegati ad un reservoir. In particolare:

- il blocco che implementa il training tramite regressione ha costruttore `RegressionTraining(reservoir, beta)`, dove `beta` è il parametro di regolarizzazione.
- la regola EH è implementata in `EHRule(reservoir, eta0, T, alpha)`, in cui: `eta0` è il valore base del learning rate, `T` è il coefficiente di decadimento del learning rate e `alpha` è la posizione del polo nel filtro per $\bar{p}$ e $\bar{z}$. Il learning rate è costante quando `T` = -1.

Entrambe le classi forniscono il metodo `train(input, output)` (e, per reti autonome, `trainNoInput(output)`), in cui `input` e `output` sono due matrici che identificano la regola da imparare. I metodi ritornano due valori:

1. una `FeedbackNet` con il readout calcolato tramite il metodo di training,
2. l'output ottenuto durante la fase di training.

4.10 Lyapunov e P

Entrambi i blocchi sono funzionalmente identici. Entrambi questi metodi *non* modificano lo stato del reservoir, ma operano su di una copia.

Il metodo `compute(reservoir, force, K, N)` permette di calcolare tale quantità. Entrambi i parametri sono opzionali.

- Il primo parametro, `reservoir` indica di quale reservoir si debba calcolare la cifra di merito.
- Il parametro `force` serve per fornire alla rete il forcing.
- Il parametro `K` indica il numero di passi da ignorare per eliminare i transitori iniziali. Il valore di default per questo parametro è 1000.

- L'ultimo parametro, `N`, è invece il numero di passi da simulare per calcolare la quantità. Il valore di default è 2000.

4.11 Dettagli di basso livello

L'implementazione dei controllori è stata fatta in MATLAB, modellando la rete come descritto in precedenza. La scelta di MATLAB è stata effettuata perché è un linguaggio che permette lo sviluppo di un prototipo in modo piuttosto efficiente. Inoltre, siccome la maggior parte delle operazioni che si devono effettuare sono moltiplicazioni fra matrici e vettori, MATLAB risulta molto più espressivo rispetto ad altre scelte.

In aggiunta, la formulazione con i calcoli matriciali permette di utilizzare operazioni di tipo *BLAS* e *LAPACK*. Le operazioni BLAS (*Basic Linear Algebra Subprograms*) sono sottoprogrammi per effettuare le operazioni di algebra lineare più comuni, come addizioni e moltiplicazioni. Attualmente BLAS è uno standard de facto, e MATLAB utilizza una libreria che fornisce queste funzionalità ad un livello più alto. Anche *LAPACK* (ovvero *Linear Algebra Package*) è una libreria standard, che però implementa routine per operazioni di più alto livello, come funzioni per risolvere sistemi di equazioni lineari o calcolare la SVD. Anche LAPACK è incorporato in MATLAB. Anche queste operazioni di conseguenza sono molto efficienti, nonostante MATLAB non sia un linguaggio compilato. Un problema di performance che MATLAB può introdurre è legato alle chiamate a funzione, che possono essere lente rispetto ad altri linguaggi. Il reale impatto di questa caratteristica è discussione nella sezione 5.1.

Una breve nota riguarda il metodo di integrazione scelto: siccome nel reservoir computing si considerano reservoir molto grandi, utilizzare algoritmi di integrazione più complicati rispetto ad Eulero-avanti potrebbe risultare in un costo computazionale troppo grande. Ovviamente l'implementazione della libreria rende molto semplice modificare questo metodo, se fosse necessario.

4.11.1 FORMATO DEI DATI

Una scelta che si deve effettuare riguarda il formato dei dati. In particolare, si deve decidere se salvare i dati in formato riga o colonna. Per allineare la mia implementazione con le librerie di calcolo scientifico e *machine learning* già esistenti, l'input e l'output sono considerati come vettori colonna. Per estensione,

ovviamente, in sistemi con più di un input (o più di un output), l'ingresso (o, rispettivamente, l'uscita) è rappresentata come matrice in cui ogni colonna rappresenta un ingresso (o una uscita) e ogni riga un istante di tempo. Questa scelta è solo una questione di rappresentazione e non modifica alcuna considerazione tra quelle effettuate. Infatti, le equazioni studiate finora utilizzano vettori riga per rappresentare ingressi e uscite, quindi basta considerare la versione trasposta.

Questa scelta di rappresentazione, inoltre, produce anche alcuni (piccoli, ma misurabili) vantaggi in termini di tempo di simulazione. Infatti, come è già stato evidenziato, l'operazione più dispendiosa da effettuare nella simulazione del reservoir è il prodotto tra matrice sparsa e vettore $Rr(t)$. Per compiere questa operazione si deve moltiplicare ogni riga di R per $r(t)$ (si tratta di un vettore colonna). Siccome MATLAB salva in memoria le matrici nel formato *column-major*, ovvero le colonne vengono salvate in modo sequenziale, accedere alle righe è più lento che accedere alle colonne.

Invece, la versione trasposta, $r(t)^\mathsf{T} R^\mathsf{T}$, accede a $r(t)^\mathsf{T}$ (essendo un vettore, i suoi elementi sono comunque salvati sequenzialmente) e alle colonne di R^T, che sono salvate sequenzialmente, di conseguenza il tempo utilizzato per accedere alla memoria è più basso. Altri linguaggi di programmazione potrebbero usare la convenzione opposta (*row-major*), quindi è un aspetto che si dovrebbe considerare qualora si dovesse riscrivere la libreria per un altro ambiente.

4.11.2 FUNZIONAMENTO BATCH

Correntemente il software è stato pensato per funzionare principalmente in maniera offline, ovvero i passi necessari vengono effettuati in maniera sequenziale: prima si salvano i movimenti, poi si addestra la rete e in seguito si usa il risultato della simulazione della rete per pilotare il robot. La lettura dei movimenti e il training potrebbero essere fatti in parallelo: appena si ha un dato nuovo lo si fornisce alla rete per il training. Naturalmente lo stesso vale per il procedimento di simulazione rete e controllo del robot. La versione online non è particolarmente più complicata dal lato implementativo, anzi, attualmente la libreria potrebbe funzionare anche in maniera online. Questa strada non è stata seguita poiché non avrebbe prodotto alcun vantaggio ma avrebbe reso il codice più intricato e la simulazione sarebbe più lenta. Questa scelta rende inoltre possibile salvare i dati e quindi il procedimento diventa più riproducibile.

CAPITOLO 5

Risultati preliminari

In questa sezione verranno presentati alcuni risultati preliminari ottenuti confrontando i Central Pattern Generators e le Reti Neurali Caotiche. Questi risultati sono basati su problemi "sintetici", ovvero il cui obiettivo è approssimare una funzione non collegata al problema della locomozione.

Lo scopo principale di questa sezione è motivare in maniera empirica alcune osservazioni effettuate nelle parti precedenti e presentare dei primi risultati per identificare le caratteristiche che portano a preferire un modello oppure l'altro.

Per valutare la qualità della soluzione si è utilizzato l'errore quadratico medio normalizzato:

$$MSE_n = \frac{1}{\mathrm{Var}\left[f(t)\right]} \mathrm{E}\left[(z(t) - f(t))^2\right]$$

Inoltre, visto che si considera prevalentemente la generazione di segnali periodici si può fare una ipotesi "biologica", adottata anche in [14], e considerare l'errore solo sulla forma del segnale, ignorando piccole differenze di frequenza. Infatti, se un animale genera un pattern periodico (per la locomozione, per esempio), non è tanto importante che riproduca in maniera esatta le frequenze del segnale obiettivo, ma la forma delle oscillazioni. Le differenze di frequenza causano dei piccoli sfasamenti, che possono essere ignorati nella valutazione delle performance. Per farlo è sufficiente valutare l'errore medio su ogni periodo del segnale generato (nelle prove effettuate questa cifra di merito è indicata come MSE_{dc}). Ovviamente non è sempre possibile calcolare questo errore in maniera semplice, siccome si deve conoscere il periodo del segnale.

Per gli esperimenti riporto anche il tempo di esecuzione dell'algoritmo utilizzato. Per rendere più significative le misure, considero una media dei tempi di 5 simulazioni con gli stessi parametri. Ovviamente, essendo il tempo di testing

indipendente dal metodo di training utilizzato (naturalmente a parità di famiglia: CNN o CPG) non è riportato in tutti i casi.

Inoltre, nelle prove che coinvolgono le reti caotiche le cifre di merito sono state mediate su 5 simulazioni, per essere sicuri della riproducibilità (e che non si trattasse semplicemente di una scelta "fortunata" dei parametri casuali). Questo procedimento è ovviamente inutile nei CPG, poiché essi sono completamente deterministici.

5.1 Misure

Come anticipato, le scelte di implementazione effettuate permettono l'utilizzo di libreria di algebra lineare, che sono molto efficienti. Nonostante questo, l'esecuzione in MATLAB fa sprecare alcuni secondi per il cosiddetto *overhead di chiamata a funzione*. Infatti, in tutti i linguaggi di programmazione, per chiamare una funzione, si incorre in un costo aggiuntivo, semplicemente per passare gli argomenti. In MATLAB questo costo è superiore rispetto ai linguaggi compilati, perché si effettuano vari controlli (come ad esempio impostare le variabili `nargin` e `nargout`, che indicano, rispettivamente, il numero di variabili in ingresso e uscita). Fortunatamente, questo overhead è costante rispetto alle caratteristiche della rete, ma dipende solo dal numero di chiamate a funzione, quindi aumenta coll'aumentare del tempo di simulazione.

5.1.1 CENTRAL PATTERN GENERATOR

La complessità dei CPG è lineare rispetto al numero di oscillatori presenti nel generatore di pattern. In questa sezione, lo scopo è controllare se la scelta di questo valore modifichi le performance in maniera significativa.

Per farlo si genera un CPG variando il numero di oscillatori, la si addestra e si misura il tempo necessario per la simulazione. Per le misure, riportate in tabella 5.1, considero lo stesso problema dettagliato nella sezione 5.2 (anche se il tipo di problema non è particolarmente importante quando si misurano le performance della simulazione). I tempi sono ricavati simulando il CPG per 10 secondi e mediati su 10 prove.

Come si vede dai risultati, il tempo necessario per simulare un CPG è molto limitato e il numero di oscillatori può essere scelto più grande del necessario senza avere grosse ripercussioni sul tempo d'esecuzione.

numero oscillatori	4	5	10	20	50
tempo (s)	1.126	1.156	1.226	1.351	1.613

Tabella 5.1: Media dei tempi di esecuzione per simulare un CPG con numero di oscillatori variabile.

5.1.2 RETI CAOTICHE

Come già analizzato nella sezione 4.3.1, ci sono essenzialmente 3 parametri che governano le performance di una rete caotica: la dimensione del reservoir, la sparsità e ovviamente il tempo di simulazione.

Lo scopo di questa sezione è verificare come la scelta di dimensione e sparsità influenzi il tempo di esecuzione. Per farlo si genera una rete con i parametri sotto esame, la si addestra e si simula per 10 volte, misurando il tempo impiegato. Nella tabella tabella seguente riporto, per ogni configurazione provata, la media dei tempi d'esecuzione. Tutti i risultati sono ottenuti simulando la rete per 10 secondi, riportati in tabella 5.2.

sparsità (%)	**tempo (secondi)**		
	500 neuroni	1000 neuroni	1200 neuroni
0.01	2.063	2.703	2.962
0.05	2.250	3.344	3.971
0.1	2.461	4.708	6.034
0.2	2.739	8.100	11.648
0.5	3.669	15.459	19.629
1	6.074	21.669	26.592

Tabella 5.2: Media dei tempi di esecuzione per la simulazione di una CNN allenata, al variare di sparsità e dimensione.

Per esempio, considerando i risultati del reservoir di 500 neuroni con sparsità variabile si può notare che c'è una costante additiva pari a circa 2 che è imputabile all'overhead causato dalle chiamate a funzioni in MATLAB. Sottraendo questa costante si vede chiaramente che il tempo di esecuzione varia in modo lineare all'aumentare della sparsità.

5.2 Generazione autonoma di pattern

I neuroni biologici producono vari tipi di attività ritmiche per diversi scopi (come attivazione di muscoli, movimento o respirazione). Il primo task che considero, ispirato a questo fatto, riguarda la generazione autonoma di una funzione sinusoidale, senza alcun input.

$$f(t) = \sin 2\pi t - 0.7 \sin 4\pi t + 0.2 \sin 6\pi t + 0.85 \cos\left(8\pi t + \frac{\pi}{3}\right)$$

Questo problema è simile, ma molto più semplice, del problema noto in letteratura col nome di *multiple superimposed oscillators*, MSO. Questo problema consiste nell'approssimare un segnale che è combinazione lineare di sinusoidi le cui frequenze di oscillazione non sono multiple fra di loro.

Il problema MSO *non* può essere risolto tramite il reservoir computing illustrato qui. Esistono alcune estensioni, che solitamente prevedono di complicare il readout, che permettono di ottenere buoni risultati anche con tempi di training relativamente brevi. Però, visto che i segnali oggetto di questo problema hanno periodi molto lunghi, non è particolarmente interessante ai nostri scopi.

5.2.1 CON CENTRAL PATTERN GENERATOR

Nei Central Pattern Generator, i parametri scelti sono i seguenti: $N = 4, \gamma = 8, \mu = 1, \tau = 0.5, \delta = 1$ ms.

Training di PCPG

Oltre ai parametri comuni, per i PCPG si sono usati i seguenti: $\eta = 0.5, \epsilon = 0.9$, tempo di training = 700, $x(0) = \mathbf{1}, y(0) = \mathbf{0}, \alpha(0) = \mathbf{0}, \phi = \mathbf{0}, \omega = [6\ 12\ 18\ 24]$.

Come si vede dalla figura 5.1, il training tramite questo metodo trova un CPG con errori molto piccoli; si ottiene infatti $MSE_n = 0.0028$.

In questo caso, il training impiega 95.17 secondi, mentre il testing 3.41. Si nota anche che il training viene completato in 700 secondi simulati con questi valori di partenza per ω, se infatti i valori iniziali di ω fossero diversi, il tempo di convergenza sarebbe molto più elevato o addirittura non convergerebbe al segnale corretto. Per esempio, con ω uniformemente distribuito tra 5 e 20 si vede che due oscillatori convergono allo stesso valore di frequenza; si può risolvere questo pro-

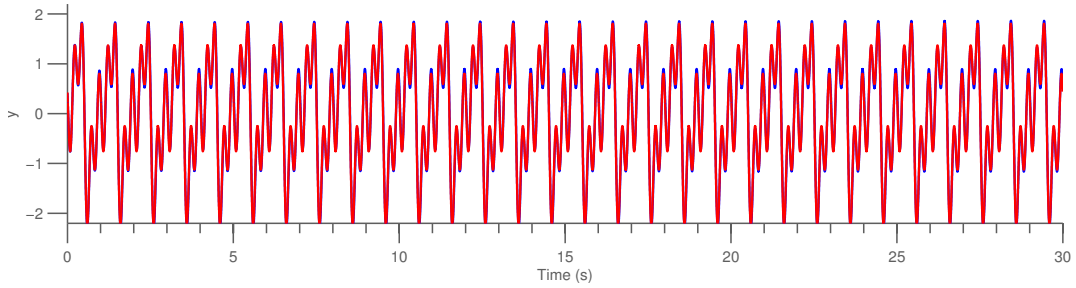


Figura 5.1: Risultato del training di PCPG. Si vedono i primi 30 secondi di test, il segnale desiderato è rappresentato in rosso, mentre l'uscita del CPG è in blu.

blema aggiungendo un ulteriore oscillatore, anche mantenendo lo stesso tempo di training.

Training tramite Fourier-PCPG

Per eliminare l'impatto della scelta dei valori iniziali, si può usare questo metodo di training. Infatti esso calcola i valori corretti di ω a partire anche semplicemente da un periodo della funzione da approssimare. Naturalmente, per ridurre l'effetto del rumore, è buona pratica considerare un numero di periodi piuttosto grande.

In questo esperimento si considerano 30 secondi di dati per il training, e la rete viene simulata per il testing per 50 secondi. Si ottiene, anche in questo caso, un errore molto basso: $MSE_n = 0.0064$.

Per questo algoritmo la fase di training è virtualmente istantanea (impiega infatti 0.181 secondi).

5.2.2 CON RETI NEURALI CAOTICHE

Usando neuroni lineari questo task è relativamente semplice: come visto nella sezione 3.8.1, questo tipo di neuroni implementano una potenza di matrice, che può formare componenti sinusoidali solo se la matrice ha degli autovalori complessi (di conseguenza bastano $2k$ neuroni per approssimare una funzione composta da k armoniche). Ma come già dimostrato, in un sistema con feedback il metodo di learning ha completo controllo sulla matrice delle connessioni interne W^{eff}. Da queste due osservazioni segue che è sempre possibile effettuare il training di un sistema formato da neuroni lineari per generare somme di sinusoidi

con errori di training molto bassi (vicini allo zero macchina). Durante la fase di testing, però il sistema si dimostra quasi sempre instabile oppure che tende a 0. I neuroni lineari, quindi, sono relativamente interessanti dal punto di vista teorico, ma poco pratici. Di contro, i sistemi nonlineari sono caratterizzati da aree di stabilità molto più grandi, quindi anche in fase di testing tendono a mantenere l'andamento voluto.

Per tutti i test di questo tipo che seguono è stata usata la configurazione seguente: rete senza input, con $\delta = 1$ ms, $N = 1000, p = 0.1, \lambda = 1.7, \tau = 40$ ms. Il reservoir è formato da neuroni di tipo Leaky Integrator (modello 3.1).

Training con regola EH

I parametri relativi a questo metodo di training sono i seguenti: tempo di training = 500 secondi, tempo di testing = 50, T = 20, $\alpha = 0.75$. Dalla figura si nota che il training ha avuto successo, e il segnale è riprodotto in maniera corretta, tranne per la piccola deriva temporale descritta sopra (comunque piccola, come si vede in figura 5.3). Si ottiene infatti un $MSE_n = 0.63$; se invece si usa l'errore corretto rispetto a questa deviazione si ottiene un $MSE_{dc} = 0.0024$. Nella figura 5.2 si può vedere un dettaglio dei primi 3 periodi del testing.

In questo caso il rumore sull'uscita è stato eliminato dopo il training; anche mantenendo il rumore sul readout, comunque, si ottengono risultati simili. Nelle prove effettuate con questa configurazione si è misurato un $MSE_n = 0.94, MSE_{dc} = 0.0314$.

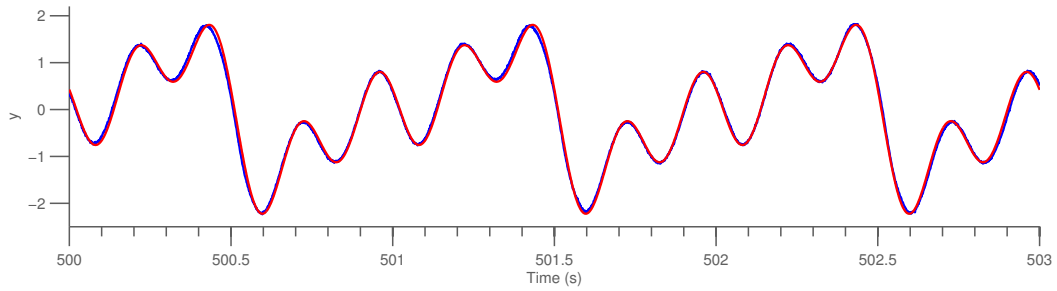


Figura 5.2: Training con regola EH. Primi 3 periodi del segnale imparato. In blu è rappresentata l'uscita della rete, mentre in rosso il segnale desiderato.

Per questo task, il training ha impiegato 239 secondi, mentre il testing è stato completato in 21.7.

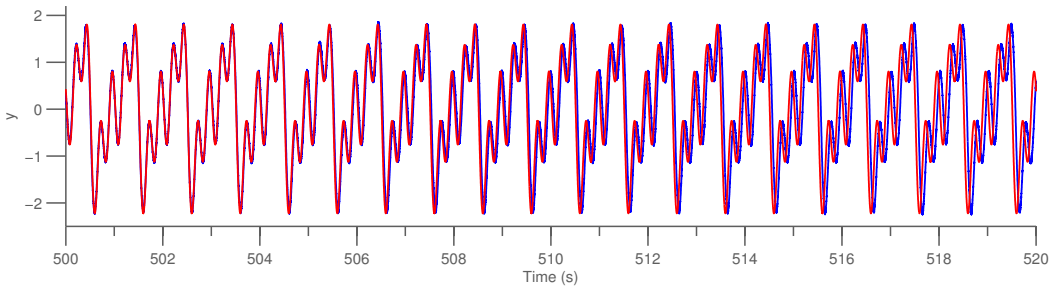


Figura 5.3: Training con regola EH. Primi 20 secondi del testing. A partire dai 10 secondi si nota una discrepanza dovuta alle frequenze non imparate in maniera precisa.

Training con regressione

I parametri relativi a questo metodo di training sono i seguenti: tempo di training = 50 secondi, tempo di testing = 50 secondi. Come si vede dalla figura, il training effettuato tramite regressione riesce a produrre errori molto bassi, infatti $MSE_n = 0.0098$. Il piccolo errore presente è causato principalmente dal rumore sulle uscite dei neuroni del reservoir. Per questa simulazione, il training ha impiegato 27.5 secondi, mentre il testing termina in 16.2 secondi.

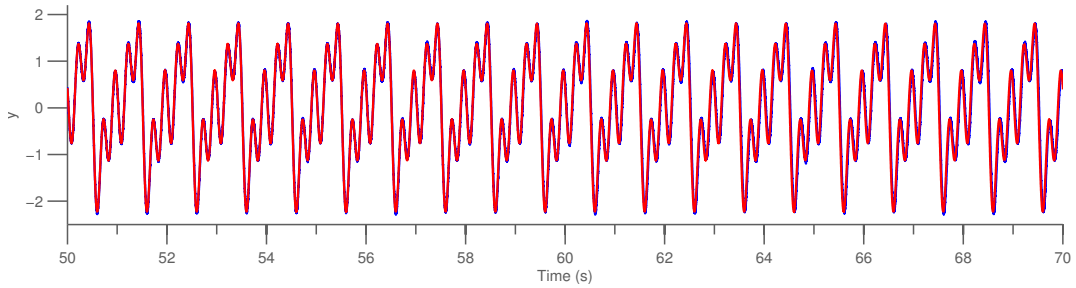


Figura 5.4: Training con regressione. Primi 20 secondi del testing. Si nota solo qualche minore errore (in blu) rispetto al segnale obiettivo (in rosso)

Il training tramite FORCE learning è stato effettuato con gli stessi parametri. Ovviamente, anche in questo caso l'errore è molto basso ($MSE_n = 0.006$) e i tempi sono: 260.3 secondi per il training e 17.2 per il testing. Come anticipato, nonostante l'algoritmo per fare il training tramite FORCE learning abbia la stessa complessità computazionale dell'addestramento tramite regressione, il tempo di esecuzione è molto diverso. Infatti il FORCE learning coinvolge, in ogni passo di training, una matrice di dimensione pari a N (la dimensione del reservoir), quindi

gli accessi a memoria occupano una parte considerevole del tempo di esecuzione.

Utilizzando la variante Leaky Integrator

In questa prova si è anche utilizzata la variante del leaky integrator presentata nella sezione 3.8.2 per verificare che il metodo di training con la regola EH funzionasse anche con questo neurone. I parametri sono i seguenti: $\delta = 1$ ms, $N = 1000, \alpha = 1, \rho(R) = 1.4, \gamma = 20$ ms.

Si effettua il training tramite regola EH in 240 secondi e il testing in 24, ottenendo un $MSE_n = 0.65$.

Come si evince sia dai risultati sperimentali riportati qui sia dall'analisi teorica riportata in sezione 3.8.2 i risultati ottenuti con questo modello o col modello 3.1 (riportati sopra) sono molto simili e i tempi d'esecuzione di entrambi i modelli sono paragonabili. Viste queste considerazioni, nel seguito non utilizzerò questo modello.

CAPITOLO 6

Risultati su problemi di locomozione

In questa sezione continuo gli esperimenti iniziati in sezione 5 ed espando al fine di generare le traiettorie per controllare i giunti dei robot presi in considerazione. Inizio quindi la sezione illustrando il procedimento seguito per raccogliere i dati e l'ambiente in cui verranno provati i robot. Dopo questa introduzione inizio gli esperimenti effettivi.

6.1 Introduzione al problema

Si vuole controllare la camminata di 3 robot: due bipedi e un quadrupede. Per controllare la validità dei risultati si lavora inizialmente in simulazione e solo dopo i risultati saranno provati su di un robot fisico.

6.1.1 SCELTA DEL SIMULATORE

La simulazione è effettuata tramite il software V-REP. Questo software è stato scelto rispetto ad altri prodotti (Gazebo o Webots) per vari motivi: si tratta di un simulatore gratuito (nella versione *EDUCATIONAL*, comunque completa), che supporta facilmente tutti i sistemi operativi più comuni, è possibile l'interfaccia con molti linguaggi di programmazione (ad esempio: Python, Java, MATLAB, Lua, ...) e ha diversi modelli robotici pronti all'uso. Supporta inoltre il paradigma client-server, che rende possibile una eventuale separazione del simulatore dal training effettivo.

V-REP, come la maggior parte dei simulatori robotici, non implementa un unico metodo di locomozione, ma permette di implementare separatamente un con-

trollore per il proprio robot, in modo che si possa eseguire l'algoritmo preferito. In questo lavoro è stata utilizzata la versione già implementata nel modello del robot in V-REP. Per esempio, NAO utilizza traiettorie calcolate tramite ZMP, mentre la camminata di Asti è dinamica[1], calcolata risolvendo la cinematica inversa.

6.1.2 ROBOT CONSIDERATI

In questo lavoro vengono considerati due bipedi umanoidi: Asti e NAO. In particolare, Asti è un robot esistente solo in simulazione, disegnato da Lyall Randell. Esso è basato su Asimo, il robot sviluppato da Honda, che cammina tramite ZMP. Asti, come la maggior parte dei robot bipedi esistenti, possiede 6 gradi di libertà per gamba, 3 per braccio e 2 sul collo, totalizzando quindi 20 gradi di libertà. In questo problema ci concentreremo solo sulle gambe, controllando quindi 12 gradi di libertà.

Inoltre, in Asti (unico fra quelli considerati) è possibile modificare la lunghezza del passo e la velocità. Entrambi sono modificabili tramite degli *sliders*, e possono assumere valori da 0 a 1000, ma valori troppo grandi tendono a rendere il robot instabile. Questa caratteristica verrà sfruttata per modulare le traiettorie del robot. Correntemente Asti non ha alcun sensore che si possa integrare, pertanto gli unici valori utilizzabili come ingressi esogeni sono questi sliders.

Invece NAO è un piccolo robot (è alto solo 58 cm) sviluppato da Aldebaran. Attualmente è molto usato sia nell'ambito della ricerca sia in quello educativo, vista la sua facilità di programmazione e basso costo. Ha 21 gradi di libertà, di cui 6 in ciascuna gamba. Questo robot è stato scelto perché l'AIRLab ne possiede uno, quindi sarà possibile provare i risultati della simulazione su di un robot fisico.

Il quadrupede considerato è Robbie, un robot disegnato anch'esso da Lyall Randell, che ha la forma di un gatto; questo robot è ispirato da AIBO, il cane robot di Sony. Esso ha 3 gradi di libertà per zampa (2 nell'anca e 1 nel ginocchio), quindi 12 in totale. Ha anche un grado di libertà nella coda e 2 nel collo, ma anche in questo caso questi ultimi giunti non sono interessanti rispetto ai nostri scopi.

6.2 Procedimento

Per verificare se le CNN possano essere dei controllori per il movimento di robot con zampe, il procedimento seguito, illustrato ad un livello piuttosto alto, è

[1] In una camminata dinamica il centro di massa esce dall'area di supporto

il seguente:

1. Registrare i movimenti delle zampe del robot,
2. Imparare le traiettorie tramite CNN o CPG,
3. Utilizzare la rete neurale per generare nuove traiettorie,
4. per verificare la qualità dell'addestramento, si guida il robot con questi dati, controllando che esso sia stabile e cammini.

Una delle prime considerazioni da effettuare quando si progetta un controllore per robot consiste nello scegliere se effettuare controllo in posizione o controllo in forza. Nel controllo in posizione l'output del controllore è la posizione desiderata dei giunti, mentre nel controllo in forza il controllore produce una forza da applicare al motore. La seconda soluzione è più diretta, ma necessita di una conoscenza abbastanza approfondita del robot a cui il controllore è applicato, mentre la prima necessita di un ulteriore livello di computazione per arrivare al valore da fornire al motore.

In questo caso, siccome si utilizza un simulatore, la differenza tra i due approcci è minima. Infatti V-REP permette di ottenere e controllare entrambi i valori senza alcuna differenza. La versione fisica di NAO, invece, è facilmente controllabile solo in posizione. Si è scelto il controllo in posizione principalmente per questo motivo, ma anche perché l'uscita di un controllore di questo tipo è la traiettoria del giunto, quindi più facilmente interpretabile rispetto ai valori calcolati da un controllore in forza.

Per svolgere i passi di comunicazione con il simulatore sono state definite delle funzioni MATLAB che semplificano la comunicazione con il server V-REP: sono spiegate nell'appendice A.

6.3 Traiettorie di Asti

In questo primo esperimento ci si occupa della versione multidimensionale del problema nella sezione precedente. Più in dettaglio, si vuole generare i segnali da fornire ai giunti di Asti per controllare la sua locomozione. Come già visto, Asti ha 12 gradi di libertà nelle gambe quindi lo scopo è generare le 12 uscite necessarie per pilotare gli arti inferiori.

In seguito l'output di ciascuno di questi metodi è stato fornito ai giunti del robot. In tutti i casi Asti ha camminato senza grossi problemi di stabilità: si vede un esempio del passo in figura 6.1.

6.3.1 CON CPG

In questo caso si replica lo schema di connessione illustrato nella sezione 2.6. I parametri scelti sono i seguenti: $N = 5, \gamma = 8, \mu = 1, \tau = 0.5$. Il passo di integrazione $\delta = 1$ ms.

PCPG

Si addestra la rete, configurata con i parametri $\epsilon = 0.9, \eta = 0.5$. Come valori iniziali per l'integrazione: di $x(0) = [1, 1, 1, 1, 1], y(0) = [0, 0, 0, 0, 0], \alpha(0) = [0, 0, 0, 0, 0], \phi = [0, 0, 0, 0, 0]$ e $\omega = [4, 8, 12, 16, 20]$. Il training dura 500 secondi.

Alla fine dell'addestramento si vede che l'uscita è riprodotta in maniera corretta, e si ottengono gli MSE_n che sono stati riportati in tabella 6.1.

Come si vede, i risultati sono molto precisi, ma il training è piuttosto lento, impiega infatti 869.52 secondi, (i 30 secondi di testing vengono simulati in 51.1 secondi). I tempi d'esecuzione sono così alti per due motivi: durante il training si utilizzano molti dati e si devono simulare molti oscillatori (5 per i 12 gradi di libertà).

FourierCPG

Anche in questo caso l'uscita è riprodotta in maniera molto precisa; i risultati in termini di MSE_n sono riportati nella tabella 6.1. Come nel problema riportato nella sezione precedente, il training è molto veloce (0.35 secondi) tanto da risultare praticamente trascurabile.

6.3.2 CON RESERVOIR COMPUTING

Per questo problema i parametri scelti sono: rete senza input, con $\delta = 1$ ms, $N = 1000, p = 0.1, \lambda = 1.9, \tau = 100$ ms. Naturalmente, siccome si devono generare 12 segnali, la rete avrà 12 uscite.

L'aumento del parametro λ è necessario solo per il training tramite regola EH, infatti tenere $\lambda = 1.5$ produce buoni risultati sia per il FORCE learning sia per la regressione, ma la regola EH genera una rete che tende ad annullarsi. Questo è abbastanza in linea sia con l'interpretazione di λ come via "economica" per aumentare il potere della rete, sia col fatto che il training tramite regressione ha il completo controllo sulla matrice delle connessioni, quindi riesce a modificare il

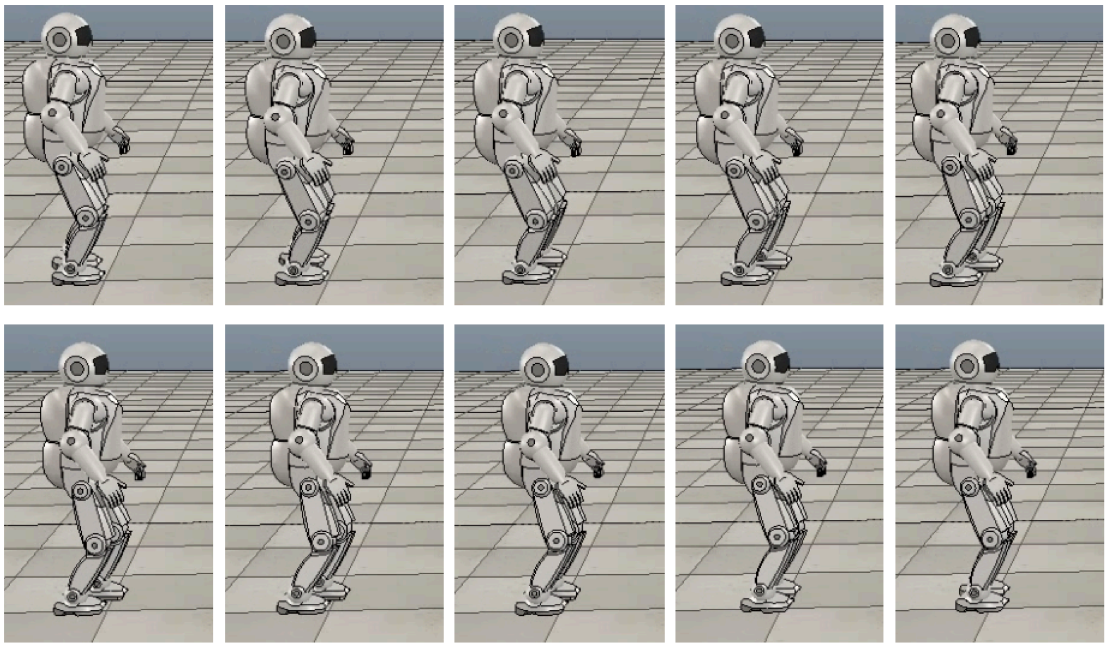


Figura 6.1: Simulazione di Asti mentre cammina, pilotato da CNN allenata tramite regola EH. Tra una fotografia e la successiva sono passati 200 ms.

valore del raggio spettrale di W^{eff} in modo tale che l'errore sia minimo.

Regola EH

I parametri aggiuntivi sono: tempo di training = 500 secondi, $\eta_0 = 5 \cdot 10^{-4}$, T = 20, $\alpha = 0.75$. Il tempo di testing è di 30 secondi. Gli errori sui singoli gradi di libertà sono riportati in tabella 6.1. Anche in questo caso, la maggior parte dell'errore è dovuto alla deriva temporale illustrata in precedenza. Considerando la cifra di merito corretta rispetto a questa deviazione si ottiene un MSE_{dc} nell'ordine di 10^{-3} per ogni giunto.

Il training impiega 480.32 secondi, mentre il testing viene completato in 16.12 secondi.

Regressione

Per il training con la regressione sono stati usati i seguenti parametri: tempo di training = 60 secondi, tempo di testing = 30 secondi, $\beta = 0.5$. Il training ha impiegato 35.3 secondi, il testing 15.21 secondi. Per il FORCE learning (con $\beta = 0.00001$), il training è stato completato in 326.1 secondi.

giunto	MSE_n				
	PCPG	FourierPCPG	Regola EH	Regressione	FORCE
1	0.1	0.149	0.889	0.248	0.116
2	0.191	0.091	0.749	0.173	0.081
3	0.133	0.273	1.068	0.340	0.149
4	0.491	0.937	1.572	1.042	0.602
5	0.287	0.114	0.751	0.173	0.081
6	0.148	0.228	1.116	0.398	0.169
7	0.098	0.153	0.849	0.277	0.135
8	0.277	0.087	0.760	0.171	0.083
9	0.131	0.256	1.045	0.368	0.184
10	0.478	0.881	1.521	1.063	0.687
11	0.286	0.114	0.758	0.171	0.084
12	0.152	0.225	1.083	0.423	0.209

Tabella 6.1: MSE_n calcolato su reti allenate con i metodi presentati, per ogni grado di libertà del robot Asti.

Anche in questo caso, il training con questo metodo produce ottimi risultati, come si vede in tabella 6.1.

6.4 Traiettorie di NAO

Questa sezione si occupa dello stesso problema della sezione precedente, ma riferito al robot NAO. Questo robot è stato scelto poiché, come già anticipato, esso esiste realmente e l'AIRLab lo possiede. Visto che il problema è già stato spiegato in dettaglio, qui riporto solo i risultati principali.

NAO ha un comportamento particolare nella camminata che adotta solitamente: immediatamente prima di avviarsi si flette leggermente sulle ginocchia, abbassando il centro di massa. Questa particolarità non può essere riprodotta in nessun modo con i metodi considerati. Si verifica però in simulazione che questo movimento è in realtà utile, visto che senza di esso il robot, nelle fasi iniziali, tende ad essere leggermente instabile e può cadere. Successivamente il segnale è riprodotto correttamente e anche in simulazione non si notano grossi problemi.

In seguito i dati sono stati letti anche da un robot fisico, e sono stati poi usati per addestrare una CNN al fine di riprodurre la camminata. Dopodiché i dati generati dalla rete sono stati forniti in ingresso al robot, che, camminando, si è dimostrato stabile. Si verifica però che la camminata non è la stessa che si vede quando il robot cammina controllato da ZMP. La motivazione di questa differenza risiede probabilmente in imprecisioni nel procedimento di lettura dati; infatti, anche fornendo al robot i dati letti in precedenza si ottiene comunque una camminata diversa da quella osservata in fase di lettura. Questa seconda camminata è però uguale a quella generata dalla rete. Un esempio di NAO mentre cammina (in simulazione) si può trovare in figura 6.2.

6.4.1 CON CPG

Per allenare i Central Pattern Generator a produrre l'uscita desiderata, è stato considerato solo l'algoritmo di Fourier-CPG perché non è stato possibile collezionare abbastanza dati per utilizzare i PCPG.

Anche in questo caso si usa lo schema di connessione in sezione 2.6. I parametri scelti sono i seguenti: $N = 10, \gamma = 8, \mu = 1, \tau = 0.2$. Il passo di integrazione $\delta = 1$ ms. Si usano 30 secondi di training (Fourier-CPG impiega 0.124 secondi). La rete viene poi eseguita per 10 secondi, che vengono simulati in 16.841 secondi.

Le traiettorie vengono poi fornite al robot simulato. Il segnale fornito alle ginocchia non è stato imparato molto bene perché i dati originali erano molto rumorosi quindi la camminata del robot non risulta ottima: rispetto a quella attesa si nota che non solleva quanto necessario i piedi. Il robot si mantiene comunque stabile.

6.4.2 CON CNN

Come per i CPG, anche per addestrare la rete caotica è stato utilizzato solo l'algoritmo di FORCE learning (il numero di dati è troppo limitato per poter utilizzare la regola EH). La rete considerata utilizza i seguenti parametri: $N = 1000, p = 0.1, \lambda = 1.5, \tau = 50$ ms. Si utilizzano, come in tutti gli altri casi, neuroni di tipo Leaky Integrator.

Il training è simulato in 170 secondi. La rete è stata poi simulata per 10 secondi per ottenere il segnale da fornire al robot, impiegando 5.23 secondi. Le traiettorie generate per la gamba sinistra sono riportate in figura 6.3 (ad esclusione del grado

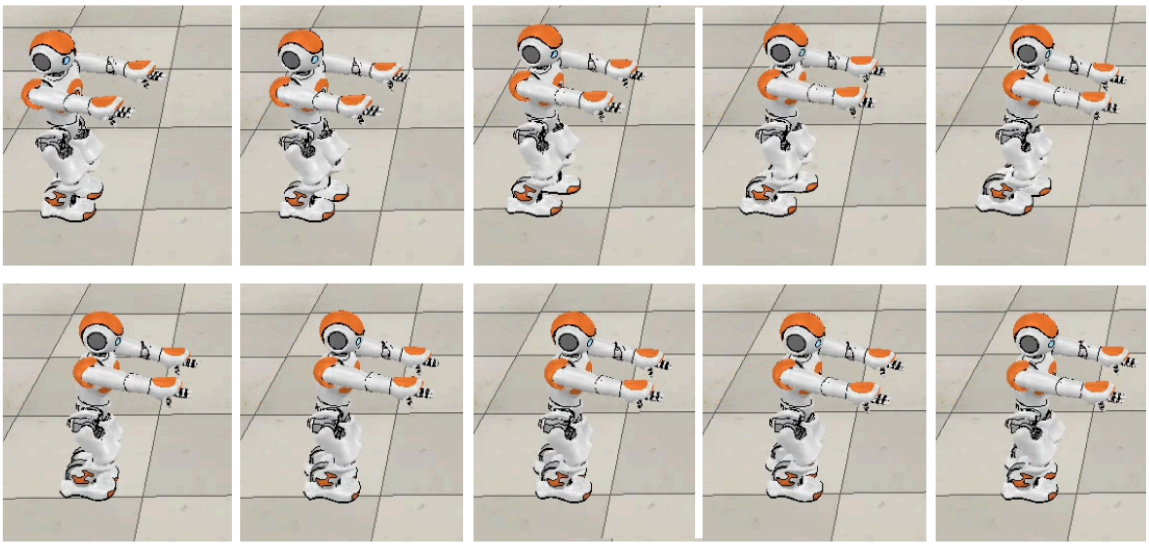


Figura 6.2: NAO simulato mentre cammina, controllato da CNN allenata tramite FORCE learning. Tra una fotografia e la successiva sono passati 100 ms.

di libertà nell'anca relativo al movimento di rotazione, che ha un valore costante pari a 0). Nella figura si vede che, nonostante la grossa quantità di rumore, che modifica in modo considerevole le traiettorie, la rete riesce comunque ad estrarre un movimento coerente, seppur non perfettamente periodico.

Successivamente i dati sono stati forniti sia al robot virtuale sia a quello fisico. In entrambi i casi è stato possibile ottenere una camminata stabile e il comportamento della simulazione e del robot reale è risultato identico. Si verifica che il passo generato non è uguale a quello osservato durante la fase di lettura dati (come già descritto in precedenza).

6.5 Robot quadrupede

Per verificare l'adattabilità dei metodi ad altre tipologie di robot, è stato anche effettuato il controllo di un robot quadrupede. In V-REP è presente un modello di quadrupede robot, Robbie, che, come ho già anticipato, ha 12 gradi di libertà.

La camminata che è implementata in V-REP per questo robot è simile a quella dei gatti reali, che muove una zampa alla volta, partendo dalla anteriore sinistra, poi posteriore destra, anteriore destra e per ultima posteriore sinistra (solitamente questo tipo di camminata è chiamata *crawl* o *creep*). Questo movimento è staticamente stabile. Nonostante il robot abbia 3 gradi di libertà per gamba, in questo pas-

so ne sfrutta solo 2, il primo nell'anca (che genera il movimento avanti/indietro) e quello nel ginocchio.

6.5.1 CON CPG

Per riuscire ad imparare il segnale tramite i CPG si è dovuto realizzare una versione sintetica delle traiettorie in modo tale che fossero meno rumorose. Utilizzando la versione letta dal simulatore, infatti, nessuno dei due metodi riesce a riprodurre in maniera soddisfacente il segnale. Il segnale utilizzato è stato calcolato prendendo pochi periodi del segnale letto da V-REP e ripetendoli fino ad arrivare alla durata desiderata.

La configurazione adottata è la seguente: $N = 12, \gamma = 8, \mu = 1, \tau = 0.5, \delta = 1$ ms. Il numero di oscillatori per ogni CPG è $M = 6$. Anche se utilizzo 12 CPG, solo 8 assumeranno un uscita periodica, in quanto gli altri 4 corrispondono al secondo grado di libertà nell'anca, che, in questa camminata, rimane fermo.

Con PCPG

Per fare il training si utilizzano 400 secondi di dati. Il testing dura 30 secondi. La fase di training è simulata in secondi, mentre la validazione impiega 36.12 secondi. Il valore iniziale per le frequenze angolari degli oscillatori è uniformemente distribuito tra 6 e 36. Anche in questo caso $\epsilon = 0.5, \eta = 0.9$.

In questo caso il sistema, dopo il training, rimane sul segnale obiettivo per pochi periodi, prima di arrivare su di un altro attrattore. L'errore è quindi molto alto, e il robot risulta instabile.

Con Fourier CPG

In questo caso per il training vengono utilizzati 30 secondi di dati, e il sistema viene testato per altri 30 secondi. Il tempo necessario per il training è praticamente trascurabile, impiega infatti 0.058 secondi, e il testing invece viene effettuato in 36.2 secondi. Tramite questo algoritmo il sistema riproduce senza grossi problemi il pattern voluto: il robot quadrupede riproduce la camminata in modo stabile.

6.5.2 CON RETI NEURALI CAOTICHE

Come nei casi precedenti, la rete in considerazione ha 1000 neuroni di tipo Leaky Integrator, connessi con sparsità $p = 0.1$. Il fattore di caos $\lambda = 1.7, \tau = 100$

ms. Lo step di integrazione è $\delta = 1$ ms.

Con regola EH

La configurazione utilizzata per l'algoritmo di training è la stessa usata anche nei casi precedenti: $\eta_0 = 0.0005, T = 20, \alpha = 0.75$. Utilizzo 400 secondi di dati per il training, e 50 per la validazione. Il training è completato in 271.2 secondi, mentre il testing impiega 24.23 secondi.

Anche in questo caso, come per il robot bipede, la regola EH impara il segnale bene, ma con qualche problema causato dalle derive temporali del segnale. Questo suggerisce che il problema è una conseguenza del metodo, e non della tipologia dei segnali (che in questo problema sono significativamente più regolari rispetto ai segnali necessari per il bipede). Comunque in simulazione si nota, anche per questo robot, che l'errore non causa problemi alla camminata.

Con regressione

In questo caso si utilizzano 50 secondi di dati per il training e 50 per il testing. L'addestramento tramite regressione impiega 30.1 secondi, e il testing è completato in 24.12. Tramite il FORCE learning, il training impiega 279.43 secondi.

Utilizzando la regressione, il sistema in fase di testing si dimostra instabile qualunque sia il fattore di regolarizzazione scelto. Più in dettaglio, il sistema riesce a riprodurre il segnale per alcuni istanti di tempo, ma in seguito diverge su di un altro comportamento (vedi figura 6.4 per un esempio). Questo è probabilmente dovuto ad un rumore abbastanza importante sul segnale da imparare. Invece il FORCE learning (con fattore di regolarizzazione $\beta = 0.0001$) riesce a riprodurre in maniera molto buona il segnale.

6.6 Modulazione della traiettoria

Questo problema consiste nel modulare le traiettorie dei giunti per realizzare un controllore più interessante del semplice replicatore presentato finora. L'aspetto più semplice della modulazione può includere il riprodurre le traiettorie imparate con velocità o ampiezze diverse. Versioni più complesse possono essere la risposta a delle forze esterne, o a degli stimoli, in modo da modificare l'andatura: per esempio, quando si passa da un terreno piano ad uno in salita il passo di deve modificare di conseguenza. Naturalmente certe rappresentazioni sono più adatte

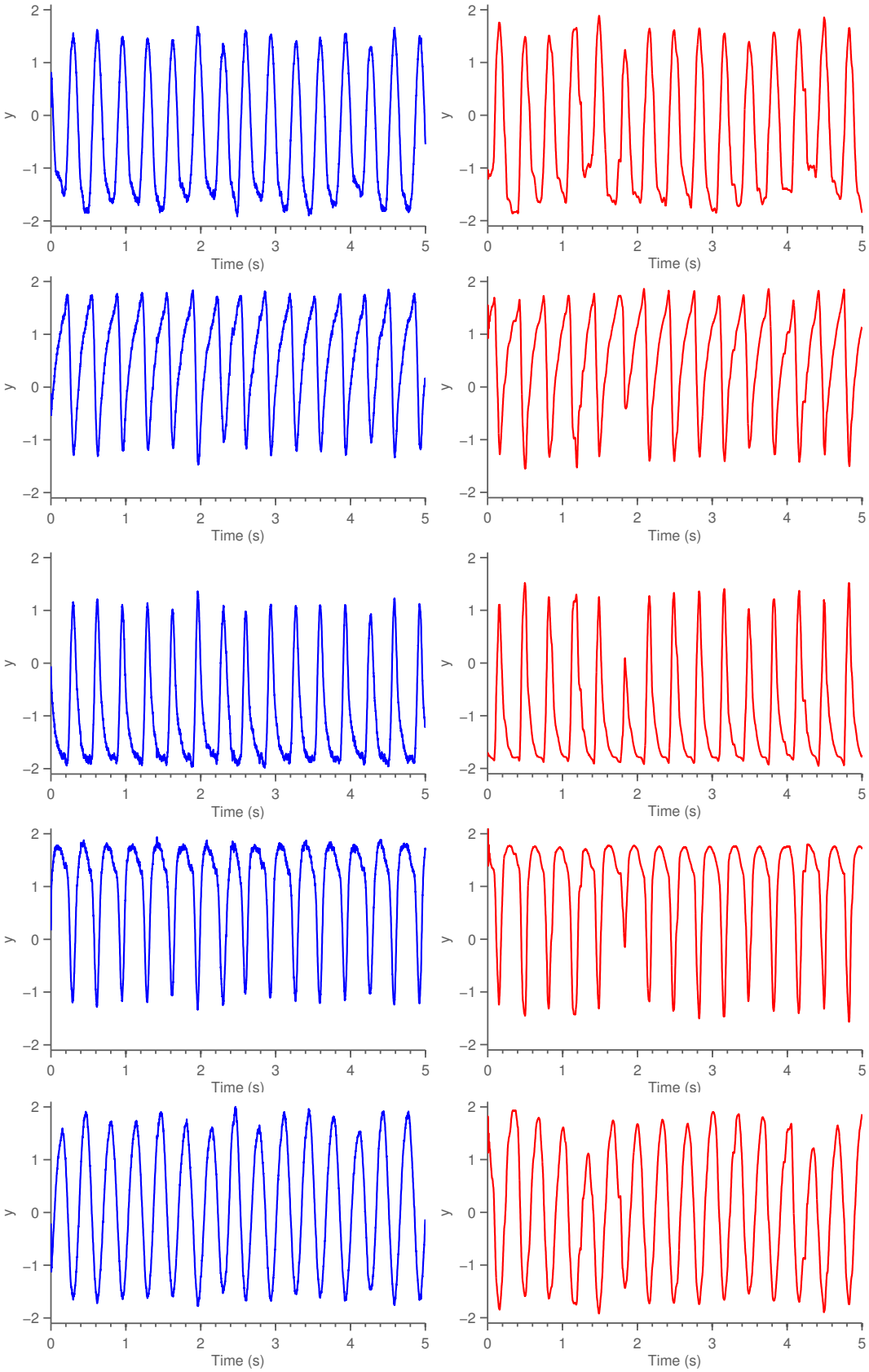


Figura 6.3: FORCE learning delle traiettorie di NAO ottenute dal robot reale. I segnali sono molto rumorosi, ma la rete riesce comunque ad approssimare l'uscita desiderata. A sinistra, in blu, il segnale generato dalla CNN, mentre in rosso è rappresentato il segnale letto dal robot.

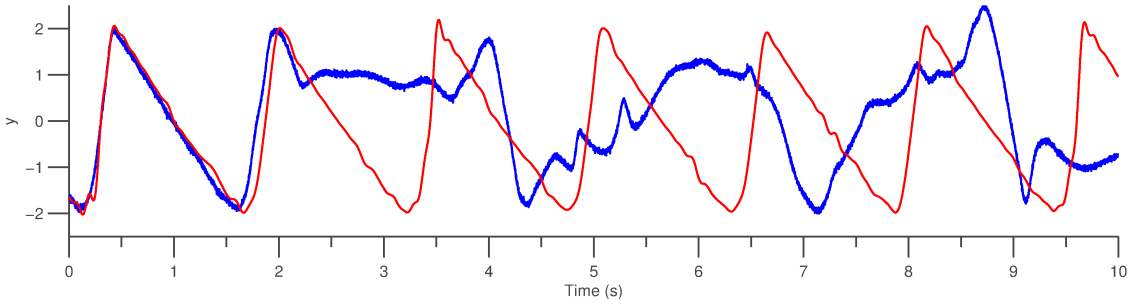


Figura 6.4: Primi 10 secondi di testing di una rete per la locomozione di Robbie allenata tramite regressione. È rappresentato solo il primo grado di libertà, ma gli altri hanno comportamenti del tutto analoghi.

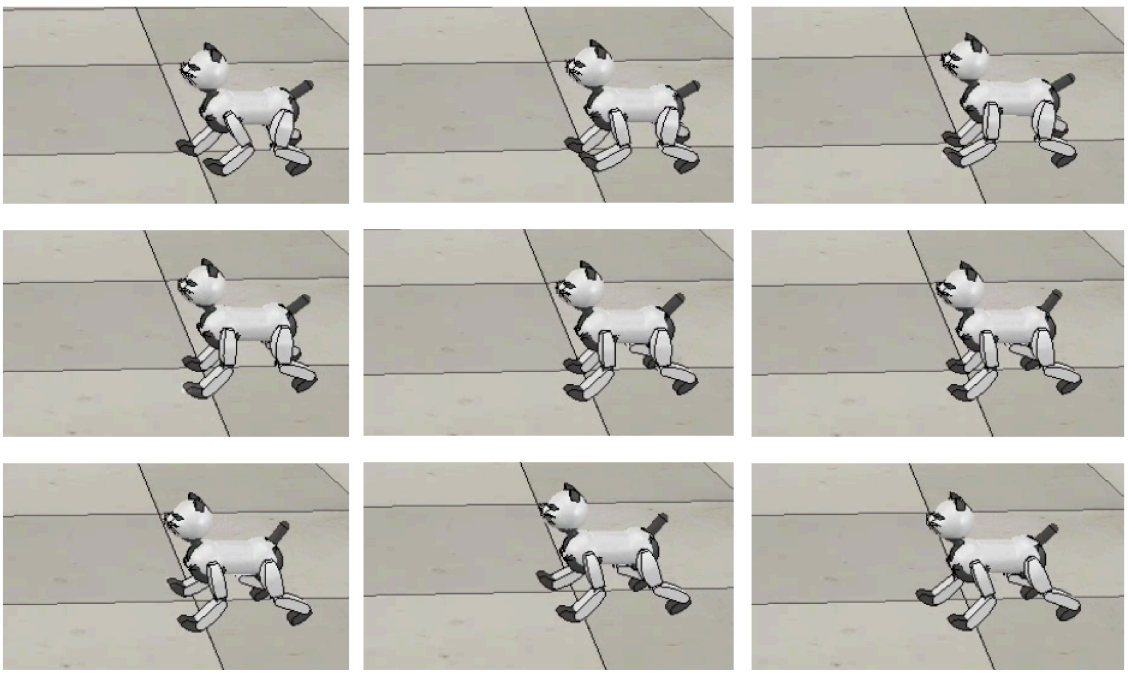


Figura 6.5: Robbie mentre cammina. Le foto sono state scattate con un intervallo pari a 0.2 secondi. Il controllore è implementato dalle CNN allenate tramite FORCE learning.

di altri per gestire la modulazione. Riprodurre traiettorie a velocità differenti è molto semplice e può essere effettuata dal modello di CPG presentato. Purtroppo caratteristiche specifiche del movimento umanoide non possono essere ricondotte a semplici fattori di scala, e in questo frangente tutti i modelli di Central Pattern Generators analizzati sono carenti, pertanto questo problema sarà presentato solo per quanto riguarda il reservoir computing.

Come già anticipato, Asti è l'unico robot, tra quelli implementati in V-REP, che permette modificare il passo in base a degli input. In particolare, esso ha 2 fattori che possono essere modulati: la velocità e l'ampiezza del passo. In questi esempi sono stati considerati tre tipi di modulazione: on/off e cambio di entrambi i valori modificabili di Asti.

In tutti questi test la rete ha le seguenti caratteristiche: $\delta = 1\text{ ms}, N = 1000, p = 0.1$. Il reservoir è formato da Leaky Integrator. Gli altri parametri variano in base alla funzione da generare, quindi sono riportati nella sezione apposita.

6.6.1 MODULAZIONE ON/OFF

Il test della modulazione acceso/spento è stato effettuato con i parametri: il tempo di membrana $\tau = 100\text{ ms}, \lambda = 1.9$, e i parametri per il training tramite regola EH: $\eta_0 = 5 \cdot 10^{-4}, T = 20, \alpha = 0.75$. Il tempo di training è di 980 secondi (simulati in 611.07) e il testing dura 50 secondi (simulati in 24.77 secondi).

Come si vede dalle immagini (in figura 6.6 sono rappresentati i primi 40 secondi del testing), la rete che si ottiene dall'addestramento con la regola EH produce un risultato piuttosto buono, ma si notano alcuni problemi: primo fra tutti, è presente una attivazione dove il sistema dovrebbe essere "spento". Questo è dovuto al fatto che la rete impara il rumore. Inoltre si vede che il sistema reagisce al cambio dell'input con un lieve ritardo, che causa un grosso sfasamento rispetto al segnale voluto. L'unione di questi due fattori rende il MSE inutile come cifra di merito, perché sarebbe molto alto (nelle prove effettuate è pari a circa 3 per tutte le uscite) e non molto indicativo dell'effettiva qualità della soluzione.

In questa prova si nota che, perché il training abbia successo, è necessario avere molte transizioni nell'input, non è sufficiente che a metà del tempo di training esso passi una sola volta da 0 a 1. Infatti, se ci fosse un unico cambiamento il sistema imparerebbe i pesi adatti a generare solo la seconda porzione del segnale.

Il training tramite FORCE learning con $\beta = 0.1$ produce dei risultati inusuali:

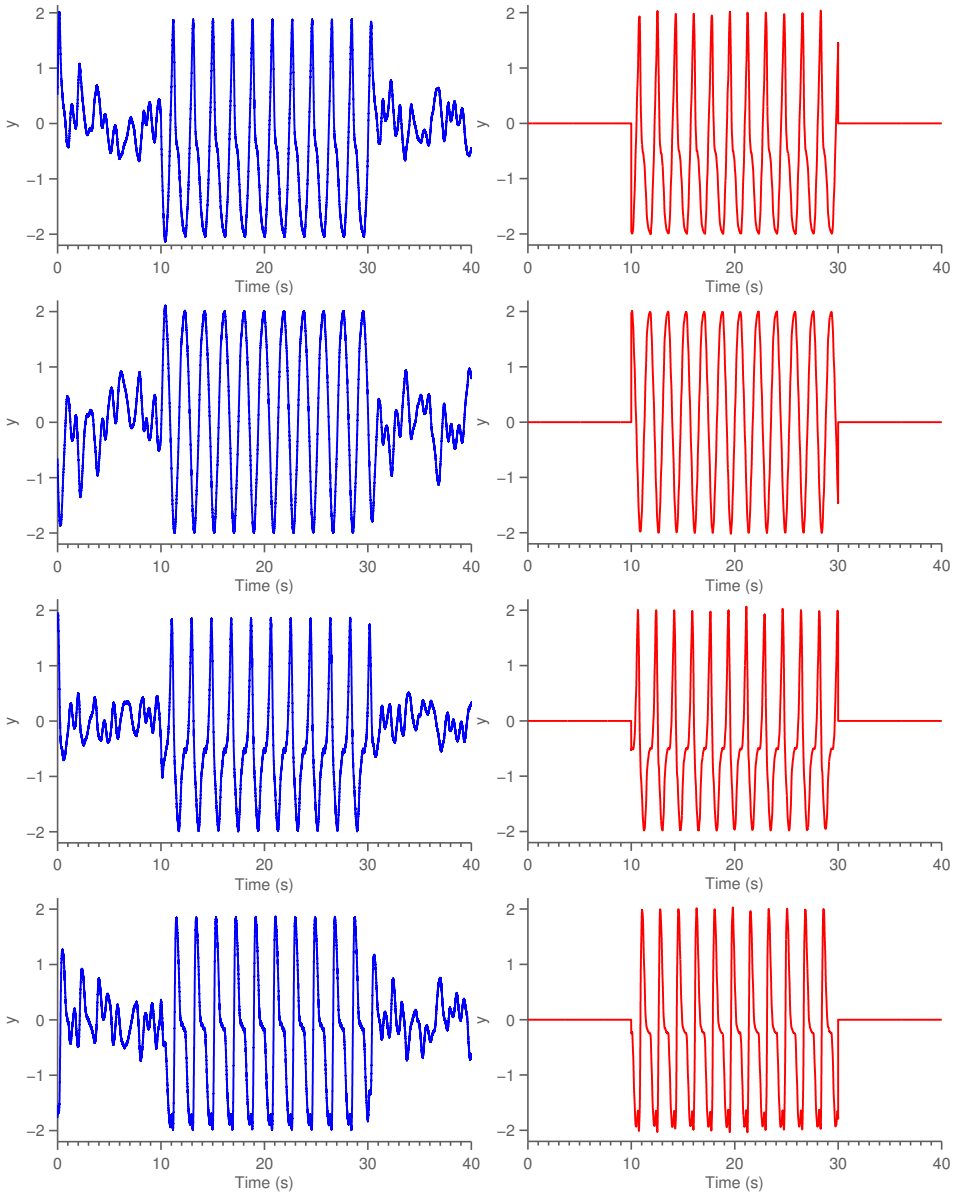


Figura 6.6: Risultato del training tramite regola EH di una rete per la modulazione on/off (solo gradi di libertà 1, 2, 3 e 6, dall'alto in basso). In blu l'uscita della rete, mentre in rosso, a destra, il segnale desiderato. I segnali sono riprodotti abbastanza bene, seppur leggermente sfasati, ma si vede che la rete non si annulla dove l'input è zero.

il segnale viene riprodotto correttamente dove esso è periodico, ma nelle parti la rete dovrebbe essere spenta il risultato è grossomodo costante ma diverso da 0. Questo problema potrebbe semplicemente essere causato da dei parametri non

impostati correttamente, però una investigazione completa della causa richiederebbe troppo tempo visto che utilizzando 80 secondi di dati per il training, esso impiega 432.12 secondi.

6.6.2 MODULAZIONE DELL'AMPIEZZA PASSO

Per la raccolta dati di questo problema, si fa variare il primo slider di Asti tra 200 e 500, cambiando ogni 3.5 secondi. Inoltre, l'input è stato modificato per essere compreso tra 0 e 2, dove 2 corrisponde ad un valore dello slider pari a 500, mentre 200 corrisponde a 0.8.

In questo problema si utilizzano 1300 secondi di dati per l'addestramento (che vengono simulati in 911.1 secondi), mentre la fase di testing dura 200 secondi, simulati in 101.32 secondi. Gli ulteriori parametri necessari sono: $\tau = 120$ ms, $\lambda = 2.2$ e per il training tramite regola EH si usano i seguenti valori: $\eta_0 = 5 \cdot 10^{-4}, T = 20, \alpha = 0.75$.

Anche qui, il risultato del training tramite regola EH (riportato in figura 6.7) è visivamente piuttosto buono, anche se si vedono comunque i problemi di sfasamento già notati prima. In questo caso però il segnale è riprodotto in maniera essenzialmente corretta per entrambi i valori dell'ingresso. Anche in questa situazione il calcolo dell'errore quadratico medio non ha molto significato. Calcolare il MSE_{dc} produce dei risultati nell'ordine di 10^{-1} per la maggior parte dei giunti (anche se per il giunto 2 vale 0.008 e per il giunto 5 0.0055).

Utilizzare il FORCE learning produce (training per 80 secondi, simulati in 437.2) una rete con errori piuttosto bassi. Anche con questo metodo, però, ci sono degli sfasamenti, che sono principalmente visibili nei punti in cui l'ingresso cambia valore. Per questo motivo calcolare il MSE non è molto utile. Calcolandolo nelle sezioni raggruppandole per valore di ingresso si ottiene un MSE_n nell'ordine di 10^{-1}.

6.6.3 MODULAZIONE DELLA VELOCITÀ

Anche per questa prova si possono fare le stesse considerazioni della modulazione della ampiezza del passo, nonostante in questo caso la rete debba modulare anche in frequenza, non solo in ampiezza.

La rete è configurata con gli ulteriori parametri: $\tau = 120$ ms, $\lambda = 2.2$ e per il training tramite regola EH si usano i seguenti valori: $\eta_0 = 5 \cdot 10^{-4}, T = 20, \alpha =$

0.75.

In questa prova il training con regola EH non ha dato risultati soddisfacenti, probabilmente perché la modulazione in frequenza è molto complicata da ottenere, specialmente in una configurazione in cui tutti i neuroni sono caratterizzati dalla stessa costante di tempo. Il training con l'algoritmo di FORCE learning con $\beta = 0.001$ invece produce un risultato molto buono. I primi 6 gradi di libertà sono riportati nella figura 6.8. Si vede solo che l'algoritmo non riesce a sopprimere molto bene il rumore presente sul reservoir che è riportato anche sull'uscita. Questo rumore potrebbe essere tolto per ottenere un risultato migliore, ma si è preferito mantenerlo, per essere in linea con l'idea più biologica dei neuroni. In generale non è un grosso problema, perché si può applicare facilmente una media mobile per ridurre il rumore.

Generalizzazione

In queste ultime due prove, inoltre, è interessante capire se la rete è in grado di generalizzare, ovvero se, dando un input nuovo, essa sia in grado di produrre un output corretto.

In particolare, è stata effettuata una seconda prova identica a quella riportata in sezione 6.6.2 (utilizzando anche la stessa configurazione per la rete). Nella prima fase, quindi, la rete è stata allenata modificando la dimensione del passo di Asti. Nella fase di testing, invece, viene fornito un ingresso nuovo, pari a 1.6 (che corrisponde ad un valore dello slider pari a 400). Si verifica che la forma dei segnali è diversa da quella attesa, ma provando effettivamente l'uscita sul robot simulato si vede che esso è comunque stabile, benché il passo non abbia la dimensione corretta.

6.7 Dati biologici

Si considera infine un esempio di camminata basata su dati ottenuti da soggetti umani. I dati utilizzati sono stati ricavati dal CMU Graphics Lab Motion Capture Database, che contiene registrazioni di vari movimenti, disponibili liberamente. Questo sistema cattura, a 120 Hz, 29 marker posizionati sul corpo del soggetto sotto registrazione, che corrispondono a 62 gradi di libertà.

Siccome anche in questo caso i dati disponibili sono piuttosto limitati (al

massimo 5 secondi, solo per pochi soggetti[2]) si possono addestrare reti neurali caotiche solo tramite FORCE learning o regressione e CPG tramite Fourier-CPG.

Per questo problema sono stati utilizzati i dati della camminata del soggetto 49, che consiste in circa 5 secondi di dati. Un esempio di questi dati si può vedere in figura 6.9a, che riporta i primi due gradi di libertà dell'anca.

6.7.1 CON FOURIERCPG

La configurazione adottata per il collegamento dei singoli CPG è uguale a quella utilizzata nei bipedi precedenti, ovvero CPG appartenenti alla stessa gamba sono collegati "in serie" più un accoppiamento tra il primo della gamba destra e della gamba sinistra. Sono stati utilizzati 5 oscillatori per CPG, $\gamma = 8, \mu = 1, \tau = 0.1, \delta = 1$ ms. Siccome le gambe in totale hanno 14 gradi di libertà, si considerano 14 CPG. Vengono utilizzati tutti i 5.2 secondi di dati disponibili per il training.

Il sistema addestrato è simulato per 30 secondi (impiegando 21.25 secondi). Durante la fase di testing, i cui risultati sono rappresentati in figura 6.9b, si vede che il training ha avuto successo solo per i gradi di libertà che mostrano un comportamento più periodico. Questo problema è causato dalle caratteristiche dei dati: sono pochi e piuttosto rumorosi, quindi l'algoritmo non riesce ad estrarre le componenti importanti.

6.7.2 CON CNN

Come già anticipato, si addestra la rete tramite FORCE Learning, perché la regola EH richiede molti più dati (sono necessari circa 200 secondi, come è stato notato nelle diverse prove effettuate). In questa prova si considera una CNN composta da 1000 neuroni leaky integrator, con $p = 0.1, \delta = 1$ ms, $\lambda = 1.5$ e $\tau = 50$ ms. Il parametro di regolarizzazione per il FORCE learning è $\beta = 0.1$. Il training viene completato in 30 secondi; la rete è successivamente eseguita per 30 secondi (simulati in 14.546 secondi).

I risultati ottenuti sono visivamente buoni: la rete produce un segnale che ricorda il movimento umano. Anche in questo caso le prime due uscite sono riportate in figura 6.9c. La simulazione della camminata è in figura 6.10. Invece il training tramite regressione produce dei sistemi spesso instabili.

[2] sono registrate anche camminate più lunghe (circa 40 secondi), ma con cambi di direzione.

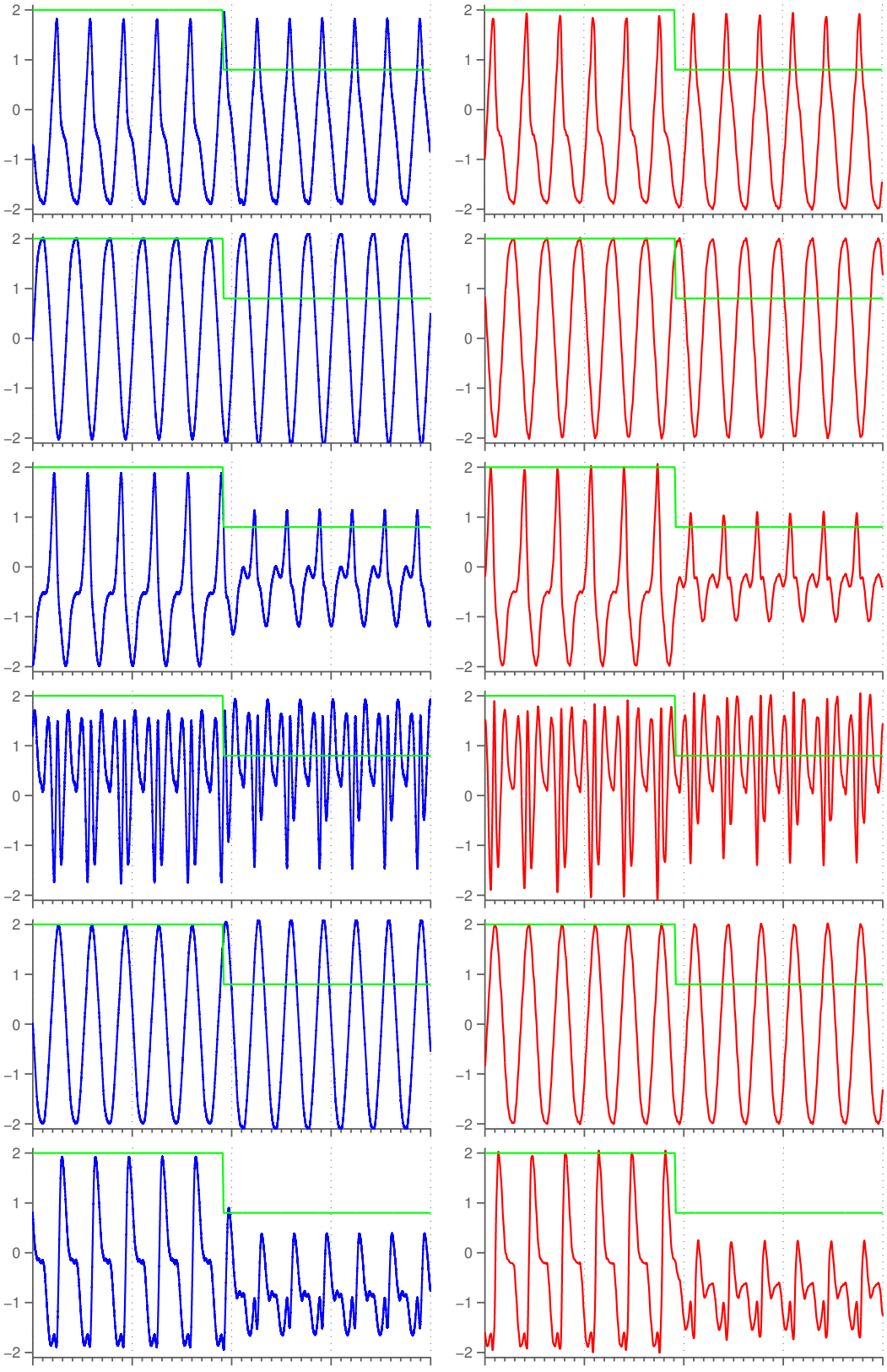


Figura 6.7: Risultato del training tramite regola EH di una rete per la modulazione della dimensione del passo. Per brevità sono riportati solo i primi 6 gradi di libertà, corrispondenti alla gamba sinistra. Sono inoltre rappresentati solo 20 secondi, di cui 10 prima del cambio dell'input e 10 dopo. A sinistra il segnale generato dalla rete, a destra il segnale desiderato. Sulle ascisse i tempi, l'uscita della rete sulle ordinate.

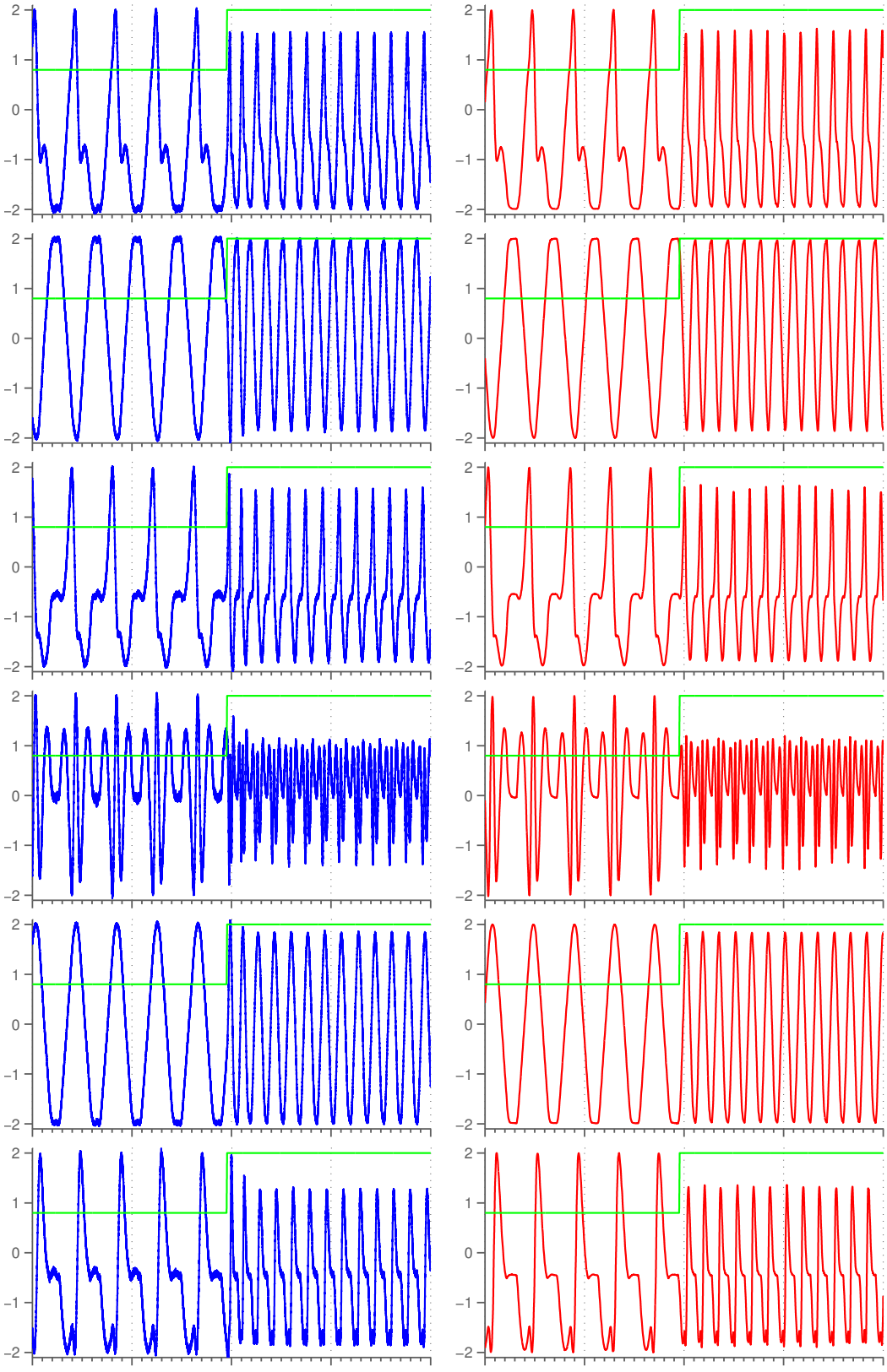
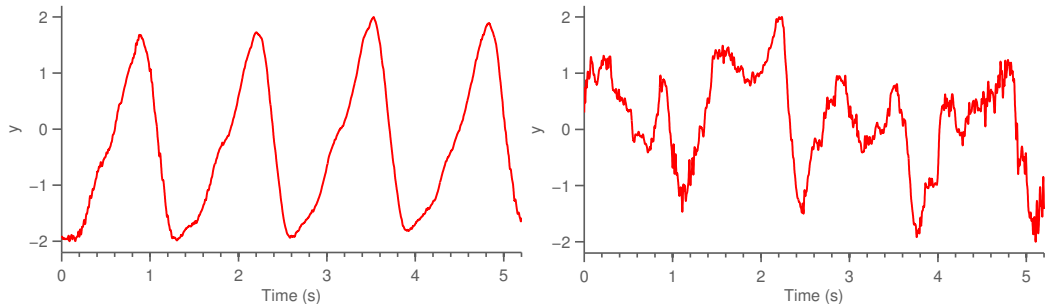
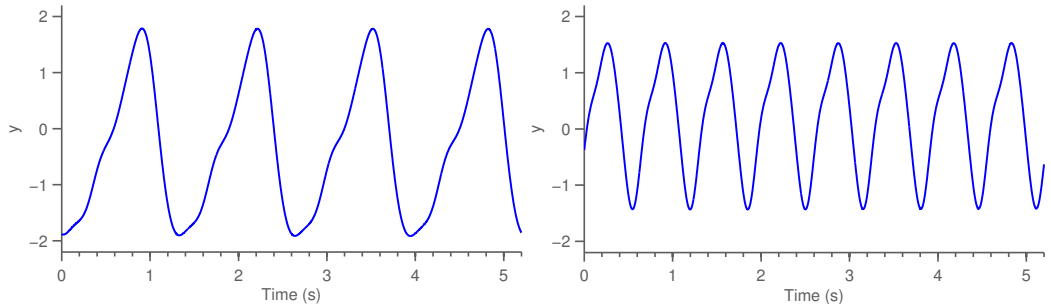


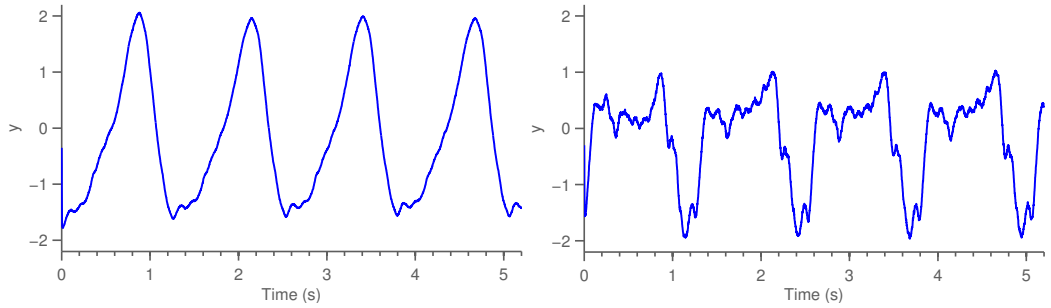
Figura 6.8: Risultato del training con FORCE di una rete per la modulazione della velocità del passo. Sono riportati solo i primi 6 giunti, ovvero la gamba sinistra. Per vedere meglio i risultati sono inoltre rappresentati solo 20 secondi, di cui 10 prima del cambio dell'input e 10 dopo. A sinistra, in blu, l'uscita della rete; a destra il segnale desiderato. Sulle x sono riportati i tempi, sulle y l'uscita della rete.



(a) Primi due gradi di libertà di una gamba.



(b) Segnale imparato dai CPG tramite Fourier-CPG. Come si vede, esso riesce ad apprendere bene la prima uscita, ma la seconda uscita è troppo rumorosa, e l'algoritmo non riesce ad approssimarla.



(c) Segnale imparato da CNN allenata tramite FORCE learning. In questo caso l'algoritmo di addestramento riesce ad estrarre un segnale dai dati, nonostante siano molto rumorosi.

Figura 6.9: Risultato dell'addestramento di CPG e CNN su dati biologici.

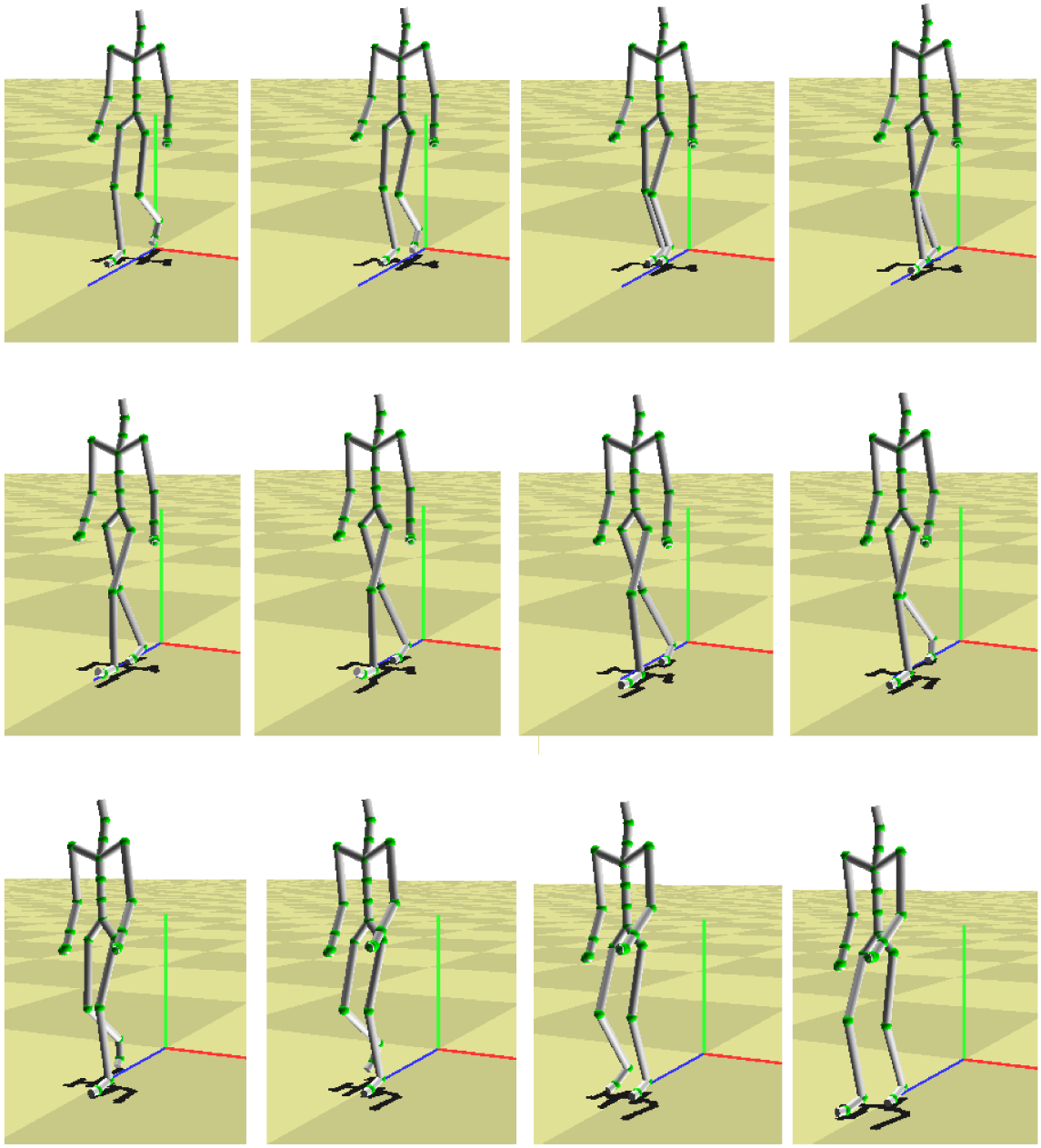


Figura 6.10: Camminata umana imparata tramite FORCE learning. Si vede circa 1 secondo di simulazione, tra una fotografia e la successiva sono passati 80 ms.

CAPITOLO 7

Discussione e conclusioni

In questa sezione riepilogo i risultati ottenuti in questo lavoro, e riporto le conclusioni che ho derivato dopo l'uso dei Central Pattern Generator e le Reti Neurali Caotiche.

7.1 Risultati principali del lavoro

Lo scopo principale di questo lavoro di tesi è stato l'introduzione delle reti neurali caotiche come controllori della camminata di un robot. In particolare si sono confrontati i Central Pattern Generator e le reti neurali caotiche. I primi sono una soluzione relativamente recente, sviluppata in ambito accademico, che però manca ancora di applicazioni pratiche. Invece le reti neurali caotiche sono state usate in passato principalmente per applicazioni di machine learning. Un primo risultato ottenuto riguarda i parametri delle CNN: da prove sperimentali si nota che essi influenzano in maniera sensibile le performance, ed empiricamente ho trovato delle regole per determinare dei valori iniziali da assegnarvi.

Per raggiungere lo scopo del lavoro è stato necessario sviluppare una soluzione software per le reti neurali caotiche. Attualmente la libreria implementata fornisce tutte le funzionalità che sono necessarie e il design rende facile le eventuali estensioni. Anche le performance in termini di tempo di computazione sono relativamente buone per una prima versione, considerando anche che si utilizza un linguaggio come MATLAB.

In questo lavoro sono stati provati 3 modelli di robot: 2 bipedi e 1 quadrupede. Inizialmente si è utilizzato un ambiente simulato, per provare in modo semplice i risultati. Una volta ottenuti dei robot in grado di muoversi in simulazione, si è pilotato la versione fisica del robot NAO. Anche in questo caso sono stati ottenuti dei risultati promettenti, seppur il robot mostrasse un comportamento diverso da quello desiderato (un problema che è causato da fattori esterni e non controllabili).

Un altro risultato molto interessante che è stato raggiunto riguarda l'approssimazione delle traiettorie del moto umano: le reti neurali caotiche si sono dimostrate in grado di imparare anche questi segnali.

7.2 Central Pattern Generator

Il problema principale che si riscontra lavorando con i Central Pattern Generator è che mancano i metodi per disegnare una rete e trovare i parametri che permettono di arrivare al segnale voluto. Per risolvere questo problema sono stati introdotti i Programmable Central Pattern Generator, che prescrivono una topologia e un metodo per il training.

I PCPG però non sono sempre la soluzione ottimale: si verifica che il metodo di training è caratterizzato da una convergenza piuttosto lenta, e che dipende dalle condizioni iniziali del problema, che quindi rende problematico fornire garanzie sulla convergenza. Con lo scopo di attenuare il problema e arrivare ad un metodo più pratico, si è introdotto un algoritmo di training offline che permette di calcolare i parametri necessari al Central Pattern Generator in maniera molto veloce, ovvero l'algoritmo Fourier-PCPG.

Come hanno evidenziato i risultati riportati nella sezione precedente, i CPG sono adatti a generare, con errori molto bassi, le traiettorie per pilotare i giunti. Come però è stato notato, i modelli di Central Pattern Generator esistenti non permettono la modulazione del segnale in funzione di un ingresso. In generale i CPG hanno un supporto per la modulazione minimo: si può modificare facilmente solo l'ampiezza o la frequenza. Cambiare questi valori può essere una soluzione valida per robot quadrupedi con un grado di libertà per gamba, in cui, ovviamente, aumentare la frequenza di oscillazione di tutti gli arti si traduce in un aumento di velocità. Invece, nei robot più complicati è necessario che la forma d'onda venga modificata in maniera considerevole per riuscire a mantenere l'equilibrio.

Tra le caratteristiche positive di questa rete sono sicuramente da annoverare la complessità di simulazione bassa e la semplicità dei modelli. La prima permette una implementazione efficiente e quindi adatta anche a robot dotati di elaboratori poco potenti. Un altro fattore che può aiutare l'implementazione su dispositivi portatili è il fatto che il modello è semplicemente distribuito, e ogni CPG può utilizzare in modo banale un calcolatore differente. Inoltre, la semplicità dei modelli, dovuta al fatto che le frequenze e le ampiezze vengono codificate direttamente

all'interno dell'oscillatore e che le interazioni tra i diversi oscillatori sono ridotte al minimo, rende il modello semplice da comprendere.

Confrontando i Central Pattern Generator con ZMP è stato verificato che i CPG permettono di generare camminate dinamiche, mentre ZMP può calcolare solo camminate statiche. ZMP inoltre permette di calcolare le traiettorie solo offline. Purtroppo però, con i CPG non si può dimostrare la stabilità della camminata, è necessario provare su di un robot sia esso reale o simulato.

Infine, per quanto riguarda le primitive di movimento, i due metodi non hanno grosse differenze. Anzi, si potrebbe addirittura pensare ad un metodo ibrido tra i due: generare le traiettorie tramite i Central Pattern Generator, e utilizzare il planner per concatenarle.

7.3 Reti Neurali Caotiche

Dagli esperimenti si verifica che le reti neurali caotiche sono adatte a generare le traiettorie per pilotare i giunti con errori relativamente bassi. Si può avere qualche minore errore causato da leggere derive, ma in generale si nota che in simulazione la stabilità del robot non ne risente in maniera troppo negativa. Naturalmente per essere sicuri che questo inconveniente sia realmente minore si devono provare le traiettorie sul robot preso in considerazione, sia esso fisico o simulato.

Se si considerano obiettivi più complicati, come ad esempio i task di modulazione presentati in sezione 6.6, si nota che le CNN riescono a raggiungere errori molto validi, seppur non ottimi come nei problemi più semplici. Nonostante la presenza di errori non trascurabili, le reti neurali caotiche sono sicuramente più interessanti rispetto ai Central Pattern Generator, in quanto questi ultimi non riescono ad approssimare funzioni qualunque. Si verifica inoltre che le reti neurali caotiche funzionano meglio in presenza di rumore rispetto ai CPG.

Naturalmente questa maggiore potenza di computazione viene ad un costo abbastanza considerevole in termini di tempo d'esecuzione: infatti le CNN hanno una complessità quadratica, quindi usano molto più tempo rispetto ai Central Pattern Generator. Questo potrebbe renderle meno adatte all'implementazione su dei robot fisici, dove la potenza computazionale è limitata.

Il problema principale del reservoir computing risiede però nei parametri: i modelli presentati sono caratterizzati da un numero considerevole di parametri (4 per il modello principale utilizzato) e anche i metodi di training posseggono

valori da scegliere. La difficoltà consiste nel trovare i parametri migliori sia per il reservoir sia per il problema considerato, considerando anche che piccole variazioni nel valore di ciascuno di essi influenzano in maniera sensibile la qualità dei risultati ottenuti. Per questo ho derivato, sia in maniera empirica sia tramite l'utilizzo di teoria dei sistemi, delle linee guida per trovare dei valori di partenza validi per questi parametri. Le regole trovate sono delle approssimazioni, in quanto una analisi esatta del modello è molto difficile (se non impossibile), ma dalle prove effettuate si nota che sono piuttosto valide. Per quanto riguarda invece gli algoritmi di training, si può sottolineare che l'algoritmo di addestramento tramite regola EH non ha bisogno di particolari aggiustamenti di parametri, si verifica che quelli standard spesso funzionano. Un discorso diverso invece va fatto per il FORCE learning e la regressione. Entrambi posseggono un unico parametro da modificare: nelle nostre prove è stato trovato tramite line search (cercando ovvero in un intervallo di valori quale producesse l'errore minore in fase di testing). Questo metodo è piuttosto lento perché richiede che la rete venga addestrata molte volte, ma è molto semplice e non richiede particolar accorgimenti.

Molti dei problemi che le reti neurali ricorrenti hanno sono in realtà condivisi con le reti neurali feed-forward (oltre alla definizione migliore dei parametri, di cui ho già parlato), per esempio sono un modello black-box. Questo fatto ha diverse conseguenze sia dal lato pratico sia dal lato più teorico. In particolare, è difficile capire come risolvano un problema o quale parametro sia la causa di un errore nell'output. Inoltre quando si ottiene una rete che nella fase di testing funziona non si può essere sicuri che essa generalizzi (ovvero abbia un output valido anche su ingressi nuovi), o addirittura non si riesce nemmeno a garantire che la traiettoria generata sia sempre "vicina" a quella desiderata.

Anche per questo modello nel confronto coi metodi più classici si possono ripetere molte delle considerazioni effettuate anche per i Central Pattern Generator: si tratta di un metodo più generale rispetto a ZMP, ma anche qui la stabilità del robot controllato può essere problematica, come è evidenziato in una prova (per esempio, nella prova in sezione 6.4 si verifica che la camminata generata per NAO non è sempre stabile). Rispetto invece alle primitive di movimento si può sottolineare una interessante corrispondenza: così come le primitive di movimento vengono concatenate da un planner, nel reservoir computing si hanno tante funzioni base che vengono sottoposte ad una combinazione lineare dal readout. Però, trovare una base significativa per le primitive di movimento non è un problema banale (infatti sono quasi sempre scelte codificate a mano da un esperto), mentre

nel reservoir computing le basi sono presenti nel reservoir e vengono selezionate dall'algoritmo di training.

In definitiva si ritengono le reti neurali caotiche un metodo molto più pratico ed efficace dei Central Pattern Generator nonostante alcuni svantaggi che possono essere attribuiti sia alle caratteristiche del modello sia alla sua novità.

7.4 Sviluppi futuri

Per gli sviluppi futuri si possono identificare alcune idee a breve termine. Si potrebbe aggiungere alla rete caotica degli ingressi forniti da sensori (come sensori di forza o di ostacoli) al fine di risolvere problemi più interessanti (come le camminate su terreni accidentati). Si potrebbe anche procedere su di un robot più semplice (per esempio un quadrupede) ed utilizzare dei dati reali ottenuti da un animale. Ulteriormente si potrebbe iniziare uno studio più sistematico di come i parametri influenzano le capacità della rete, oppure studiare dei modelli di readout leggermente più complessi.

Per quanto riguarda la libreria software sviluppata, alcuni possibili miglioramenti includono la riscrittura in un linguaggio compilato. Benché non si tratti di una operazione ad alta priorità, una implementazione in (per esempio) C++ potrebbe migliorare ulteriormente le performance, e si potrebbe aggiungere la possibilità di interfacciarsi con Python per rendere possibile l'utilizzo di ROS (*Robot Operating System*, un framework standard per la robotica) in modo tale di semplificare la comunicazione con un robot fisico.

Una altra possibilità è invece l'implementazione in hardware, su FPGA, nonostante questa strada potrebbe essere più complicata viste le dimensioni delle reti che solitamente si considerano e il fatto che si necessiti ancora di regolazioni per trovare i parametri, che potrebbe rendere il ciclo di sviluppo molto lento.

APPENDICE A

Comunicazione con V-REP

In questa appendice spiego brevemente alcuni dettagli di più basso livello riguardanti la comunicazione con V-REP. In particolare, dettaglio i gradi di libertà dei robot considerati e le funzioni sviluppate per semplificare la comunicazione con il simulatore.

A.1 Gradi di libertà dei robot

In V-REP i gradi di libertà di Asti sono identificati con nomi da `leftLegJoint0` a `leftLegJoint5` per la gamba sinistra, e analogamente `rightLegJointK` per l'arto destro.

Per NAO i giunti hanno nome `LHipYawPitch3`, `LHipRoll3`, `LHipPitch3`, `LKneePitch3`, `LAnklePitch3`, `LAnkleRoll3` nella gamba sinistra, e i giunti che si trovano nella gamba destra hanno nomi analoghi, che iniziano con `R`. La versione fisica del robot ha gli stessi nomi, ma senza il numero finale.

In Robbie, i 3 gradi di libertà nella zampa anteriore sinistra si chiamano `robbieLegJoint1`, `robbieLegJoint2`, `robbieLegJoint3`. Nella zampa posteriore sinistra si chiamano `robbieLegJoint1#0`, `robbieLegJoint2#0` e `robbieLegJoint3#0`; analogamente nella anteriore destra e posteriore destra (rispettivamente `robbieLegJointK#1` e `robbieLegJointK#2`)

A.2 Lettura e preparazione dati

Per registrare i movimenti dei robot si è effettuata una simulazione con passo di integrazione di 25 ms, registrando le posizioni dei vari giunti ogni 50 ms.

Per questo scopo è stata definita una funzione MATLAB che effettua la comunicazione con un server V-REP:

`values = logJointValues(jointsNames, pauseTime, numberOfReadings, sliders)`. Questa funzione:

1. inizia una connessione col server V-REP,
2. avvia la simulazione,
3. legge le posizioni dei `jointsNames` specificati,
4. se `sliders` è fornito, inoltre, modifica il valore degli ingressi supportati dal robot (solo Asti, fra i robot provati, supporta questo parametro),
5. si ferma `pauseTime` millisecondi, prima di ricominciare dal punto 3. Se ha letto `numberOfReadings` valori, la funzione termina e ritorna i valori letti in un vettore.

Una volta raccolti i dati, essi devono essere sovra-campionati al fine di portarli ad avere lo stesso δ della rete e normalizzati nell'intervallo $[-2, 2]$ (le motivazioni che rendono necessario questo procedimento sono state discusse nella sezione 3). La funzione sviluppata per preparare i dati si chiama `[newData, normalizationConstants] = prepareData(data, resamplingRate)`.

A.3 Controllo del robot

Una volta generata una nuova traiettoria tramite la rete neurale si può passare al controllo del robot. La rete, al termine del training, produrrà un segnale normalizzato nell'intervallo di riferimento ($[-2, 2]$) ed eventualmente sovra-campionato, che dovrà essere riportato all'ampiezza e frequenza di campionamento corretta prima di essere fornito al simulatore per pilotare il robot. A questo scopo è stata sviluppata la funzione `trajectory = invertDataTransform(data, normalizationConstants, downsampleRate)`.

Una volta riportati i dati nel corretto sistema di riferimento, si può controllare l'umanoide. La funzione apposita si chiama `setJointsValues(jointsNames, pauseTime, values)`. Come la funzione speculare per la registrazione, essa apre una connessione col server V-REP, inizia una simulazione, e, ogni `pauseTime` millisecondi fornisce nuovi valori di posizione ai giunti specificati in `jointsNames` del robot. Per utillizzare questa funzione è importante verificare che i giunti del robot siano controllabili, siano ovvero dei giunti in modalità ibrida. Inoltre non deve agire alcun altro controllore: si deve quindi eliminare il controllore implementato.

Bibliografia

[1] Janczak Andrzej. *Identication of Nonlinear Systems Using Neural Networks and Polynomial Models*. Springer-Verlag Berlin Heidelberg, 2004. DOI: 10.1007/b98334 (cit. alle pp. 20, 44).

[2] Eric Antonelo, Benjamin Schrauwen, Xavier Dutoit, Dirk Stroobandt e Marnix Nuttin. «Event Detection and Localization in Mobile Robot Navigation Using Reservoir Computing». In: *Artificial Neural Networks ICANN 2007* 4669 (2007), pp. 660–669. DOI: 10.1007/978-3-540-74695-9_68 (cit. a p. 21).

[3] Atılım Güneş Baydin. «Evolution of central pattern generators for the control of a five-link bipedal walking mechanism». In: *Paladyn, Journal of Behavioral Robotics* 3.1 (2012), p. 10. DOI: 10.2478/s13230-012-0019-y. arXiv: 0801.0830 (cit. a p. 9).

[4] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., 1995 (cit. a p. 30).

[5] Joschka Boedecker, Oliver Obst, Norbert Michael Mayer e Minoru Asada. «Studies on Reservoir Initialization and Dynamics Shaping in Echo State Networks». In: *Proceedings of the European Symposium on Neural Networks ESANN* April (2009), pp. 1–6 (cit. a p. 25).

[6] Pierre Courrieu. «Fast Computation of Moore-Penrose Inverse Matrices». In: *Neural Information Processing - Letters and Reviews* 8.2 (2008), pp. 25–29. arXiv: 0804.4809 (cit. a p. 47).

[7] Armando C. De Pina Filho, Max S. Dutra e Luciano S. C. Raptopoulos. «Modeling of a bipedal robot using mutually coupled Rayleigh oscillators.» In: *Biological cybernetics* 92.1 (2005), pp. 1–7. DOI: 10.1007/s00422-004-0531-1 (cit. a p. 9).

[8] Michele Folgheraiter. «A combined B-spline-neural-network and ARX model for online identification of nonlinear dynamic actuation systems». In: *Neurocomputing* (2015), pp. 1–10. DOI: 10.1016/j.neucom.2015.10.077 (cit. a p. 22).

[9] Michele Folgheraiter. «Adaptive Joint Trajectory Generator Based on a Chaotic Recurrent Neural Network». In: 5th Intern (2015), pp. 285–290 (cit. a p. 22).

[10] Andrej Gams, Auke Jan Ijspeert, Stefan Schaal e Jadran Lenarčič. «On-line learning and modulation of periodic movements with nonlinear dynamical systems». In: *Autonomous Robots* 27.1 (2009), pp. 3–23. DOI: 10.1007/s10514-009-9118-y (cit. a p. 16).

[11] Stuart Geman. «The Spectral Radius of Large Random Matrices». In: *The Annals of Probability* 14.4 (1986), pp. 1318–1328. DOI: 10.1214/aop/1176992372 (cit. a p. 25).

[12] Alireza Goudarzi e Darko Stefanovic. «Towards a Calculus of Echo State Networks». In: *Procedia Computer Science* 41 (2014), pp. 176–181. DOI: 10.1016/j.procs.2014.11.101. arXiv: 1409.0280 (cit. a p. 34).

[13] David G. Hagner, Mohamad H. Hassoun e Paul B. Watta. «Comparison of Recurrent Neural Networks for Trajectory Generation». In: *Recurrent Neural Networks: Design and Application*. International Series on Computational Intelligence. CRC Press, dic. 2001. Cap. 10, pp. 243–276. DOI: 10.1201/9781420049176.ch10 (cit. alle pp. 20, 44).

[14] Gregor M. Hoerzer, Robert Legenstein e Wolfgang Maass. «Emergence of Complex Computational Structures From Chaotic Neural Networks Through Reward-Modulated Hebbian Learning». In: *Cerebral Cortex* 24.3 (2014), pp. 677–690. DOI: 10.1093/cercor/bhs348 (cit. alle pp. 31–33, 42, 55).

[15] Scott L. Hooper. «Central Pattern Generators». In: *Encyclopedia of Life Sciences* (2001), pp. 1–9 (cit. a p. 7).

[16] Jose Hugo, Cesar Torres-Huitzil e Bernard Girau. «Configurable Embedded CPG-based Control for Robot Locomotion». In: *International Journal of Advanced Robotic Systems* 9 (2012), p. 1. DOI: 10.5772/50985 (cit. a p. 9).

[17] Auke Jan Ijspeert. «Central pattern generators for locomotion control in animals and robots: A review». In: *Neural Networks* 21.4 (2008), pp. 642–653. DOI: 10.1016/j.neunet.2008.03.014 (cit. a p. 9).

[18] Herbert Jaeger. «Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication». In: *Science* 304.5667 (2004), pp. 78–80. DOI: 10.1126/science.1091277 (cit. alle pp. 22, 35, 36).

[19] Herbert Jaeger. *The "echo state" approach to analysing and training recurrent neural networks - with an Erratum note*. Rapp. tecn. GMD - German National Research Institute for Computer Science, 2010 (cit. alle pp. 20, 25).

[20] Arthur D. Kuo. «The relative roles of feedforward and feedback in the control of rhythmic movements.» In: *Motor control* 6.2 (2002), pp. 129–145 (cit. a p. 7).

[21] Cai Li, Robert Lowe e Tom Ziemke. «A novel approach to locomotion learning: Actor-Critic architecture using central pattern generators and dynamic motor primitives». In: *Frontiers in Neurorobotics* 8.October (2014), pp. 1–17. DOI: 10.3389/fnbot.2014.00023 (cit. a p. 3).

[22] Liang Li, Chen Wang, Guangming Xie e Hong Shi. «Digital Implementation of CPG controller in AVR system». In: *Control Conference (CCC), 2014 33rd Chinese*. IEEE, 2014, pp. 2–7. DOI: 10.1109/ChiCC.2014.6896390 (cit. a p. 9).

[23] Xiaojun Li e Lin Li. «Efficient implementation of FPGA based central pattern generator using distributed arithmetic». In: *IEICE Electronics Express* 8.21 (2011), pp. 1848–1854. DOI: 10.1587/elex.8.1848 (cit. a p. 9).

[24] Xiaowei Lin, Zehong Yang e Yixu Song. «Short-term stock price prediction based on echo state networks». In: *Expert Systems with Applications* 36.3 (2009), pp. 7313–7317. DOI: 10.1016/j.eswa.2008.09.049 (cit. a p. 44).

[25] Mantas Lukoševičius. «A practical guide to applying echo state networks». In: *Neural Networks: Tricks of the Trade, Reloaded* (2012). DOI: 10.1007/978-3-642-35289-8-36 (cit. a p. 26).

[26] Mantas Lukoševičius e Herbert Jaeger. «Reservoir computing approaches to recurrent neural network training». In: *Computer Science Review* 3.3 (2009), pp. 127–149. DOI: 10.1016/j.cosrev.2009.03.005 (cit. a p. 25).

[27] Wolfgang Maass, Prashant Joshi e Eduardo D. Sontag. «Computational Aspects of Feedback in Neural Circuits». In: *PLoS Computational Biology* 3.1 (2007), e165. DOI: 10.1371/journal.pcbi.0020165 (cit. a p. 26).

[28] Wolfgang Maass, Henry Markram e Thomas Natschläger. «The "liquid computer": A novel strategy for real-time computing on time series». In: *Special Issue on Foundations of Information Processing of TELEMATIK* 8.1 (2002), pp. 39–43 (cit. a p. 20).

[29] Takamitsu Matsubara, Jun Morimoto, Jun Nakanishi, Masa-aki Sato e Kenji Doya. «Learning CPG-based biped locomotion with a policy gradient method». In: *5th IEEE-RAS International Conference on Humanoid Robots, 2005.* Vol. 2005. 2005, pp. 208–213. DOI: 10.1109/ICHR.2005.1573569 (cit. a p. 9).

[30] Jun Nakanishi, Jun Morimoto, Gen Endo, Gordon Cheng, Stefan Schaal e Mitsuo Kawato. «Learning from demonstration and adaptation of biped locomotion». In: *Robotics and Autonomous Systems* 47.2-3 (2004), pp. 79–91. DOI: 10.1016/j.robot.2004.03.003 (cit. a p. 16).

[31] Mustafa C. Ozturk, Dongming Xu e José C. Príncipe. «Analysis and design of echo state networks.» In: *Neural computation* 19.1 (2007), pp. 111–38. DOI: 10.1162/neco.2007.19.1.111 (cit. a p. 36).

[32] Mattia Pirotti. «Central pattern generator per costruire le andature del robot quadrupede warugadar». Tesi di Laurea Magistrale. Politecnico di Milano, 2010, p. 133 (cit. a p. 9).

[33] Paul G. Ploger, Adriana Arghir, Tobias Giinther e Ramin Hosseiny. «Echo State Networks for Mobile Robot Modeling and Control». In: *RoboCup 2003 Robot Soccer World Cup VII* Robocup 20 (2004), pp. 157–168. DOI: 10.1007/978-3-540-25940-4_14 (cit. a p. 21).

[34] Alfio Quarteroni, Fausto Saleri e Paola Gervasio. *Scientific Computing with MATLAB and Octave.* 4th. Vol. 2. Springer Berlin Heidelberg, 2014. DOI: 10.1007/978-3-642-45367-0 (cit. alle pp. 19, 46).

[35] René Felix Reinhart. «Reservoir Computing with Output Feedback». In: *KI - Künstliche Intelligenz* 26.4 (2012), pp. 415–416. DOI: 10.1007/s13218-012-0187-2 (cit. a p. 26).

[36] Ludovic Righetti. «Control of Legged Locomotion using Dynamical Systems: Design Methods and Adaptive Frequency Oscillators». Tesi di dott. École Polytechnique Fédérale de Lausanne, 2008. DOI: 10.5075/epfl-thesis-4222 (cit. alle pp. 9, 12, 17).

[37] Serge Rossignol. «Locomotion and its recovery after spinal injury in animal models.» In: *Neurorehabilitation and neural repair* 16.2 (2002), pp. 201–6 (cit. a p. 8).

[38] Stefan Schaal. «Dynamic Movement Primitives – A Framework for Motor Control in Humans and Humanoid Robotics». In: *Adaptive Motion of Animals and Machines* (2002), pp. 261–280. DOI: 10.1007/4-431-31381-8_23 (cit. a p. 4).

[39] Mark D. Skowronski e John G. Harris. «Automatic speech recognition using a predictive echo state network classifier». In: *Echo State Networks and Liquid State Machines* 20.3 (2007), pp. 414–423. DOI: 10.1016/j.neunet.2007.04.006 (cit. a p. 22).

[40] Haim Sompolinsky, Andrea Crisanti e Hans-Jurgen Sommers. «Chaos in random neural networks.» In: *Physical review letters* 61.3 (1988), pp. 259–262. DOI: 10.1103/PhysRevLett.61.259 (cit. a p. 25).

[41] Julien Clinton Sprott. *Chaos and Time-Series Analisys*. 1st. Oxford University Press, Inc., 2003 (cit. a p. 37).

[42] Julien Clinton Sprott. «Is chaos good for learning?» In: *Nonlinear dynamics, psychology, and life sciences* 17.2 (2013), pp. 223–32 (cit. a p. 20).

[43] Xiao-chuan Sun, Hong-yan Cui, Ren-ping Liu, Jian-ya Chen e Yun-jie Liu. «Modeling deterministic echo state network with loop reservoir». In: *Journal of Zhejiang University SCIENCE C* 13.9 (2012), pp. 689–701. DOI: 10.1631/jzus.C1200069 (cit. a p. 44).

[44] Malur K. Sundareshan, Yee Chin Wong e Thomas Condarcure. «Training Algorithms for Recurrent Neural Nets That Eliminate The Need for Compution of Error Gradients With Application to Trajectory Production Problem». In: *Recurrent Neural Networks: Design and Applications*. International Series on Computational Intelligence. CRC Press, dic. 1999. Cap. 11. DOI: 10.1201/9781420049176.ch11 (cit. alle pp. 20, 44).

[45] David Sussillo e Laurence F. Abbott. «Generating Coherent Patterns of Activity from Chaotic Neural Networks». In: *Neuron* 63.4 (2009), pp. 544–557. DOI: 10.1016/j.neuron.2009.07.018 (cit. alle pp. 21, 31).

[46] David Verstraeten, Benjamin Schrauwen e Dirk Stroobandt. «Reservoir-based techniques for speech recognition». In: *The 2006 IEEE International Joint Conference on Neural Network Proceedings* (2006), pp. 1050–1053. DOI: 10.1109/IJCNN.2006.246804 (cit. alle pp. 20, 22).

[47] David Verstraeten, Benjamin Schrauwen, Dirk Stroobandt e J. Van Campenhout. «Isolated word recognition with the Liquid State Machine: a case study». In: *Information Processing Letters* 95.6 (2005), pp. 521–528. DOI: 10.1016/j.ipl.2005.05.019 (cit. a p. 21).

[48] Donald M. Wilson. «The central nervous control of flight in a locust». In: *Journal of Experimental Biology* 38.2 (1961), pp. 471–490 (cit. a p. 7).

[49] Francis Wyffels e Benjamin Schrauwen. «A comparative study of Reservoir Computing strategies for monthly time series prediction». In: *Neurocomputing* 73.10-12 (2010), pp. 1958–1964. DOI: 10.1016/j.neucom.2010.01.016 (cit. alle pp. 22, 35).

[50] Fumitaka Yamaoka, Takayuki Kanda, Hiroshi Ishiguro e Norihiro Hagita. «Interacting with a human or a humanoid robot?» In: *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on Intelligent Robots and Systems* (2007), pp. 2685–2691. DOI: 10.1109/IROS.2007.4399183 (cit. a p. 1).