Università di Pisa

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

Tesi di laurea

Modelli neurali costruttivi di tipo Reservoir Computing per domini strutturati

Giulio Visco

Relatori	Controrelatore
Prof. Alessio Micheli	Prof. Stefano Chessa
Dott. Claudio Gallicchio	

Anno Accademico 2010/2011

In God we trust, all others bring data.

— William Edwards Deming

Sommario

Il presente lavoro di tesi introduce e discute nuovi modelli di Reti Neurali Ricorsive per l'apprendimento supervisionato di trasduzioni su grafi. Due sono i maggiori contributi apportati: l'adozione di un approccio costruttivo, e l'introduzione di un meccanismo stabile di output-feedback, entrambi innovativi nell'ambito del Reservoir Computing a cui si rifanno i modelli considerati. La combinazione di una strategia costruttiva e dell'utilizzo di modelli di Reservoir Computing ha inoltre permesso la realizzazione di modelli molto efficienti dal punto di vista computazionale.

I modelli e le strategie individuate si configurano come uno strumento utile e flessibile nel trattamento di domini complessi attraverso tecniche di Machine Learning, e propongono soluzioni ad alcuni dei problemi aperti nell'ambito del Reservoir Computing.

L'analisi sperimentale svolta riguarda l'apprendimento di trasduzioni strutturali da dataset reali appartenenti all'ambito della Chemioinformatica.

Ringraziamenti

Il mio primo ringraziamento va al professor Alessio Micheli, per avermi introdotto al mondo del Machine Learning e per avermi coinvolto in un'esperienza didattica diversa e stimolante, fatta di idee e di discussioni, di ricerche e di ricerca. Conserverò un ricordo prezioso delle lezioni, delle riunioni, dei consigli e delle piacevoli chiacchierate.

Un ringraziamento particolare va anche a Claudio Gallicchio che mi ha guidato in questo lungo e non sempre facile percorso, tendendomi la mano o pungolandomi secondo necessità. Tutto quello che segue è anche merito suo, e chissà se troverò mai il modo di ripagarlo del supporto ricevuto — tecnico e soprattutto umano — o anche solo di farmi perdonare dei grattacapi che ha dovuto subire per vedermi arrivare in fondo a questa tesi.

Ci sarebbero altre persone da ringraziare, e non sarebbero poche: amici vicini e lontani, recenti o di vecchia data, parenti, coinquilini e case, qualche professore, qualche luogo. A molti devo un ringraziamento per i consigli, per avermi supportato o semplicemente distratto, per aver ascoltato le mie lamentele o per aver cercato di capire cosa stessi combinando. Sono troppi perché un elenco possa rendere giustizia a tutti: nessuno se ne abbia a male dunque se qui sorvolo, nella speranza di poter ringraziare ognuno come merita.

Ogni singola parola, formula, tabella o figura di questa tesi è dedicata a mia madre e Cecilia, che mi hanno supportato e sopportato più di quanto si potesse desiderare.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Il Machine Learning [44, 24] si propone di ampliare la classe dei problemi trattabili laddove non risultino applicabili approcci algoritmici o analitici. Rilevante è in questo ambito il trattamento di domini strutturati: in molti settori applicativi, infatti, i dati trattati trovano una propria rappresentazione naturale in forma di grafi. Estendere le metodologie di Machine Learning per affrontare problemi definiti su domini strutturati rappresenta dunque un'opportunità per dare risposte efficaci a problemi esistenti, aprendo spazio per lo sviluppo di nuove applicazioni.

Nel realizzare questo obiettivo, quanto verrà discusso nel seguito si orienta in particolare al paradigma neurale, che nell'ambito del Machine Learning ricopre un ruolo rilevante, attingendo a tecniche e modelli esistenti per introdurre soluzioni originali nel campo del trattamento dei domini strutturati. Le Reti Neurali Artificiali [25, 5] sono infatti un insieme vasto e variegato di modelli computazionali, efficacemente applicati a problemi reali. Nate per sfruttare un approccio connessionista di rappresentazione delle informazioni, le Reti Neurali Artificiali offrono importanti caratteristiche computazionali e, nella loro forma più semplice di Reti Neurali Feedforward [5, 25], risultano in

grado di approssimare con precisione arbitraria qualsiasi funzione continua [6].

Benché l'apprendimento in una Rete Neurale Artificiale avvenga modificando i pesi sulle connessioni, nascono nell'ambito delle Reti Neurali Feedforward anche algoritmi e strategie che guardano alla struttura della rete e alla sua realizzazione per migliorarne o semplificarne il processo di creazione e di allenamento. L'approccio costruttivo [52, 9, 36], ad esempio, propone un procedimento incrementale per la costruzione della rete nella suo complesso, secondo un meccanismo guidato direttamente dalle caratteristiche del problema affrontato. In questo caso la rete cresce in maniera iterativa, potendo sfruttare ad ogni passo tutte le informazioni precedentemente apprese ed adattando la propria dimensione alla complessità del problema. Ne risulta il vantaggio di non dover stabilire a priori la topologia della rete, strettamente legata alla sua capacità computazionale, e di poter scomporre il problema in sotto-problemi, singolarmente affrontati attraverso procedimenti di allenamento più semplici e computazionalmente efficienti.

Nonostante le Reti Neurali Feedforward, usate per apprendere funzioni definite su vettori, siano la classe di Reti Neurali Artificiali più ampiamente diffusa nel contesto dei Machine Learning, l'approccio neurale propone anche paradigmi e modelli orientati all'apprendimento su domini più complessi. È il caso delle Reti Neurali Ricorrenti [59, 8], che generalizzano le Reti Neurali Feedforward al trattamento di sequenze attraverso l'introduzione di cicli nella topologia delle connessioni fra le unità. Benché reti di questo tipo risultino efficienti nel contesto del Machine Learning, il costo computazionale del loro allenamento risulta un campo di studio aperto; è in questo contesto infatti che nasce il paradigma del Reservoir Computing [38, 61, 32]: una classe di

modelli caratterizzati dalla separazione concettuale tra una parte ricorrente (i.e. con connessioni cicliche fra le unità) ed una parte non-ricorrente per il calcolo del segnale di uscita. L'opportunità di limitare l'apprendimento alla sola parte non-ricorrente del modello, lasciando inalterata la parte ricorrente, rappresenta il punto di forza del Reservoir Computing, consentendo l'impiego di strategie di allenamento molto efficienti dal punto di vista computazionale. A fronte di questo vantaggio la presenza di una porzione di rete non soggetta ad allenamento determina un limite per i modelli esistenti: una parte delle dinamiche della rete risulta infatti fissata a priori e non ha la capacità di adattarsi al problema affrontato.

L'apprendimento di funzioni definite su domini strutturati (i.e. trasduzioni strutturali), obiettivo della tesi, è affrontato in ambito neurale attraverso le Reti Neurali Ricorsive [12, 54], che generalizzano le Reti Neurali Ricorrenti. Benché in svariati campi si abbiano esempi applicativi di modelli neurali ricorsivi — per citarne solo alcuni, nella Chimica [4, 43, 3], nella Proteomica [1], nel trattamento e riconoscimento di immagini [12] o nell'Ingegneria del Software [12] — il legame fra la topologia dell'input e le dinamiche interne della rete impone, nelle Reti Neurali Ricorsive, dei vincoli sulla classe degli input trattabili: nella loro formulazione originale, reti di questo tipo risultano infatti applicabili unicamente a dati strutturati che presentino un ordinamento topologico fra i vertici, il che limita il dominio di input alla sola classe dei Grafi Diretti Aciclici. Tale assunzione ha impatto sulla capacità dei modelli [42], limitando dunque il numero di problemi affrontabili rispetto a quanto avverrebbe potendo trattare la classe più generale dei grafi.

In risposta a questa esigenza, nella storia recente delle Reti Neurali Ricorsive

si ritrovano vari modelli tesi all'ampliamento della classe di input. Due approcci distinti, in particolare, lasciano emergere elementi interessanti per la realizzazione di modelli neurali in grado di apprendere trasduzioni strutturali definite su grafi.

Il modello *Neural Networks for Graphs* (NN4G) [41] adotta una strategia costruttiva per evitare i problemi determinati dalla presenza di cicli nell'input e per sfruttare localmente informazioni globali, legate alla topologia dei dati: ad ogni iterazione del processo di costruzione della rete, infatti, nuove informazioni relative alla struttura del grafo vengono raccolte e rese disponibili per semplificare l'apprendimento nelle unità aggiunte successivamente.

Un approccio differente è individuato dal modello *Graph Echo State Network* (GraphESN) [15], appartenente all'ambito del Reservoir Computing, che sfrutta le caratteristiche strutturali della rete per realizzare il trattamento di domini strutturati generici. Ne risulta un modello di Rete Neurale Ricorsiva efficiente per l'apprendimento di trasduzioni strutturali definite su grafi, che tuttavia mantiene inalterate alcune delle criticità caratteristiche del Reservoir Computing.

Da quanto detto emergono dunque necessità e limitazioni legate all'apprendimento su domini strutturati attraverso il paradigma neurale. In particolare, in una disciplina fortemente orientata alle applicazioni come il Machine Learning, il costo computazionale e la classe degli input trattabili risultano essere aspetti molto rilevanti. Il paradigma del Reservoir Computing offre strumenti efficaci per rispondere a simili necessità, ma presenta tuttavia degli elementi critici, legati alla presenza di una porzione di rete fissata a priori e mantenuta inalterata durante l'apprendimento. La determinazione della topologia

appropriata e l'opportunità di modificare le dinamiche dell'intera rete sulla base del problema affrontato, sono infatti limiti la cui risoluzione rimane ad oggi un problema aperto [37, 64, 47]. L'approccio costruttivo offre soluzioni in tal senso, proponendo una strategia efficiente per determinare la struttura della rete in maniera automatica e per sfruttare le informazioni raccolte nel corso del processo incrementale di costruzione della rete.
Quanto discusso nel seguito ha dunque l'obiettivo di sintetizzare questi elementi, introducendo nuovi modelli ricorsivi appartenenti all'ambito del Reservoir Computing, che permettano l'adozione di una strategia costruttiva nell'apprendimento di trasduzioni strutturali. In particolare, i modelli descritti nel seguito rappresentano un'estensione delle GraphESN, caratterizzandosi dunque come modelli efficienti per il trattamento di domini strutturati generali. L'introduzione della strategia costruttiva, innovativa nell'ambito del Reservoir Computing, ha impatto sui modelli proposti dal punto di vista computazionale ed algoritmico, riducendo il costo necessario all'allenamento ed evitando il problema di dover fissare a priori la topologia della rete. L'opportunità di sfruttare localmente informazioni globali e supervisionate viene invece usata per modificare le dinamiche di quella porzione di rete che nel Reservoir Computing non è soggetta ad allenamento. Ne risulta un meccanismo stabile per introdurre informazione supervisionata, anche in assenza di una fase di adattamento specifica, che possa influenzare le dinamiche della rete in maniera consistente con il problema affrontato.

Le capacità dei modelli introdotti sono testate, attraverso una procedura rigorosa di validazione, su problemi di Chemioinformatica relativi a dataset reali noti in letteratura. Parte dei risultati riportati nel seguito è inoltre og-

getto dell'articolo *Constructive Reservoir Computation with Output Feedbacks for Structured Domains* [19], che sarà discusso nel corso del XX European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning[1].

L'esposizione degli argomenti nel testo è articolata come segue:

Nel capitolo 1 si introducono i temi, le tecniche ed i modelli alla base del lavoro svolto. Nel corso di questa rassegna dello stato dell'arte vengono in particolare presentati i principi dell'approccio costruttivo e del Reservoir Computing, descrivendo anche modalità ed obiettivi nel trattamento dei domini strutturati.

Nel capitolo 2 vengono introdotti i modelli originali, oggetto del lavoro di tesi, ne vengono presentate le caratteristiche e discussi gli aspetti specifici.

Nel capitolo 3 sono descritti i risultati sperimentali ottenuti testando le capacità dei modelli su una serie di problemi del mondo reale appartenenti all'ambito della Chemioinformatica.

Nel capitolo 4 viene presentata una valutazione del lavoro svolto, con l'obiettivo di determinare vantaggi e criticità dei modelli proposti nonché di individuare di quegli aspetti che potrebbero essere oggetto di un ulteriore, specifico, lavoro di indagine e sperimentazione.

[1] `http://www.dice.ucl.ac.be/esann/`

1 Stato dell'arte

Il capitolo introduce teorie e tecniche alla base del lavoro svolto. È opportuno premettere che quanto verrà descritto nel seguito rappresenta un sottoinsieme eterogeneo all'interno del mondo del Machine Learning, sia dal punto di vista cronologico che da quello modellistico, che però trova una sintesi nei modelli sviluppati. La trattazione procede dunque secondo una suddivisione concettuale che guarda principalmente ai modelli ed ai domini applicativi.

Il paragrafo 1.1 descrive i principi del Machine Learning ed introduce le Reti Neurali Artificiali come modello computazionale, presentando alcune delle tecniche adottate nel loro utilizzo.

Il paragrafo 1.2 introduce l'approccio costruttivo nell'allenamento delle Reti Neurali e descrive l'algoritmo Cascade Correlation per lo sviluppo di reti feedforward costruttive.

Nel paragrafo 1.3 vengono introdotte le Reti Neurali Ricorrenti per il trattamento di sequenze.

Nel paragrafo 1.4 viene presentato il paradigma del Reservoir Computing,

delineandone caratteristiche e motivazioni, e vengono introdotte le Echo State Networks nella loro formulazione classica di Reti Neurali Ricorrenti utilizzate per processare sequenze di input.

Il paragrafo 1.5 descrive il problema dell'apprendimento di dati strutturati, indicando alcune delle soluzioni offerte dal paradigma neurale in questo campo.

1.1 Machine Learning e Reti Neurali

Il Machine Learning [44, 24] è un settore dell'Informatica che propone metodi ed algoritmi per l'apprendimento e la predizione, ovvero mirati ad apprendere da esempi un compito computazionale definito. I modelli di Machine Learning hanno il compito di risolvere problemi del mondo reale difficili da trattare con tecniche tradizionali e si configurano come strumenti complementari rispetto ai modelli analitici basati sulla conoscenza pregressa, agli algoritmi propri della programmazione imperativa o all'Intelligenza Artificiale classica. Obiettivo del Machine Learning è dunque far sì che l'esperienza (collezione di esempi) possa essere appresa in maniera automatica per risolvere un compito, in modo da costruire dei *modelli* (o *ipotesi*) utili per poter fare predizioni. In altri termini, possiamo dire che il Machine Learning studia e propone metodi per inferire dipendenze (o funzioni, o ipotesi), da esempi di dati osservati, che siano in grado di approssimare i dati noti (*fitting*) e che siano capaci di *generalizzare*, ovvero rispondere in modo accurato per dati nuovi.

I modelli di Machine Learning hanno dunque lo scopo di catturare le relazioni tra i dati e definiscono la classe di funzioni che il sistema può calcolare

(*spazio delle ipotesi*). L'apprendimento (*learning*) avviene modificando i parametri liberi di un modello in modo da trovare, all'interno dello spazio delle ipotesi, una funzione che possa correttamente approssimare la *funzione obiettivo* (non nota) in modo da poter essere usata per fare predizioni su nuovi dati di input.

Distinguiamo due tipologie di problemi, o *task*, che i modelli di Machine Learning si propongono di affrontare: task di apprendimento *supervisionato* e di apprendimento *non-supervisionato*.
Nel primo caso la sorgente di esperienza per la funzione obiettivo f_t è rappresentata da un dataset contenente un insieme di coppie, o *esempi etichettati*

$$D = \{(x_i, d_i)\}_{i=1}^{l}$$

dove x è un input e d è un valore di output atteso, o *target*, dato da un *teacher* in accordo ad $f_t(x)$. Nel contesto dell'apprendimento supervisionato è inoltre possibile un'ulteriore suddivisione: si parla di *classificazione* nel caso di funzioni obiettivo a valori discreti, tali che $f_t(x)$ restituisce il valore della classe di appartenenza dell'input x, mentre sono task di *regressione* quelli in cui la funzione obiettivo abbia valori reali, in $\mathbb{R}$ o $\mathbb{R}^n$.
Nel caso dell'apprendimento non-supervisionato la sorgente di esperienza è invece caratterizzata dalla mancanza di valori target, il dataset usato per allenare un modello è quindi composto da soli esempi non etichettati:

$$D = \{(x_i)\}_{i=1}^{l}$$

L'obiettivo è in questo caso quello di trovare *raggruppamenti naturali* in un in-

sieme di dati. L'apprendimento non-supervisionato viene utilizzato tipicamente per fare clustering, riduzione della dimensionalità dei dati, visualizzazione e preprocessing, modellazione della densità dei dati.

Tutti i task ed i modelli che verranno discussi nel corso della tesi si riferiscono a casi di apprendimento *supervisionato*, affrontati in particolare ricorrendo al paradigma neurale, ovvero attraverso modelli di Reti Neurali Artificiali.

1.1.1 Reti Neurali Artificiali

Le Reti Neurali Artificiali [25, 5] sono un vasto insieme di modelli di Machine Learning, sviluppati secondo un'analogia — più o meno marcata — con le strutture e le dinamiche che si ritrovano nel cervello. Reti di questo tipo sono infatti formate da più unità (neuroni), collegate fra loro tramite connessioni pesate (sinapsi). La relazione tra ingresso ed uscita della rete è dunque determinata dalla propagazione del segnale di input sulle connessioni ed attraverso le unità. In analogia con il modello biologico, un singolo neurone artificiale ha un'uscita, o *attivazione*, che dipende dai segnali in ingresso, corrispondenti alle attivazioni di altre unità:

$$o(\mathbf{x}) = f(\sum_j w_j x_j) \tag{1.1}$$

dove x_j indica il j-esimo input, parte del segnale di ingresso $\mathbf{x}$, w_j indica il peso assegnato alla connessione corrispondente ed f è detta *funzione di attivazione*.

Nonostante la semplicità di un singolo neurone artificiale, la composizione di più unità permette la creazione di modelli computazionalmente molto potenti e flessibili: è il caso del Multilayer Perceptron [5, 25], che rappresenta certamente uno dei modelli più utilizzati nell'ambito del Machine Learning. Reti di questo tipo sono realizzate componendo più livelli, o *layer*, di unità collegate secondo una topologia aciclica (i.e. Reti Neurali Feedforward): il segnale viene quindi fatto passare da un layer di input ad uno o più livelli, detti *nascosti*, fino ad un livello di output, le cui attivazioni vengono usate per determinare l'uscita della rete. La funzione, o ipotesi, realizzata da un Multilayer Perceptron con un layer nascosto è

$$h(\mathbf{x}) = f_p(\sum_j w_{pj} f_j(\sum_i w_{ji} x_i)) \tag{1.2}$$

dove f_p ed f_j sono rispettivamente le funzioni di attivazione delle unità di output e delle unità nascoste e w_{rs} indica il peso sulla connessione che va dall'unità s-esima verso l'unità r-esima. La figura 1.1 mostra l'architettura aciclica di un Multilayer Perceptron con un unico layer nascosto.

Come si vede dalle equazioni (1.1) e (1.2), i *parametri liberi* di una Rete Neurale Artificiale sono rappresentati dai pesi, a valori reali, sulle connessioni: lo spazio delle ipotesi è dunque in questo caso continuo, il che permette di ascrivere il paradigma neurale alla classe degli approcci *sub-simbolici* nel campo del Machine Learning. Sono invece detti *iperparametri* di un modello i fattori che ne determinano lo spazio delle ipotesi (e.g. il numero di unità nascoste) o che influenzano l'apprendimento (e.g. i parametri dell'algoritmo di learning utilizzato).

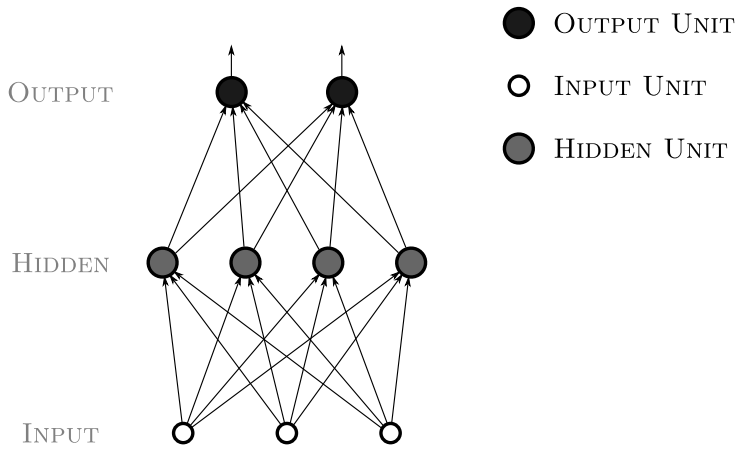


Figura 1.1: Multilayer Perceptron. Architettura di una rete con 3 unità input, 2 unità di output e 4 unità nascoste, in un unico layer.

Modelli di questo tipo si caratterizzano per la capacità di trattare efficacemente dati sia continui che discreti, realizzando task di classificazione o regressione, e mostrando tolleranza rispetto a dati di input rumorosi o incompleti. La presenza di almeno un layer nascosto di unità con funzioni di attivazione non lineari (tipicamente sigmoidali, e.g. tanh) ha inoltre l'effetto di rendere il Multilayer Perceptron un *approssimatore universale* [6], in grado cioè di approssimare con precisione arbitraria qualsiasi funzione continua definita su un sottoinsieme compatto di $\mathbb{R}^n$.

L'apprendimento avviene generalmente, nelle Reti Neurali Feedforward, attraverso l'algoritmo di Backpropagation [25], che sfrutta una discesa del gradiente per modificare iterativamente i pesi di tutte le connessioni, minimizzando l'errore commesso sugli esempi etichettati di training.

1.1.2 Validazione

Come detto in precedenza, lo scopo del processo di apprendimento in un modello di Machine Learning è quello di trovare una buona approssimazione, a partire da dati noti, all'interno di uno spazio di funzioni. Per definire cosa si intenda per *buona* è necessario riferirsi alla capacità di *generalizzazione*, ovvero all'accuratezza nel fare previsioni su dati sconosciuti. La generalizzazione è dunque un punto determinante del Machine Learning e viene esplorata durante una fase di test in cui si esegue la valutazione dell'ipotesi in termini di capacità predittiva.

Per poter valutare la qualità dell'approssimazione di un modello si ricorre a delle funzioni di *loss*

$$L(h(x), d) \tag{1.3}$$

che misurano la discrepanza tra l'output del modello ed il valore desiderato d su un campione x in ingresso (i.e. valori alti della loss indicano scarsa approssimazione). Per i task di apprendimento supervisionato consideriamo le seguenti funzioni di loss:

- Classificazione: *errore di classificazione*

$$L(h(x_i), d_i) = \begin{cases} 0 & \text{se } h(x_i) = d_i \\ 1 & \text{altrimenti} \end{cases} \tag{1.4}$$

- Regressione: *errore quadratico*

$$L(h(x_i), d_i) = (d_i - h(x_i))^2 \tag{1.5}$$

Il legame tra funzione di loss e generalizzazione è determinato dall'*ipotesi dell'apprendimento induttivo*, secondo cui ogni ipotesi h che approssima bene f_t sugli esempi di training approssimerà bene f_t anche su (nuove) istanze sconosciute di input.

Tale assunzione è fondante dell'approccio che si adotta nella realizzazione dell'apprendimento. In accordo con tale ipotesi, per determinare l'approssimazione migliore di f_t, si ricerca quindi un'ipotesi h che minimizzi l'*errore reale* R

$$R = \int L(d, h(x))\, dP(x, d) \tag{1.6}$$

dove d è un valore dato dal teacher, $P(x, d)$ è la distribuzione dei dati — che generalmente non è nota a priori — e $L(d, h(x))$ è una funzione di loss. Disponendo solo di un insieme finito di dati di training

$$D = \{(x_i, d_i)\}_{i=1}^{l} \tag{1.7}$$

per cercare h si ricorre alla minimizzazione del *rischio empirico* (errore di training), alla ricerca dei migliori valori per i parametri liberi del modello

$$R_{emp} = \frac{1}{l} \sum_{i} L(d_i, h(x_i)) \tag{1.8}$$

seguendo il principio induttivo della *Minimizzazione del Rischio Empirico*.

Esiste tuttavia la possibilità che l'ipotesi dell'apprendimento induttivo non sia fondata, in particolare nel caso di *overfitting*. Considerando un'ipotesi generica $h_i \in H$ e chiamando ϵ_i il suo errore reale e E_i il suo errore empirico, diciamo che un learner fa *overfitting* se l'ipotesi in output h_{out} è tale che esiste

una ipotesi diversa $h_j \in H$ per cui: $E_{out} < E_j$ e $\epsilon_{out} > \epsilon_j$. Intuitivamente, quindi, l'overfitting corrisponde alla situazione in cui un modello sia stato allenato fino a raggiungere un livello di approssimazione dei dati di training eccessivamente alto, perdendo di conseguenza la capacità di generalizzare.

In modo particolare nell'uso di Reti Neurali, caratterizzate da un'estrema flessibilità nell'approssimare la funzione obiettivo, si rende dunque necessario un procedimento che permetta di valutare con accuratezza la performance di un modello. Nel seguito vengono brevemente descritte alcune delle tecniche comunemente usate a questo scopo (si veda anche [24]).

Hold-out

Questa tecnica può essere utilizzata qualora si disponga di dataset di grosse dimensioni, e rappresenta il caso più semplice di validazione di un modello.

Il dataset D viene partizionato in due insiemi *disgiunti*: un *training-set* D_{tr} ed un *test-set* $D_{\text{ts}} = D \setminus D_{\text{tr}}$. Si ha dunque

$$D = D_{\text{tr}} \cup D_{\text{ts}}$$

Il modello viene allenato sul training-set e, successivamente, il test-set viene utilizzato per valutarne la capacità di generalizzazione.

Per garantire che i risultati possano essere rappresentativi della capacità predittiva del modello, il test-set non viene usato per l'adattamento dei parametri liberi né per la selezione del modello (i.e. scelta degli iperparametri). A questo scopo, se si hanno maggiori dati a disposizione, è possibile partizionare

ulteriormente il training-set, ottenendo un *validation-set* D_{val} che può essere usato per scegliere l'ipotesi migliore. In questo caso, quindi si ha

$$D = D_{\text{tr}} \cup D_{\text{val}} \cup D_{\text{ts}}$$

con i dati in D_{tr} usati per allenare più modelli, D_{val} impiegato per scegliere, fra i vari modelli allenati, quello considerato più adatto al problema affrontato ed, infine, D_{ts} usato per determinare la performance del modello scelto. Il processo di scelta dell'ipotesi migliore attraverso un validation-set è chiamato *model selection* ed avviene dunque secondo un procedimento di *trial and error*.

K-fold cross-validation

Nel caso in cui si abbiano pochi dati a disposizione, l'hold-out può comportare una riduzione eccessiva di dati utili per il training. In questo caso si procede con la variante della *k-fold cross-validation*, che prevede il partizionamento del dataset D in k sottoinsiemi, o *fold*, mutuamente esclusivi

$$D = D_1 \cup D_2 \cup \cdots \cup D_k$$

Il modello viene dunque allenato e testato per k volte, facendo in modo che la porzione di dati usati per il test non sia usata nel training del modello

$$D_{\text{tr}} = D \setminus D_i \quad \text{e} \quad D_{\text{ts}} = D_i \qquad \text{per} \quad i = 1, \ldots, k$$

La performance è infine ottenuta come media delle performance ottenute sul test-set delle varie fold.

L'impiego di k-fold cross-validation consente dunque di usare tutti i dati a disposizione sia per il training che per il test e può essere combinato, come nel caso dell'hold-out, con l'uso di un validation-set per la selezione del modello.

Double k-fold cross-validation

In questo caso vengono realizzati due cicli di cross-validation, di cui quello "interno" viene utilizzato per la selezione del modello.

Come per la k-fold cross-validation, il dataset viene inizialmente partizionato in fold

$$D = D_1 \cup D_2 \cup \cdots \cup D_k$$

e successivamente, ogni fold viene a sua volta suddivisa in t sotto-fold

$$D_i = D_{i1} \cup D_{i2} \cup \cdots \cup D_{it} \qquad \text{per} \quad i = 1, \ldots, k$$

Ogni sotto-fold viene dunque utilizzata per allenare e valutare più iperparametrizzazioni

$$D_{\text{tr}} = D_i \setminus D_{ij} \quad \text{e} \quad D_{\text{val}} = D_{ij} \qquad \text{per} \quad j = 1, \ldots, t$$

Tale processo permette di ottenere k modelli "vincenti", uno per ogni fold, selezionati in base alla performance media sui validation-set. Il test viene dunque eseguito, su ogni fold, valutando il modello vincente attraverso una

k-fold cross-validation

$$D_{\text{tr}} = D \setminus D_i \quad \text{e} \quad D_{\text{ts}} = D_i \qquad \text{per} \quad i = 1, \ldots, k$$

La performance del modello è quindi calcolata come la media delle performance raggiunte dalle iperparametrizzazioni vincenti su ogni fold.

La double k-fold cross-validation è un processo accurato, ma computazionalmente dispendioso, che consente di ottenere una stima dell'accuratezza di un modello anche su dataset di dimensioni ridotte. Si distingue tuttavia dalle tecniche precedenti per il fatto di restituire non una, ma k, iperparametrizzazioni vincenti.

Stratificazione

Nel partizionare un dataset secondo le tecniche descritte è ovviamente necessario che tutti i sottoinsiemi siano sufficientemente rappresentativi per il problema trattato e che riescano dunque a descrivere le relazioni fra i dati. Eventuali sbilanciamenti nella natura dei dati in uno dei sottoinsiemi possono infatti comportare un *bias* che può avere impatto negativo sulla capacità di generalizzazione dei modelli.

La *stratificazione* è una tecnica mirata ad evitare questo scenario e prevede che i dati vengano partizionati, prima del sampling, in gruppi omogenei, in modo che i sottoinsiemi possano essere formati mantenendo lo stesso rapporto fra i gruppi presente nell'intero dataset.

1.1.3 Algoritmi di apprendimento

In questo paragrafo sono riportati alcuni degli algoritmi di apprendimento che verranno riferiti nel seguito e che sono stati utilizzati nel corso del lavoro svolto. Gli algoritmi coprono un caso specifico dell'apprendimento nell'ambito neurale: l'allenamento di singole unità che operano come il neurone artificiale descritto nell'equazione (1.1).

Quanto verrà descritto non si rivolge dunque all'adattamento dei pesi di una rete multistrato (si veda il paragrafo 1.1.1), che richiede tecniche specifiche, quanto l'allenamento di un singolo livello di output. Per questo motivo, gli algoritmi riportati si caratterizzano per la loro semplicità ed efficienza computazionale.

Least Mean Squares

Consideriamo un'unità con una funzione di attivazione non lineare e differenziabile (e.g. tanh) che abbia $\mathbb{N}_U$ ingressi. Possiamo scrivere l'uscita per l'input $\mathbf{x} \in \mathbb{R}^{N_U}$ come

$$o(\mathbf{x}) = f(\mathbf{x}^{\mathrm{T}}\mathbf{w}) \tag{1.9}$$

dove $\mathbf{w} \in \mathbb{R}^{N_U}$ rappresenta il vettore dei pesi associati ai singoli input.
Siamo interessati a determinare un valore appropriato per $\mathbf{w}$, che minimizzi la somma degli errori al quadrato

$$E(\mathbf{w}) = \sum_p (d_p - o(\mathbf{x}_p))^2 = \sum_p (d_p - f(\mathbf{x}_p^{\mathrm{T}}\mathbf{w}))^2 \tag{1.10}$$

Utilizzando l'algoritmo Least Mean Squares (LMS) [25] la minimizzazione procede attraverso un *discesa del gradiente*

$$-\Delta\mathbf{w} = \frac{\partial E(\mathbf{w})}{\partial w_j} = -2\sum_{p\to l}(\mathbf{x}_p)_j(d_p - f(\mathbf{x}_p^{\mathrm{T}}\mathbf{w}))f'(\mathbf{x}_p^{\mathrm{T}}\mathbf{w}) \tag{1.11}$$

Iterativamente, i pesi $\mathbf{w}$ vengono dunque modificati come

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta\,\Delta\mathbf{w}_{old} \tag{1.12}$$

dove η è il *learning-rate*, iperparametro dell'algoritmo. Ad ogni iterazione, il costo dell'algoritmo è lineare rispetto alla dimensione dell'input N_U.

Il procedimento si può inoltre estendere, aggiungendo alla funzione di errore un fattore di penalizzazione che impedisca ai pesi di assumere valori troppo alti, in accordo con il principio della *minimizzazione del rischio strutturale* [60]. L'aumento del valore dei pesi corrisponde infatti ad un aumento della complessità del modello, che favorisce il verificarsi di situazioni di overfitting. In questo caso l'aggiornamento dei pesi viene modificato in

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta\,\Delta\mathbf{w}_{old} - \lambda_{\mathrm{wd}}\,\mathbf{w}_{old} \tag{1.13}$$

dove λ_{wd} è un parametro corrispondente al cosiddetto *weight decay*.
La riduzione della complessità del modello viene chiamata *regolarizzazione*.

Ridge Regression

Consideriamo un insieme di $|D|$ vettori di input rappresentati in forma matriciale, $\mathbf{X} \in \mathbb{R}^{|D|\times N_U}$, ed un insieme di output desiderati, o valori target,

$\mathbf{Y}_{\text{target}} \in \mathbb{R}^{|D| \times N_Y}$. L'algoritmo di Ridge Regression [24] permette di calcolare i pesi di N_Y unità con funzione di attivazione lineare, in modo da minimizzare l'errore (1.10) e tenendo conto di un fattore di penalizzazione, come nel caso di LMS con weight decay.
La matrice dei pesi, $\mathbf{W} \in \mathbb{R}^{N_U \times N_Y}$, è in questo caso calcolata come

$$\mathbf{W} = (\mathbf{X}^T\mathbf{X} + \lambda_{\text{r}}\mathbf{I})^{-1}\mathbf{X}^T\mathbf{Y}_{\text{target}} \tag{1.14}$$

dove $\mathbf{I} \in \mathbb{R}^{N_U \times N_U}$ è la matrice identità e λ_{r} è un parametro di regolarizzazione.

A differenza di LMS, l'algoritmo di Ridge Regression permette di ottenere la soluzione in maniera diretta, senza ricorrere ad un procedimento iterativo. Il costo computazionale dell'algoritmo è cubico rispetto alla dimensione dell'input N_U.

Benché sia definito su unità lineari, è possibile applicare l'algoritmo al caso di unità con funzioni di attivazioni non lineari semplicemente modificando i valori target, applicando f^{-1} elemento per elemento: $\mathbf{Y}'_{\text{target}} = f^{-1}(\mathbf{Y}_{\text{target}})$. In questo caso è ovviamente necessario che la funzione di attivazione sia invertibile e che i valori target rientrino nel suo codominio.

1.2 Reti Neurali Costruttive

La maggior parte degli algoritmi per Reti Neurali Artificiali prevede l'uso di modelli con architetture statiche. Più esattamente, reti con una topologia prefissata vengono create e successivamente allenate: i pesi sulle connessioni vengono fatti variare, ma non l'architettura in sé.

Uno degli evidenti svantaggi di un simile approccio sta nella necessità di dover evitare i casi in cui la rete risulti essere, per la propria struttura e quindi al netto delle modifiche ai pesi, troppo o troppo poco complessa per il task che si vuole affrontare.

Le *Reti Neurali Costruttive* [52, 45] offrono una valida soluzione al problema di dover stabilire a priori la topologia della rete affinché possa ben adattarsi al task che si vuole risolvere. L'allenamento di una rete neurale costruttiva inizia tipicamente con una rete di piccole dimensioni — anche senza alcuna unità nascosta, nel caso delle reti feedforward — e procede aggiungendo nuove unità alla rete finché questa non abbia raggiunto una complessità compatibile con il problema in questione.

Al di là dei dettagli implementativi possiamo dunque individuare le due caratteristiche alla base dell'approccio costruttivo.

- Una rete viene vista come formata da più *unità computazionali* con una capacità limitata, che vengono allenate per risolvere un sotto-problema rispetto al task affrontato.

- La rete *aumenta la propria complessità* nel corso del training, adattandosi al task che le viene sottoposto.

È importante sottolineare come questi due principi, benché sviluppati nell'ambito delle Reti Feedforward, possano essere applicabili ad un'ampia classe di problemi o modelli.

I principali vantaggi offerti dall'adozione di un approccio costruttivo sono:

- Il superamento della necessità di dover fissare a priori la topologia della rete, che diventa dunque adattiva e viene dinamicamente "appresa" dalla rete stessa.

- La possibilità di definire per le sotto-reti dei task specifici, che possano essere trattati in maniera più efficace o più efficiente rispetto al problema affrontato.

- L'adozione di una politica locale nell'aggiornamento dei pesi, con il duplice vantaggio di evitare i problemi legati all'adattamento dei pesi dell'intera rete (e.g. *vanish del gradiente*) e di consentire l'implementazione di meccanismi di caching/memoization per le porzioni di rete non direttamente interessate dal learning locale.

- La possibilità di circoscrivere il learning ad unità computazionali, o sotto-reti, più semplici della rete nel suo complesso. Questo consente in particolare l'impiego di algoritmi di apprendimento specifici e meno onerosi dal punto di vista computazionale rispetto a quelli necessari ad allenare l'intera rete.

A fronte dei vantaggi offerti, le Reti Neurali Costruttive presentano tuttavia alcune criticità legate alla capacità della rete di crescere. Poiché generalmente ogni nuova unità è connessa alle precedenti, infatti, le reti di grandi dimensioni tendono ad avere molti layer ed unità con un un numero di connessioni in input molto elevato: questo può avere un grosso impatto sul processo di learning e compromettere la scalabilità del modello. Il fatto che la rete possa aumentare il proprio numero di unità indefinitamente, guidata unicamente

dall'input, espone inoltre i modelli costruttivi al verificarsi di situazioni di overfitting (si veda il paragrafo 1.1.2).

1.2.1 Cascade Correlation

L'algoritmo *Cascade Correlation* [9, 36, 45] rappresenta probabilmente uno dei più diffusi casi di applicazione dell'approccio costruttivo nell'allenamento di Reti Neurali Feedforward. L'intuizione alla base dell'algoritmo sta nell'idea di allenare nuove unità computazionali perché possano (i) contribuire alla risoluzione del task affrontato risolvendo dei sotto-problemi di natura diversa e semplificata e (ii) avvalersi delle informazioni precedentemente apprese dalla rete nel corso del processo costruttivo. In particolare ad ogni nuova unità viene affidato il compito di *massimizzare la correlazione* fra il proprio output e l'errore commesso dalla rete, con lo scopo di correggerlo.

L'evoluzione dell'algoritmo è la seguente. Inizialmente la rete non ha unità nascoste; le connessioni input–output vengono dunque allenate sul training-set attraverso un algoritmo di apprendimento adatto a reti con un singolo strato (e.g. *delta-rule* o algoritmo di apprendimento del Perceptron) e senza necessità di utilizzare Backpropagation.

Dopo aver effettuato l'allenamento viene calcolato l'errore commesso dalla rete: se si è soddisfatti della performance raggiunta, in termini di fitting, allora l'algoritmo termina, altrimenti si procede nel tentativo di ridurre l'errore.

Per ridurre l'errore, una nuova unità nascosta, chiamata *candidata*, viene aggiunta alla rete: collegata sia all'input che ad ogni altra unità nascosta esistente, viene allenata perché il suo output abbia correlazione massima con

l'errore residuo commesso dalla rete. La correlazione (non normalizzata) S fra l'uscita della rete V e l'errore residuo E_o osservato all'unità di output o-esima è definita come

$$S = \sum_o \left| \sum_p (V_p - \bar{V})(E_{p,o} - \bar{E}_o) \right| \tag{1.15}$$

dove p indica i pattern in input e $\bar{V}$ ed $\bar{E}_o$ sono i valori medi, dell'uscita e dell'errore rispettivamente, calcolati su tutti i pattern del training-set.
L'allenamento della candidata avviene attraverso una *ascesa del gradiente* che sfrutta la derivata parziale di S rispetto al generico peso w_i

$$\frac{\partial S}{\partial w_i} = \sum_{p,o} \sigma_o \left(E_{p,o} - \bar{E}_o\right) f'_p \, I_{i,p} \tag{1.16}$$

dove σ_o è il segno della correlazione fra l'output della candidata e l'output o, f'_p è la derivata della funzione di attivazione della candidata (e.g. tangente iperbolica) applicata ai suoi input per il pattern p-esimo ed $I_{i,p}$ è l'i-esimo input che la candidata riceve per il pattern p.
In questa fase è inoltre possibile ricorrere all'uso di un *pool* di unità candidate, ognuna con pesi iniziali random, in modo da variare le condizioni iniziali dell'algoritmo di apprendimento e scegliere poi l'unità che abbia raggiunto il massimo valore di S.

Ad apprendimento ultimato la nuova unità viene stabilmente aggiunta alla rete: i pesi sulle sue connessioni in ingresso vengono "congelati", di modo che rimarranno inalterati per il resto del processo di costruzione della rete, ed il suo output viene collegato allo strato di output della rete. Il "congelamento"

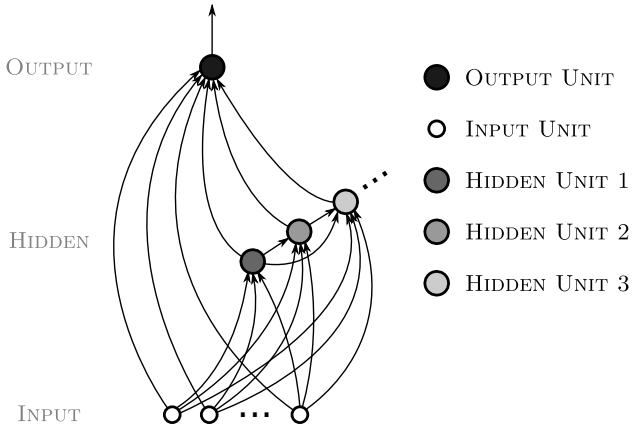


Figura 1.2: Cascade Correlation. Architettura di una rete con 1 unità di output e 3 unità nascoste.

dei pesi sulle connessioni in input alla nuova unità ha risvolti importanti sulle caratteristiche dell'algoritmo. Una volta allenata, infatti, l'unità candidata agirà permanentemente come un *feature detector* e la sua uscita potrà essere presentata allo strato di output o ad altre unità nascoste esattamente come un input aggiuntivo. Benché la rete evolva secondo una architettura a più layer (i.e. ogni unità rappresenta un layer a sé stante, figura 1.2), dunque, gli input e le uscite delle unità nascoste possono essere considerati come appartenenti ad un unico strato ai fini dell'allenamento, il che permette l'uso di algoritmi di apprendimento semplici e poco onerosi dal punto di vista computazionale.

Aggiunta una nuova unità nascosta si procede nuovamente con l'allenamento delle connessioni verso l'output della rete, con la valutazione dell'errore commesso ed eventualmente con l'aggiunta di nuove unità, finché il fitting non sia ritenuto soddisfacente. L'algoritmo evolve dunque in maniera *greedy*, aggiungendo unità nascoste finché non siano rispettati determinati criteri di stop.

Per chiarire meglio in che modo evolve la topologia di una rete allenata con Cascade Correlation, la figura 1.2 nella pagina precedente mostra l'architettura di una rete con 3 unità nascoste ed un'unica unità di output.

1.3 Reti Neurali Ricorrenti

Le Reti Neurali Ricorrenti [59, 25] generalizzano le Reti Neurali Feedforward (si veda il paragrafo 1.1) al trattamento di sequenze. Ciò che distingue le due classi di modelli risiede nel fatto che le Reti Neurali Ricorrenti presentano cicli nella topologia delle connessioni. Questa semplice caratteristica ha un profondo impatto sulle proprietà della rete, in particolare:

- Per effetto dei cicli nella struttura delle connessioni, una Rete Neurale Ricorrente può sviluppare e mantenere delle dinamiche interne anche in assenza di input. Reti di questo tipo realizzano infatti un *sistema dinamico*, mentre le Reti Neurali Feedforward realizzano *funzioni*.

- Se guidate da un segnale di ingresso, le Reti Neurali Ricorrenti mantengono nel proprio stato interno (i.e. nelle attivazioni delle proprie unità) una trasformazione non lineare del passato dell'input. La rete possiede dunque una *memoria dinamica* che le permette di elaborare informazioni in un contesto temporale.

Grazie all'introduzione di cicli nella topologia, dunque, la rete può essere usata per elaborare dati sequenziali realizzando una *trasduzione di sequenza*, ovvero una funzione da un dominio di sequenze di input ad un dominio di sequenze di output.

Nella sua forma più semplice una Rete Neurale Ricorrente è composta da un layer di ingresso, un successivo layer nascosto ricorrente (i.e. con connessioni cicliche fra le unità) ed un ultimo layer feedforward (i.e. con connessioni acicliche) di uscita: il layer nascosto calcola una *funzione locale di encoding* dell'input, mentre il layer di output calcola una *funzione di output*. Per realizzare l'encoding, in una rete che abbia N_U unità di input ed N_R unità nascoste, il layer nascosto calcola la *funzione di transizione di stato* ricorrente $\tau : \mathbb{R}^{N_U} \times \mathbb{R}^{N_R} \to \mathbb{R}^{N_R}$ come

$$\mathbf{x}(n) = \tau(\mathbf{u}(n), \mathbf{x}(n-1)) = f(\mathbf{W}_{\text{in}}\mathbf{u}(n) + \hat{\mathbf{W}}\mathbf{x}(n-1)) \tag{1.17}$$

dove f è la funzione di attivazione tipicamente sigmoidale delle unità nascoste, $\mathbf{u}(n)$ rappresenta l'input n-esimo della sequenza, $\mathbf{x}(t)$ è lo stato interno della rete (i.e. le attivazioni delle unità nascoste), la matrice $\mathbf{W}_{\text{in}} \in \mathbb{R}^{N_R \times (N_U+1)}$ contiene i pesi sulle connessioni dal livello di input al livello nascosto (compreso un *bias*) e la matrice $\hat{\mathbf{W}} \in \mathbb{R}^{N_R \times N_R}$ contiene i pesi sulle connessioni ricorrenti fra le unità nascoste.
Per una rete che abbia N_Y unità di uscita, il layer di output calcola invece la funzione di output $g_{\text{out}} : \mathbb{R}^{N_R} \to \mathbb{R}^{N_Y}$ come

$$\mathbf{y}(n) = g_{\text{out}}(\mathbf{x}(n)) = f_{\text{out}}(\mathbf{W}_{\text{out}}\mathbf{x}(n)) \tag{1.18}$$

dove $\mathbf{y}(n)$ è l'uscita della rete al passo n-esimo, f_{out} è la funzione di attivazione delle unità di output e $\mathbf{W}_{\text{out}} \in \mathbb{R}^{N_Y \times (N_R+1)}$ contiene i pesi delle connessioni fra il layer nascosto ed il layer di output (più il *bias*).

L'allenamento delle Reti Neurali Ricorrenti usa generalmente tecniche di discesa del gradiente in cui tutte le connessioni della rete vengono modificate per minimizzare l'errore. Gli algoritmi di apprendimento più comuni sono *Back Propagation Through Time* (BPTT) [62] e *Real-Time Recurrent Learning* (RTRL) [63], che estendono al caso ricorrente le tecniche applicate per l'allenamento di Reti Neurali Feedforward, sfruttando la costruzione di una rete multistrato con topologia aciclica, ottenuta "copiando" il layer nascosto, ricorrente, della rete lungo ogni passo della sequenza temporale di input (*unfolding*) [49].

1.4 Reservoir Computing

Il *Reservoir Computing* [38, 61, 32] è un paradigma emergente, nell'ambito delle *Reti Neurali Ricorrenti*, che ha origine da due classi di modelli: *Echo State Networks* (ESN) [29] e *Liquid State Machines* (LSM) [39]. Benché entrambi i modelli condividano — e contribuiscano a definire — i tratti distintivi del Resevoir Computing, nel seguito si farà riferimento in maniera specifica alle ESN, maggiormente legate ad un approccio computazionale[1] e vere progenitrici dei modelli oggetto del lavoro svolto.
Prima di descrivere il funzionamento di una ESN, è utile delineare motivazioni, intuizioni e metodi che caratterizzano il Reservoir Computing come paradigma a sé stante per il trattamento di dati strutturati[2].

[1] Le LSM nascono infatti nell'ambito delle neuroscienze e mantengono caratteristiche di forte ispirazione biologica.

[2] Il termine *dati strutturati* è in questo contesto volutamente generico. Benché in questo paragrafo ci si riferisca a dati in forma di sequenze, dominio proprio delle Reti Neurali Ricorrenti e quindi delle ESN, è vero infatti che le caratteristiche del paradigma possono essere riportate a domini strutturati anche più complessi, come i grafi (si veda il

Riprendendo quanto descritto nel paragrafo 1.3, una generica Rete Neurale Ricorrente viene usata per elaborare dati sequenziali apprendendo una *trasduzione di sequenza*. Tale compito è realizzato modellando un sistema dinamico in cui lo stato interno e l'uscita sono determinati come definito nelle equazioni (1.17) e (1.18) rispettivamente. Osservando le formule risulta evidente come l'encoding e l'output giochino un ruolo differente: la *funzione di transizione di stato* τ implementa infatti un processo di codifica ricorsivo della sequenza in input, che sfrutta informazioni sul contesto ed ha quindi una propria memoria, mentre $\mathbf{y}(n)$ è il risultato di una funzione pura, senza memoria, della codifica in $\mathbf{x}(n)$.

Secondo l'approccio classico, l'allenamento delle Reti Neurali Ricorrenti usa tecniche di discesa del gradiente che prevedono l'adattamento di tutti i pesi della rete: nessuna suddivisione concettuale fra stato interno ed output viene realizzata benché le differenze emergano dal punto di vista algoritmico (l'output, a differenza dello stato interno, è infatti direttamente confrontabile con il target). Gli algoritmi di apprendimento più comuni, *Back Propagation Through Time* (BPTT) [62] e *Real-Time Recurrent Learning* (RTRL) [63], soffrono tuttavia di alcuni svantaggi. In particolare:

- L'aggiornamento graduale dei parametri può modificare le dinamiche della rete fino a far degenerare le informazioni del gradiente, di modo che la convergenza possa non essere garantita [7].

- Il costo computazionale per l'aggiornamento è molto alto — $O(N^2)$ per BPTT e $O(N^4)$ per RTRL, in una rete con N unità — e riduce la

paragrafo 1.5.2 a pagina 36).

possibilità di utilizzare reti di grandi dimensioni.

- Il gradiente decresce esponenzialmente nel tempo, per cui risulta difficile apprendere dipendenze a lungo termine [2].

Il paradigma del Reservoir Computing nasce dunque con l'idea di evitare almeno parzialmente questi problemi adottando un approccio radicalmente differente:

- Una rete neurale ricorrente, chiamata *reservoir*, viene generata *in maniera casuale* ed ha lo scopo di espandere il segnale in input, mantenendo nel proprio stato interno una trasformazione non lineare del passato. I pesi del reservoir rimangono *inalterati* durante il training.

- L'output viene generato come una *combinazione lineare* dei segnali provenienti dalle unità del reservoir, passivamente eccitate dall'input. Il *readout* lineare viene adattato, in fase di learning, utilizzando il segnale del teacher come target.

La suddivisione fra stato interno della rete ed output è quindi in questo caso resa esplicita e si sfrutta la capacità del reservoir di mantenere una trasformazione non lineare del passato, anche senza dover effettuare l'apprendimento [58], per limitare l'azione del learning al solo readout.

Benché il Reservoir Computing si caratterizzi come un paradigma a sé, è giusto osservare che l'idea di gestire in maniera specifica e separata readout e stato interno trova altri esempi nell'ambito del Machine Learning: è il caso, ad esempio, dell'algoritmo di apprendimento *Backpropagation-Decorrelation*[3]

[3]Per le sue caratteristiche, l'algoritmo BPDC viene talvolta indicato a tutti gli effetti come appartenente all'ambito del Reservoir Computing. Si veda, ad esempio, [53].

(BPDC) [56] o delle *Extreme Learning Machines* (ELM) [28]. In termini ancor più generali, l'approccio del Reservoir Computing è inoltre riconducibile all'utilizzo di un kernel, come sottolineato anche in [33].

1.4.1 Echo State Networks

Definiamo formalmente le ESN [29, 30, 32], fissando anche parte della terminologia che verrà adottata nel seguito.

Consideriamo una rete con N_U unità di input, N_R unità ricorrenti interne (reservoir), ed N_Y unità di output. Indichiamo con $\mathbf{u}(n) \in \mathbb{R}^{N_U}$ l'input all'istante n, appartenente alla sequenza $s(\mathbf{u}) = [\mathbf{u}(1), \mathbf{u}(2), \ldots, \mathbf{u}(k)]$, con $\mathbf{x}(n) \in \mathbb{R}^{N_R}$ le attivazioni delle unità del reservoir e con $\mathbf{y}(n) \in \mathbb{R}^{N_Y}$ l'output. Usiamo le seguenti matrici per i pesi delle connessioni: $\mathbf{W}_{\text{in}} \in \mathbb{R}^{N_R \times (N_U+1)}$ per l'input, $\mathbf{W} \in \mathbb{R}^{N_R \times N_R}$ per le connessioni interne e $\mathbf{W}_{\text{out}} \in \mathbb{R}^{N_Y \times (N_R+1)}$ per le connessioni verso le unità di output, ovvero verso il readout.
Le equazioni di base che determinano l'evoluzione del sistema sono le seguenti:

$$\begin{aligned} \mathbf{x}(n) &= \tau(\mathbf{u}(n), \mathbf{x}(n-1)) = f(\mathbf{W}_{\text{in}}\mathbf{u}(n) + \hat{\mathbf{W}}\mathbf{x}(n-1)) \\ \mathbf{y}(n) &= g_{\text{out}}(\mathbf{x}(n)) = f_{\text{out}}(\mathbf{W}_{\text{out}}\mathbf{x}(n)) \end{aligned} \tag{1.19}$$

dove f ed f_{out} sono funzioni applicate elemento per elemento e corrispondono rispettivamente alle funzioni di attivazione delle unità del reservoir, tipicamente sigmoidali (e.g. tanh), e delle unità di output, tipicamente lineari. La figura 1.3 mostra in maniera schematica la struttura di una ESN come quella descritta dall'equazione (1.19).

Varianti comuni all'equazione (1.19) prevedono inoltre l'esistenza di con-

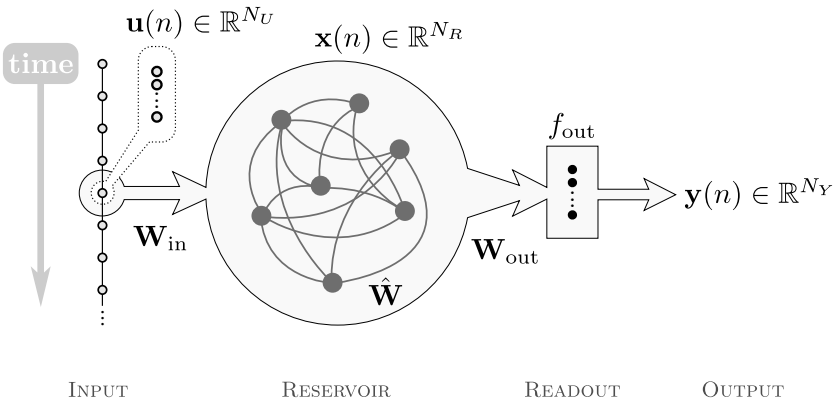


Figura 1.3: Schematizzazione grafica di una Echo State Network (ESN).

nessioni dirette dall'input al readout o all'indietro, dal readout al reservoir (si veda ad esempio [29, 30]). La presenza di queste ultime in particolare implica considerazioni specifiche e risulta interessante ai fini del lavoro svolto. L'introduzione di connessioni che portino informazione all'interno del reservoir risponde infatti ad un limite specifico del Reservoir Computing: la presenza di dinamiche fissate a priori, che non vengono adattate sulla base del task affrontato. Con l'introduzione di connessioni all'indietro, dal readout verso il reservoir, si cerca dunque di realizzare un meccanismo, che chiameremo di *output-feedback*, che permetta di influenzare le dinamiche del reservoir in maniera consistente con il problema che si vuole risolvere. Una soluzione di questo tipo presenta tuttavia dei problemi. In particolare:

- La presenza di di connessioni all'indietro, dal readout al reservoir, tende a determinare situazioni di instabilità nel modello [37, 64], e risulta dunque di difficile applicazione.

- Il sistema di output-feedback proposto per le ESN non risulta direttamente generalizzabile al caso in cui si trattino domini di input più

complessi delle sequenze (si veda il paragrafo 1.5.2 a pagina 36).

- La presenza di output-feedback, può avere l'effetto di influenzare le dinamiche del reservoir, ma non quello di introdurvi effettivamente informazione supervisionata. Al passo n, infatti, il segnale di output-feedback introduce nel reservoir un segnale corrispondente all'uscita della rete al passo $n-1$, che non ha legami con l'uscita desiderata al passo n-esimo.

Per questi motivi, l'introduzione di un meccanismo stabile di output-feedback è ad oggi un tema aperto nell'ambito del Reservoir Computing.

L'allenamento di una ESN avviene, in maniera supervisionata, sulla base dei valori target $\mathbf{y}_{\text{target}}(n)$. A differenza di quanto accade nell'approccio classico, solo $\mathbf{W}_{\text{out}}$ è interessata dall'aggiornamento dei pesi in fase di learning, mentre le altre matrici dei pesi rimangono invariate. Proprio per questa caratteristica è tuttavia necessario che le matrici $\mathbf{W}_{\text{in}}$ e $\hat{\mathbf{W}}$ soddisfino determinati requisiti: informalmente possiamo dire che è necessario che lo stato interno della rete sia un'*eco* della sequenza in input.

In [29, 30] viene definita la *echo state property*, che descrive la condizione basilare per il corretto funzionamento del reservoir. Intuitivamente la echo state property implica che l'effetto dello stato $\mathbf{x}(n)$ e dell'input $\mathbf{u}(n)$ sugli stati futuri, $\mathbf{x}(n+t)$, svanisca con il passare del tempo ($t \to \infty$), senza persistere né essere amplificato. Di conseguenza le attivazioni della rete, dopo una fase transitoria, dipenderanno unicamente dalla sequenza in input e non dallo stato iniziale, che può dunque essere arbitrario.

Assumendo che una rete abbia funzioni di attivazione sigmoidali è possibile individuare una condizione sufficiente per il verificarsi di una simile situazione ed una condizione sufficiente perché invece non si verifichi. Sia la matrice $\hat{\mathbf{W}}$ tale che

$$\sigma_{max} < 1 \tag{1.20}$$

con σ_{max} massimo valore singolare, allora la rete possiede la echo state property per ogni input ammissibile $s(\mathbf{u})$.
Sia la matrice $\hat{\mathbf{W}}$ tale da avere raggio spettrale

$$\rho(\hat{\mathbf{W}}) = |\lambda_{max}| > 1 \tag{1.21}$$

dove λ_{max} è l'autovalore di modulo massimo, allora la rete non ha la echo state property se la sequenza nulla è un input ammissibile.

In [17] la presenza della echo state property viene messa in relazione con la *contrattività* della funzione di transizione di stato τ. In particolare si dimostra che se τ è contrattiva con parametro $C < 1$, ovvero

$$\exists C \in [0,1) \text{ tale che } \forall \mathbf{u} \in \mathbb{R}^{N_U}, \forall \mathbf{x}, \mathbf{x}' \in \mathbb{R}^{N_R} :$$
$$\|\tau(\mathbf{u}, \mathbf{x}) - \tau(\mathbf{u}, \mathbf{x}')\| \leq C\|\mathbf{x} - \mathbf{x}'\| \tag{1.22}$$

per una qualsiasi norma $\|\cdot\|$ nello spazio degli stati $\mathbb{R}^{N_R}$, allora la echo state property è garantita. Oltre ad assicurare la stabilità della rete, la contrattività della funzione di transizione di stato del reservoir determina dei vincoli sull'evoluzione degli stati della rete. Lo spazio degli stati del reservoir risulta infatti avere una *natura Markoviana*: gli stati corrispondenti

a due diverse sequenze di input che abbiano un suffisso comune saranno tanto più vicini quanto maggiore sarà la lunghezza del suffisso comune. Questo cosiddetto bias Markoviano ha forte influenza sulla capacità computazionale del modello: in [23, 58] si mostra come una Rete Neurale Ricorrente che abbia una funzione di transizione di stato contrattiva e spazio degli stati limitato possa essere approssimata arbitrariamente bene dalla classe dei modelli su sequenze con dinamiche di stato Markoviane (e.g. Variable Memory Length Markov Models [48]). Ne risulta che, anche senza alcun learning, gli stati interni corrispondenti a sequenze di input con suffissi comuni tendano ad essere naturalmente raggruppati o, in altri termini, che la rete abbia per una propria caratteristica architetturale la capacità di discriminare le sequenze di input sulla base del loro suffisso.
Questo bias architetturale Markoviano, come visto strettamente legato alla echo state property, rappresenta di fatto l'essenza del modello e ne giustifica l'intuizione di base: sfruttare le caratteristiche strutturali del reservoir per realizzare un processo di encoding in grado di discriminare sequenze con suffissi diversi anche in assenza di un processo specifico di adattamento, per limitare l'azione del learning alla sola, semplice, funzione di output g_{out}. Di contro, secondo una visione complementare, il vantaggio computazionale è ottenuto nelle ESN accettando la presenza di dinamiche definite a priori, che non vengono adattate per risolvere il task specifico affrontato e che limitano dunque la capacità espressiva del modello.

Nonostante l'apprendimento richieda un basso costo in termini computazionali, le ESN sono state applicate con successo a molti problemi ottenendo risultati migliori rispetto ad altri approcci precedenti (si veda ad esempio

[32], [51, pag. 8], o [38, pag. 5] per un elenco dettagliato). Il modello è inoltre semplice e segue un paradigma molto generale: questo offre ampi margini di scelta e sperimentazione nell'implementazione del learning, nella topologia, nella scelta delle funzioni utilizzate e addirittura nell'implementazione fisica della rete[4]. Fra gli aspetti positivi delle ESN è infine opportuno menzionare l'alta plausibilità biologica, che caratterizza il Reservoir Computing in generale e che continua a rappresentare un fattore importante nell'ambito delle Reti Neurali.

Le ESN mostrano tuttavia anche dei limiti: alcuni di natura intrinseca al modello, come l'alto numero di unità impiegate che rende difficile l'implementazione fisica con risorse hardware limitate [46], ed altri legati al fatto che la ricerca in materia sia ancora in una fase poco avanzata. Gli effetti del rumore nei dati in input o la replicabilità delle condizioni di buon funzionamento della rete sono ad esempio fattori determinanti ma non completamente spiegati, che possono mettere in dubbio l'applicabilità del modello ad ampie classi di problemi [46].

Per contestualizzare meglio quanto verrà descritto in seguito è inoltre opportuno sottolineare come la ricerca sulle ESN sia ad oggi in larghissima parte orientata all'apprendimento di trasduzioni di sequenze e come l'applicazione dei principi del Reservoir Computing nell'ambito del trattamento di domini più complessi, come alberi o grafi, sia da considerarsi di vivissima attualità.

[4]Si veda [10] per un esempio in cui la rete — basata però su LSM — viene implementata attraverso hardware "poco convenzionale".

1.5 Modelli per domini strutturati

La possibilità di trattare domini di input in cui le informazioni siano strutturate (e.g. grafi) rappresenta una delle sfide che attualmente dominano e danno impulso ad una parte del mondo del Machine Learning. Molti problemi dell'informatica, della chimica, relativi all'elaborazione del linguaggio naturale, all'analisi di immagini o documenti, trovano infatti una propria naturale formulazione all'interno di domini più complessi della semplice rappresentazione tramite vettori numerici o sequenze.

Nell'ambito neurale, le *Reti Neurali Ricorsive* [12, 54] generalizzano le Reti Neurali Ricorrenti (si veda il paragrafo 1.3) affinché possano gestire domini di input strutturati. Reti di questo tipo sono basate su un processo di encoding in cui gli stati vengono calcolati ricorsivamente in accordo con le relazioni topologiche fra i vertici dell'input. La possibilità di tenere in considerazione la topologia durante l'encoding e di far sì che l'intero processo sia adattivo, delimita la separazione, sia concettuale che pratica, tra le Reti Neurali Ricorsive e l'approccio classico, che prevede invece l'uso di conoscenza pregressa per codificare i dati (e.g. estraendo indici topologici [20]).

Il processo di encoding rappresenta dunque la chiave dell'approccio che caratterizza i modelli neurali ricorsivi: ne determina i vantaggi e, tuttavia, ne mette anche in luce i limiti. In particolare, per garantire che ogni input abbia una rappresentazione appropriata, ovvero per evitare condizioni cicliche nel processo di encoding, si ricorre all'assunzione di *causalità* del sistema di transizione [54]. Essendo la codifica "guidata" dalla topologia dell'input, si assume infatti che lo stato corrispondente ad un vertice dipenda unicamente

dall'etichetta del vertice stesso e dagli stati corrispondenti ai vertici discendenti da esso. Se questo ha il vantaggio di garantire la convergenza ha, d'altra parte, forte impatto sulla classe degli input trattabili: l'assunzione di causalità impone infatti la presenza di un ordinamento topologico fra i vertici, limitando il dominio di input alla classe dei Grafi Diretti Aciclici (che comprende gli alberi radicati).

Nella storia recente delle Reti Neurali Ricorsive si ritrovano tuttavia dei modelli tesi a superare le limitazioni imposte dal vincolo di causalità. Ai fini di una completa comprensione del lavoro svolto, siamo particolarmente interessati a menzionare due approcci che, seppur non direttamente legati fra loro, trovano una sintesi nei modelli proposti.

Prima con la *Contextual Recursive Cascade Correlation* (CRCC) [42, 21] poi con il modello *Neural Network for Graphs* (NN4G) [41], si fa leva sull'approccio costruttivo per limitare e poi superare gli effetti dell'assunzione di causalità. In particolare NN4G sfrutta il processo incrementale di costruzione della rete per comporre una rete feedforward, che non risenta dunque delle eventuali dipendenze cicliche nell'input. I modelli citati mettono anche l'accento sull'importanza del fatto che, durante il processo di costruzione della rete, le nuove unità possano usufruire delle informazioni rese disponibili dalle unità precedenti (i.e. le unità "congelate"). Questa caratteristica, vera anche nel caso delle reti feedforward, acquisisce particolare rilevanza nel caso di input strutturati: lo stato di ogni unità è infatti funzione anche della topologia dell'input. Per questo le informazioni trasmesse alle unità successive contribuisco alla formazione di un *contesto*, che cresce con il crescere della rete [41]. L'approccio costruttivo permette dunque a questi modelli di sfruttare

informazioni contestuali altrimenti inaccessibili.

Un secondo approccio per garantire la stabilità del processo di encoding anche in presenza di dipendenze cicliche nell'input fa invece affidamento su un setting contrattivo della funzione di transizione di stato (si veda il paragrafo 1.4.1). Rientrano in questo filone i modelli *Graph Neural Network* (GNN) [50] e le *Graph Echo State Network* (GraphESN). Mentre in GNN la contrattività è il risultato di un processo iterativo di apprendimento, che coinvolge tutti i pesi della rete, le GraphESN ricorrono alla contrattività della funzione di transizione di stato in accordo ai principi del Reservoir Computing, sfruttando dunque la presenza di un reservoir fisso e di un readout adattivo. In quanto diretta progenitrice dei modelli proposti, quest'ultima tipologia di reti verrà discussa in dettaglio nel seguito (si veda il paragrafo 1.5.2), non prima di aver introdotto formalmente il problema dell'apprendimento su domini strutturati.

1.5.1 Trasduzioni su domini strutturati

Nel corso di questo paragrafo sarà formulato il problema dell'apprendimento su domini strutturati, introducendo terminologia e notazione che verranno utilizzate nel corso della tesi.

Un grafo $\mathbf{g}$, appartenente ad un insieme di grafi $\mathcal{G}$, è una coppia $(V(\mathbf{g}), E(\mathbf{g}))$, dove $V(\mathbf{g})$ denota l'insieme dei vertici di $\mathbf{g}$ ed $E(\mathbf{g}) = \{(u, v)|\ u, v \in V(\mathbf{g})\}$ denota l'insieme di archi di $\mathbf{g}$. Per semplicità nel seguito gli insiemi $V(\mathbf{g})$ e $E(\mathbf{g})$ saranno talvolta indicati come V ed E rispettivamente, mantenendo implicito il riferimento al grafo $\mathbf{g}$. Nel caso di grafi indiretti definiamo i

vicini del vertice v di un grafo come l'insieme dei vertici ad esso adiacenti[5], $\mathcal{N}(v) = \{u \in V |\ (u, v) \in E\}$. Sia il grado di un vertice v il numero dei suoi vicini, $degree(v) = |\mathcal{N}(v)|$, indichiamo con k il *grado massimo* riscontrabile sull'insieme $\mathcal{G}$. Diciamo che due grafi $\mathbf{g}_1 = (V(\mathbf{g}_1), E(\mathbf{g_1}))$ e $\mathbf{g}_2 = (V(\mathbf{g}_2), E(\mathbf{g_2}))$ sono *isomorfi* se esiste una funzione bigettiva $f : V(\mathbf{g}_1) \rightarrow V(\mathbf{g}_2)$ tale che $(u, v) \in E(\mathbf{g_1})$ se e solo se $(f(u), f(v)) \in E(\mathbf{g_2})$ (si veda anche [22]).

Consideriamo che ad ogni vertice di un grafo sia associata un'etichetta numerica $\mathbf{u}(v) \in \mathbb{R}^{N_U}$, dove $\mathbb{R}^{N_U}$ denota uno spazio di input di etichette vettoriali. Indichiamo l'insieme dei grafi con etichette in $\mathbb{R}^{N_U}$ con $(\mathbb{R}^{N_U})^{\#}$.

Obiettivo di un modello per domini strutturati è l'apprendimento di una *trasduzione strutturale* $\mathcal{T}$, ovvero di una funzione da un dominio di grafi in input $(\mathbb{R}^{N_U})^{\#}$ ad un dominio di grafi in output $(\mathbb{R}^{N_Y})^{\#}$:

$$\mathcal{T} : (\mathbb{R}^{N_U})^{\#} \rightarrow (\mathbb{R}^{N_Y})^{\#} \tag{1.23}$$

Indichiamo con $\mathbf{y}(v) \in \mathbb{R}^{N_Y}$ l'etichetta vettoriale associata al vertice v del grafo nel dominio di output.

Distinguiamo due tipi di trasduzioni. In una trasduzione *structure-to-structure* (anche chiamata *node-focused* [50]) il grafo in output è isomorfo a quello in input, ovvero $\mathcal{T}$ associa un vertice in output ad ogni vertice dell'input, mantenendone inalterata la struttura. In una trasduzione *structure-to-element* (o *graph-focused* [50]), invece, un singolo output vettoriale, $\mathbf{y}(\mathbf{g}) \in \mathbb{R}^{N_Y}$, è associato all'intero grafo. La figura 1.4 e la figura 1.5 mostrano

[5]Nel caso di grafi diretti è possibile un'ulteriore suddivisione dei vertici adiacenti in *predecessori* e *successori*. Per semplicità e per una maggiore aderenza ai task affrontati questa formulazione viene tralasciata.

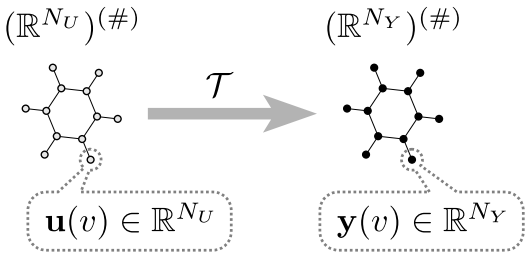


Figura 1.4: Trasduzione structure-to-structure.

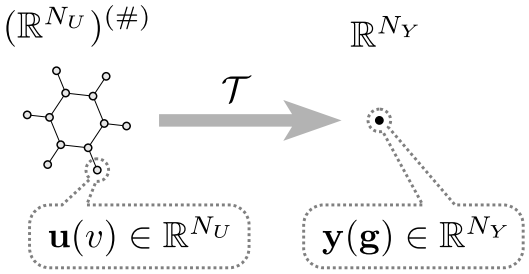


Figura 1.5: Trasduzione structure-to-element.

schematicamente i due tipi di trasduzione strutturale descritti.

Nel caso dell'apprendimento supervisionato, l'allenamento avviene avvalendosi di un training-set $\mathfrak{T} = \{(\mathbf{g}, \mathbf{y}_{\text{target}}(g))|\ \mathbf{g} \in \mathcal{G}, \mathbf{y}_{\text{target}}(\mathbf{g}) \in (\mathbb{R}^{N_Y})^{\#}\}$, dove $\mathbf{y}_{\text{target}}(\mathbf{g})$ indica il target associato ad uno specifico input e può essere un grafo etichettato (isomorfo a $\mathbf{g}$) nel caso di trasduzioni structure-to-structure o un vettore di reali a dimensione fissa nel caso di trasduzioni structure-to-element.

1.5.2 Graph Echo State Networks

Le *Graph Echo State Network* (GraphESN) [15] rappresentano un'estensione delle ESN (si veda il paragrafo 1.4.1) — e delle TreeESN [16] — per il trattamento del dominio dei grafi.

In maniera simile ad una ESN, in una GraphESN sono distinguibili tre layer: uno di *input*, uno nascosto detto *reservoir* e formato da unità ricorsive e

non lineari ed infine un *readout* feedforward. Anche in questo caso è richiesto che la funzione di transizione di stato del reservoir sia contrattiva. Il reservoir viene dunque inizializzato affinché rispetti tale vincolo e rimane poi inalterato, mentre il solo readout viene allenato.
È importante sottolineare come la contrattività della funzione di transizione di stato assuma in questo caso un significato specifico di grossa rilevanza, ampliando la classe delle strutture supportate dal modello. La contrattività garantisce infatti la stabilità del processo di codifica anche nel caso di dipendenze cicliche fra le variabili di stato (si veda il paragrafo 1.5), determinando l'applicabilità del modello ad una classe di strutture che comprende grafi non diretti e/o ciclici.

Passiamo ora a caratterizzare più formalmente una GraphESN come modello per l'apprendimento di trasduzioni strutturali su un dominio di grafi.

Riprendendo quanto descritto in precedenza (si veda il paragrafo 1.5.1), una trasduzione strutturale (1.23) può essere efficacemente decomposta come

$$\mathcal{T} = \mathcal{T}_{\text{out}} \circ \mathcal{T}_{\text{enc}} \tag{1.24}$$

dove $\mathcal{T}_{\text{enc}} : (\mathbb{R}^{N_U})^{\#} \to (\mathbb{R}^{N_R})^{\#}$ è una *funzione di encoding*, che mappa l'input in un dominio strutturato di features, e $\mathcal{T}_{\text{out}} : (\mathbb{R}^{N_R})^{\#} \to (\mathbb{R}^{N_Y})^{\#}$ la *funzione di output*. La figura 1.6 mostra la realizzazione di una generica trasduzione strutturale (structure-to-structure) attraverso la composizione di una funzione di encoding, $\mathcal{T}_{\text{enc}}$, ed una funzione di output, $\mathcal{T}_{\text{out}}$.

Secondo l'approccio neurale — ricorsivo e ricorrente — la funzione di encoding $\mathcal{T}_{\text{enc}}$ viene realizzata da un sistema dinamico; lo spazio delle features

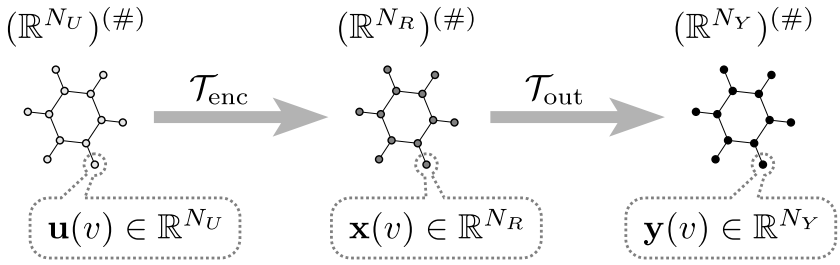


Figura 1.6: Decomposizione di una trasduzione strutturale tramite una funzione di encoding $\mathcal{T}_{\text{enc}}$ ed una funzione di output $\mathcal{T}_{\text{out}}$.

$(\mathbb{R}^{N_R})^{\#}$ è composto da variabili di stato, associate ad ogni vertice dell'input. Chiamiamo $\mathbf{x}(v) \in \mathbb{R}^{N_R}$ l'informazione di stato associata al vertice v del grafo in input $\mathbf{g}$. Indichiamo inoltre con $\mathbf{x}(\mathbf{g}) \in \mathbb{R}^{|V(\mathbf{g})|N_R}$ la concatenazione degli stati associati a tutti i vertici dell'input, secondo un ordine arbitrario, e con $\mathbf{x}(\mathcal{N}(v)) \in \mathbb{R}^{|\mathcal{N}(v)|N_R}$ la concatenazione degli stati associati ai vicini di un vertice v. La funzione $\mathcal{T}_{\text{enc}}$ associa dunque ad ogni vertice dell'input $v \in V(\mathbf{g})$ una corrispondente informazione di stato secondo una *funzione locale di encoding* τ

$$\mathbf{x}(v) = \tau(\mathbf{u}(v), \mathcal{N}(v)) \tag{1.25}$$

che esprime la dipendenza dello stato associato al vertice v da quelli associati ai suoi vicini. Equivalentemente possiamo dire che l'applicazione dell'equazione (1.25) ad ogni vertice dell'input realizza la *funzione globale di encoding* $\hat{\tau}$

$$\mathbf{x}(\mathbf{g}) = \hat{\tau}(\mathbf{g}, \mathbf{x}(\mathbf{g})) \tag{1.26}$$

Dato un grafo in input $\mathbf{g}$, dunque, l'output della funzione di encoding $\mathcal{T}_{\text{enc}}$ corrisponde alla soluzione dell'equazione (1.26).

In una GraphESN la funzione $\mathcal{T}_{\text{enc}}$ viene realizzata dal reservoir, che calcola

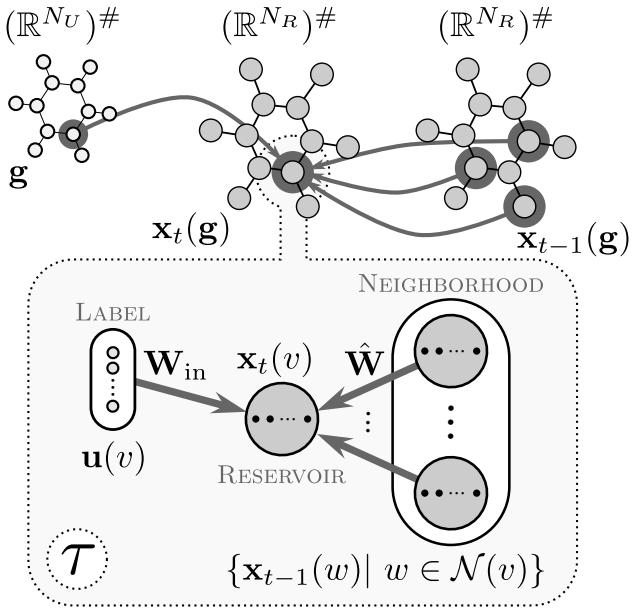


Figura 1.7: Schema di applicazione della funzione di transizione locale di stato ad un vertice v dell'input $\mathbf{g}$.

iterativamente una *funzione di transizione locale di stato*, versione iterativa dell'equazione (1.25). Al passo t, la funzione associa ad ogni vertice v una codifica, o valore di stato, $\mathbf{x}_t(v)$ secondo la seguente equazione

$$\mathbf{x}_t(v) = \tau(\mathbf{u}(v), \mathbf{x}_{t-1}(\mathcal{N}(v))) = f(\mathbf{W}_{\text{in}}\mathbf{u}(n) + \sum_{w \in \mathcal{N}(v)} \hat{\mathbf{W}}\mathbf{x}_{t-1}(w)) \qquad (1.27)$$

dove $\mathbf{W}_{\text{in}} \in \mathbb{R}^{N_R \times (N_U+1)}$ è la matrice dei pesi sulle connessioni tra input e reservoir, $\hat{\mathbf{W}} \in \mathbb{R}^{N_R \times N_R}$ è la matrice dei pesi ricorrenti tra i vicini di un vertice ed f è la funzione di attivazione, tipicamente sigmoidale, delle unità del reservoir. La figura 1.7 mostra un passo del procedimento iterativo di encoding, evidenziando come lo stato calcolato per ogni vertice v dipenda dall'etichetta numerica di input corrispondente, $\mathbf{u}(v)$, nonché dallo stato in corrispondenza dei vertici vicini calcolato al passo precedente, $\mathbf{x}(\mathcal{N}(v))$.

Per poter garantire che l'encoding converga, e quindi che l'equazione (1.26) abbia una soluzione, si ricorre ad un setting contrattivo della funzione τ. Perché τ sia *contrattiva* rispetto allo stato, è necessario che valga la seguente condizione:

$$\exists C \in [0,1) \text{ tale che } \forall \mathbf{u} \in \mathbb{R}^{N_U}, \forall \mathbf{x}_1, \ldots, \mathbf{x}_k, \mathbf{x}'_1, \ldots, \mathbf{x}'_k \in \mathbb{R}^{N_R} :$$
$$\|\tau(\mathbf{u}, \mathbf{x}_1, \ldots, \mathbf{x}_k) - \tau(\mathbf{u}, \mathbf{x}'_1, \ldots, \mathbf{x}'_k)\| \leq C \max_{i=1,\ldots,k} \|\mathbf{x}_i - \mathbf{x}'_i\| \quad (1.28)$$

dove $\|\cdot\|$ è una qualsiasi norma su $\mathbb{R}^{N_R}$. Considerando per τ l'implementazione dell'equazione (1.27), con $f = \tanh$ usata come funzione di attivazione delle unità del reservoir, e scegliendo la distanza euclidea come norma, risulta che la contrattività è garantita per [18]

$$\sigma = \|\hat{\mathbf{W}}\|_2 k < 1 \quad (1.29)$$

dove k è il grado massimo calcolato su tutti i grafi in $\mathcal{G}$ e σ, che controlla il grado di contrattività delle dinamiche del reservoir, è chiamato *coefficiente di contrazione*. Una volta che la contrattività di τ sia assicurata, la convergenza del calcolo iterativo del reservoir è garantita dal *Principio di Contrazione di Banach* [40] per ogni stato iniziale $\mathbf{x}_0(\mathbf{g})$; nella pratica questo permette di calcolare la codifica di un grafo in maniera iterativa, interrompendo l'elaborazione una volta che, per ogni vertice del grafo, la distanza tra due stati successivi nel processo di encoding sia inferiore ad una soglia prefissata ϵ: $\|\mathbf{x}_t(v) - \mathbf{x}_{t-1}(v)\|_2 \leq \epsilon$. L'algoritmo 1 nella pagina seguente mostra il processo di codifica ricorsivo implementato dal reservoir di una GraphESN.

Algoritmo 1 GraphESN: algoritmo iterativo di encoding.

for all $\mathbf{g} \in \mathcal{G}$ **do**
  $t = 0$
  for all $v \in V(\mathbf{g})$ **do**
    $\mathbf{x}_0(v) = 0$
  end for
  repeat
    $t = t + 1$
    for all $v \in V(\mathbf{g})$ **do**
      $\mathbf{x}_t(v) = \tau(\mathbf{u}(v), \mathbf{x}_{t-1}(\mathcal{N}(v)))$
    end for
  until $\forall v \in V(\mathbf{g}) : \|\mathbf{x}_t(v) - \mathbf{x}_{t-1}(v)\|_2 \leq \epsilon$
end for
return $\mathbf{x}(\mathbf{g})$

È opportuno sottolineare come la contrattività della funzione di transizione di stato caratterizza le GraphESN sotto diversi importanti aspetti, che vanno oltre quello algoritmico. Innanzi tutto la stabilità del processo di codifica rende le GraphESN in grado di gestire grafi ciclici, il che rappresenta invece un problema per le reti ricorrenti classiche. Inoltre la contrattività implica la echo state property (si veda il paragrafo 1.4.1), garantendo che gli stati calcolati dal reservoir dipendano (asintoticamente) unicamente dal grafo in input e non dallo stato iniziale. Infine il fatto che τ sia contrattiva determina, come nel caso delle ESN, una caratterizzazione Markoviana delle dinamiche del reservoir, con il concetto di *suffisso comune* esteso, rispetto al caso di sequenze [17] o alberi [16], al concetto di *vicinato comune* [15]. La figura 1.8 nella pagina successiva mostra quali siano i vertici che contribuiscono al calcolo dello stato corrispondente ad un vertice v, evidenziando come nel corso di ogni iterazione la "frontiera" dei vertici coinvolti si allarghi. È bene sottolineare come, nella sua schematizzazione, la figura non renda conto del fatto che uno

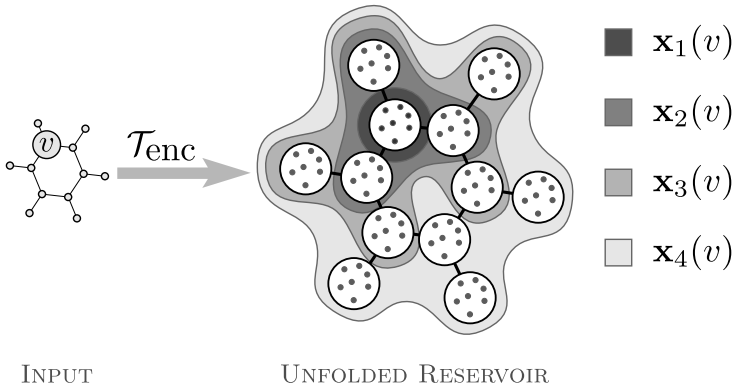


Figura 1.8: Vertici coinvolti nel calcolo di $\mathbf{x}_t(v)$ durante quattro passi del processo di encoding.

stesso vertice possa contribuire al calcolo dello stato di v sia direttamente (i.e. il vertice appartiene a $\mathcal{N}(v)$) sia in maniera indiretta (i.e. per ogni cammino che partendo da v si estenda, vicino dopo vicino ed iterazione dopo iterazione, fino al vertice in questione).

Nel caso di trasduzioni structure-to-element (si veda il paragrafo 1.5.1) si ricorre ad un'ulteriore funzione $\mathcal{X} : (\mathbb{R}^{N_R})^{\#} \to \mathbb{R}^{N_R}$, detta *state mapping function*, che applicata al risultato dell'encoding nello spazio strutturato di features restituisce una rappresentazione in uno spazio vettoriale a dimensione fissa per l'intero grafo. In questo caso l'equazione (1.24) viene dunque decomposta in

$$\mathcal{T} = \mathcal{T}_{\text{out}} \circ \mathcal{X} \circ \mathcal{T}_{\text{enc}} \tag{1.30}$$

come mostrato dalla figura 1.9 nella pagina seguente.

Benché la scelta della funzione $\mathcal{X}$ sia arbitraria, si distinguono principalmente due alternative. Con *supersource state mapping* [16], lo stato dell'intero grafo $\mathcal{X}(\mathbf{x}(\mathbf{g}))$ viene mappato nello stato di un solo vertice *supersource*, qualora per la natura dei dati o del problema sia individuabile un vertice che

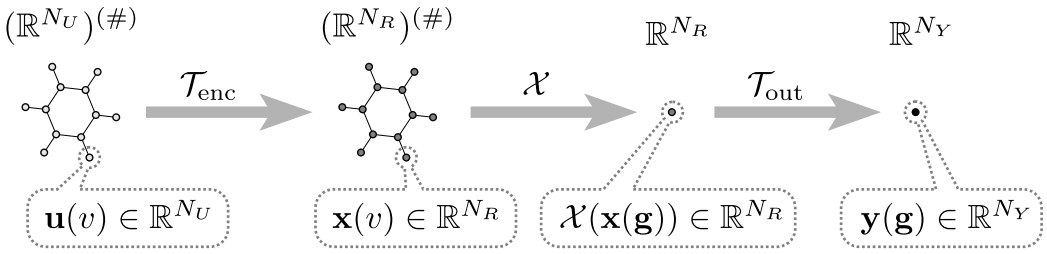


Figura 1.9: Decomposizione di una trasduzione structure-to-element tramite una funzione di encoding $\mathcal{T}_{\text{enc}}$, una state-mapping-function $\mathcal{X}$ ed una funzione di output $\mathcal{T}_{\text{out}}$.

dipenda da tutti gli altri[6]. In alternativa, con *mean state mapping*, $\mathcal{X}(\mathbf{x}(\mathbf{g}))$ viene calcolato come la media degli stati corrispondenti ai vertici di $\mathbf{g}$:

$$\mathcal{X}(\mathbf{x}(\mathbf{g})) = \frac{1}{|V(\mathbf{g})|} \sum_{v \in V(\mathbf{g})} \mathbf{x}(v) \tag{1.31}$$

Come in una ESN, la funzione di output $\mathcal{T}_{\text{out}}$ è realizzata da un *readout* formato da N_Y unità che ricevono input dal reservoir (o dalla state mapping function).

Nel caso di trasduzioni structure-to-structure, la *funzione di output locale* applicata agli stati corrispondenti ai singoli vertici è la seguente:

$$\mathbf{y}(v) = g_{\text{out}}(\mathbf{x}(v)) = f_{\text{out}}(\mathbf{W}_{\text{out}}\mathbf{x}(v)) \tag{1.32}$$

dove $\mathbf{y}(v) \in \mathbb{R}^{N_Y}$ è il vettore dei valori di output per il vertice v, $\mathbf{W}_{\text{out}} \in \mathbb{R}^{N_Y \times (N_R+1)}$ è la matrice dei pesi delle connessioni tra reservoir e readout ed f_{out} è la funzione di attivazione delle unità di output, tipicamente lineare.

[6]Questa caratteristica, naturale nel caso di alberi radicati e sequenze, è comunque realizzabile "artificialmente" per qualsiasi grafo aggiungendo un vertice che sia collegato a tutti gli altri.

L'applicazione di g_{out} all'encoding ottenuto per ogni vertice dell'input definisce il calcolo della funzione di output $\mathcal{T}_{\text{out}}$.
Per trasduzioni structure-to-element, l'output a dimensione fissa relativo all'intero grafo, $\mathbf{y}(\mathbf{g}) \in \mathbb{R}^{N_Y}$, è invece ottenuto applicando la funzione di output locale al risultato della state mapping function:

$$\mathbf{y}(\mathbf{g}) = g_{\text{out}}(\mathcal{X}(\mathbf{x}(\mathbf{g})) = f_{\text{out}}(\mathbf{W}_{\text{out}}\mathcal{X}(\mathbf{x}(\mathbf{g}))) \tag{1.33}$$

In entrambi i casi, ed in maniera simile a quanto accade in una ESN standard, l'allenamento di una GraphESN prevede l'adattamento dei pesi della matrice $\mathbf{W}_{\text{out}}$, realizzabile attraverso regressione lineare[7] sulla base dei valori di target $\mathbf{y}_{\text{target}}(v)$, o $\mathbf{y}_{\text{target}}(\mathbf{g})$ nel caso di trasduzioni structure-to-element.

La possibilità di limitare l'apprendimento alla sola funzione di output g_{out} e la realizzazione dell'encoding sfruttando il setting contrattivo del reservoir caratterizzano positivamente le GraphESN dal punto di vista computazionale rispetto ad altri modelli neurali per il trattamento di domini strutturati.
Assumendo che il reservoir sia sparso, con M connessioni in input per ogni unità, il costo computazionale di un passo del processo di encoding ha costo

$$O(|V(\mathbf{g})|\ k\ N_R\ M) \tag{1.34}$$

ed ha dunque una dipendenza lineare sia con il numero di vertici dell'input che con le dimensioni del reservoir. Tale costo risulta molto inferiore, ad esempio, rispetto al processo di encoding realizzato durante la fase di training delle

[7] Purché f_{out} sia invertibile, è possibile ricondurre g_{out} ad una combinazione lineare semplicemente modificando i valori target: $\mathbf{y}'_{\text{target}} = f^{-1}_{\text{out}}(\mathbf{y}_{\text{target}})$.

GNN [50], che avviene attraverso centinaia o migliaia di iterazioni, ognuna corrispondente all'intero processo di codifica dell'input di una GraphESN. Il costo computazionale dell'encoding è confrontabile anche con metodi basati su kernel applicati al trattamento di grafi. Mantenendo l'assunzione che k sia il grado massimo riscontrabile sul dataset, l'applicazione dell'*optimal assignment kernel* e del *expected match kernel* [14] risulta infatti avere costo rispettivamente cubico e quadratico rispetto al numero di vertici dell'input. Il costo del trainig varia invece a seconda dell'algoritmo utilizzato, potendo comunque fare affidamento sull'impiego di algoritmi efficienti per l'allenamento di un unico layer di unità feedforward (si veda il paragrafo 1.1.3).

La figura 1.10 nella pagina successiva riassume schematicamente la struttura ed il funzionamento complessivo di una GraphESN che modelli traduzioni structure-to-element. Procedendo da sinistra verso destra, un grafo in input $\mathbf{g}$ viene codificato dal reservoir, che lo mappa in uno spazio delle features strutturato. La figura mostra in questo caso l'unfolding del reservoir, che viene "copiato" su ogni vertice dell'input. La varie copie così ottenute sono collegate, tramite connessioni con pesi in $\hat{\mathbf{W}}$, in accordo alla topologia del grafo (i.e. in base ai vicini di ogni vertice). Successivamente l'encoding ottenuto viene trasformato, dalla state mapping function $\mathcal{X}$, in un vettore di features a dimensione fissa. Il risultato dell'applicazione della state mapping function viene infine usato come input del readout, unica componente adattiva del modello, ottenendo in output il vettore a dimensione fissa $\mathbf{y}(\mathbf{g})$. In alto a sinistra la figura mostra la dipendenza che lega gli stati nello spazio delle features, sia all'etichetta di input del vertice corrispondente che agli stati calcolati in corrispondenza dei suoi vicini (i.e. un unico vicino nella figura).

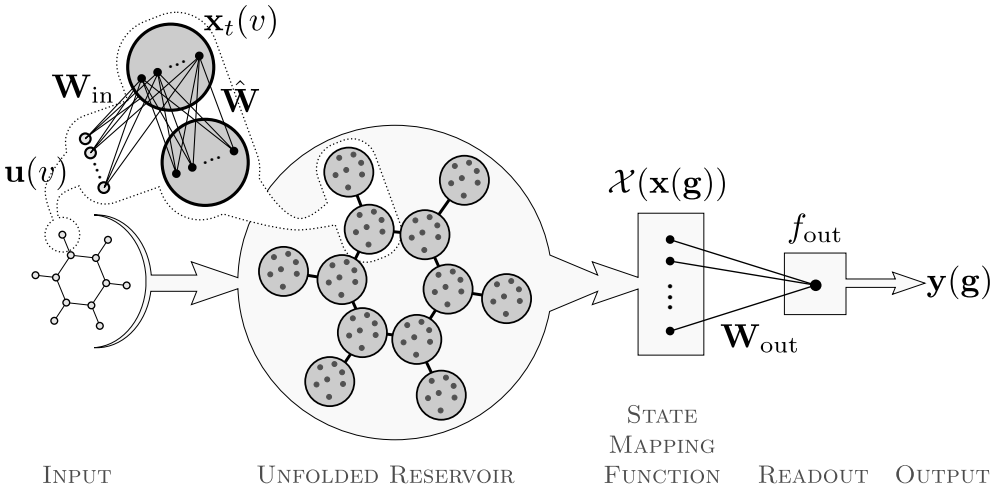


Figura 1.10: Schema di una GraphESN per l'apprendimento di trasduzioni structure-to-element.

2 Modelli

Il capitolo descrive i modelli realizzati, definendone sia le caratteristiche strutturali e topologiche che quelle computazionali.

Il paragrafo 2.1 fornisce un'introduzione ai modelli proposti, definendone i principi di base in relazione a quanto descritto nel capitolo 1.

Il paragrafo 2.2 introduce e discute i dettagli dell'estensione delle Graph-ESN al caso costruttivo.

Nel paragrafo 2.3 vengono descritti nello specifico i modelli oggetto della sperimentazione svolta, definendone le caratteristiche strutturali.

Nel paragrafo 2.4, viene presentata un'analisi del costo computazionale relativo all'uso dei modelli proposti.

Infine, il paragrafo 2.5 descrive le caratteristiche basilari dell'implementazione software dei modelli.

2.1 Introduzione ai modelli

Tecniche, paradigmi e modelli descritti nel capitolo 1, benché appartenenti a periodi ed ambiti diversi, contribuiscono a formare i tratti distintivi dei

modelli che saranno discussi nel corso del capitolo. Nel seguito saranno infatti introdotti dei modelli neurali, appartenenti all'ambito del Reservoir Computing, che adottino una strategia costruttiva nel trattamento di dati strutturati.

I modelli realizzati sono stati concepiti secondo un approccio incrementale, mirato ad introdurre e poi sfruttare l'approccio costruttivo con l'obiettivo di proporre soluzioni ad alcuni problemi aperti nell'ambito del Reservoir Computing. Ognuno dei modelli individua dunque nuove funzionalità specifiche, generalizzando i precedenti.

(i) Il modello *GraphESN Constructive Flat* (GraphESN-CF) introduce una strategia costruttiva usando le GraphESN come unità computazionali. L'approccio costruttivo permette al modello di affrontare la necessità di fissare la topologia della rete a priori, particolarmente importante nel caso del Reservoir Computing data la presenza di reservoir non adattivi. Attraverso un procedimento iterativo di costruzione della rete, le GraphESN-CF sono dunque in grado di determinare il numero di unità da impiegare in maniera automatica e dipendente dal task: le singole sotto-reti vengono allenate per apprendere, e correggere, il segnale di errore commesso dalla rete, specializzandosi dunque ognuna in un sotto-task specifico, e vengono aggiunte alla rete in modo da formare una topologia *flat*, che non preveda connessioni fra sotto-reti distinte.

(ii) Il modello *GraphESN Forward* (GraphESN-FW) estende il precedente in modo da sfruttare le informazioni apprese nel corso del processo di costruzione incrementale della rete. Riprendendo la strategia della

Cascade Correlation (si veda il paragrafo 1.2.1), in questo caso vengono aggiunte delle connessioni (in avanti, o *forward*) fra l'output di ogni sotto-rete verso i readout di tutte le sotto-reti seguenti: quanto appreso da ogni sotto-rete contribuisce dunque alle fasi di learning successive, con lo scopo di semplificare la risoluzione del sotto-task affrontato. Grazie al fatto che ogni sotto-rete realizza una trasformazione non lineare dell'input, la GraphESN-FOF introduce una rete con topologia multilayer non lineare nell'ambito del Reservoir Computing.

(iii) Il modello *GraphESN Forward Output-Feedback* (GraphESN-FOF) generalizza i due modelli precedenti introducendo uno schema output-feedback capace influenzare le dinamiche del reservoir. L'approccio costruttivo è in questo caso sfruttato per realizzare una strategia mirata ad affrontare uno dei problemi caratteristici del Reservoir Computing: l'impiego di un processo di encoding prefissato, guidato dalle caratteristiche dell'input ma non da quelle del task affrontato.

La topologia in questo caso si sviluppa connettendo gli output di ogni sotto-rete sia al readout che ai reservoir delle successive. I segnali di output delle sotto-reti, ottenuti tramite apprendimento ed utilizzabili sia su dati di training che su dati di test, vengono quindi introdotti all'interno del processo di codifica degli input, che pur rimanendo non adattivo beneficia di informazioni supervisionate, legate dunque al task affrontato.

Altri aspetti caratterizzano i tre modelli realizzati. Come appartenenti all'ambito del Reservoir Computing, i modelli fanno affidamento su una fase

di learning estremamente vantaggiosa dal punto di vista computazionale, che interessa solo una parte delle connessioni della rete (si veda il capitolo 1.4). Infine, essendo estensioni delle GraphESN, i modelli proposti risultano in grado di apprendere trasduzioni strutturali generiche, che abbiano come input anche grafi non diretti o grafi diretti che presentino dei cicli (si veda il paragrafo 1.5.2).

Nel seguito i modelli verranno descritti nel dettaglio, privilegiando laddove possibile la formulazione più generale, ovvero riferendosi principalmente a GraphESN-FOF, in modo da non rendere la trattazione ridondante. Eventuali differenze specifiche, prevalentemente riferite alla topologia della rete, verranno evidenziate dove necessario.

2.2 GraphESN costruttive

Stando anche a quanto descritto in precedenza (si veda il paragrafo 1.2 a pagina 15), l'approccio costruttivo prevede l'uso di singole unità computazionali, o sotto-reti, che interconnesse fra loro formano una rete nella sua totalità. Prima di poter descrivere i modelli sviluppati e definirne la topologia è dunque necessario dare una formulazione esatta di quali siano le unità di cui sono composti.

Come avviene in generale per i modelli costruttivi, consideriamo il caso in cui ogni nuova unità computazionale prenda in input gli output delle sotto-reti precedenti. Ogni sotto-rete agisce quindi come una sorta di feature-detector adattivo e la sua uscita può essere trattata come un input aggiuntivo dalle sotto-reti seguenti. Per far sì che più reti possano essere interconnesse fra loro,

estendiamo reservoir e readout di una GraphESN standard in modo da rendere possibile la presenza di nuove connessioni (i.e. gli output di altre sotto-reti) che chiamiamo *output-feedback*, provenienti da altre unità computazionali che operano indipendentemente da quella considerata.

Nel caso di trasduzioni structure-to-structure, gli output-feedback rappresentano valori calcolati in corrispondenza di ogni vertice. Il reservoir esteso della sotto-rete i-esima calcola, per un grafo in input $\mathbf{g}$ ed al passo t del processo di codifica, un valore di stato $\mathbf{x}_t^{(i)}(v) \in \mathbb{R}^{N_R^{(i)}}$ per ogni vertice $v \in V(\mathbf{g})$ secondo la seguente equazione

$$\begin{aligned} \mathbf{x}_t^{(i)}(v) &= \tau(\mathbf{u}(v), \mathbf{x}_{t-1}^{(i)}(\mathcal{N}(v)), \mathbf{z}^{(1)}(v), \ldots, \mathbf{z}^{(i-1)}(v)) \\ &= f(\mathbf{W}_{\text{in}}^{(i)}\,\mathbf{u}(v) + \sum_{w \in \mathcal{N}(v)} \hat{\mathbf{W}}^{(i)}\,\mathbf{x}_{t-1}^{(i)}(w) + \sum_{j=1}^{i-1} \mathbf{W}_{\text{fof}}^{(ij)}\,\mathbf{z}^{(j)}(v)) \end{aligned} \tag{2.1}$$

dove $\mathbf{z}^{(j)}(v) \in \mathbb{R}^{N_Z^{(j)}}$ rappresenta l'uscita della sotto-rete j-esima e $\mathbf{W}_{\text{fof}}^{(ij)} \in \mathbb{R}^{N_R^{(i)} \times N_Z^{(j)}}$ è la matrice dei pesi per i (forward) output-feedback: connessioni tra il reservoir i-esimo e l'uscita della j-esima sotto-rete.

Nel caso di task structure-to-element cambia la natura degli output delle singole sotto-reti e l'equazione di transizione di stato del reservoir viene modificata di conseguenza come

$$\begin{aligned} \mathbf{x}_t^{(i)}(v) &= \tau(\mathbf{u}(v), \mathbf{x}_{t-1}^{(i)}(\mathcal{N}(v)), \mathbf{z}^{(1)}(\mathbf{g}), \ldots, \mathbf{z}^{(i-1)}(\mathbf{g})) \\ &= f(\mathbf{W}_{\text{in}}^{(i)}\,\mathbf{u}(v) + \sum_{w \in \mathcal{N}(v)} \hat{\mathbf{W}}^{(i)}\,\mathbf{x}_{t-1}^{(i)}(w) + \sum_{j=1}^{i-1} \mathbf{W}_{\text{fof}}^{(ij)}\,\mathbf{z}^{(j)}(\mathbf{g})) \end{aligned} \tag{2.2}$$

con $\mathbf{z}^{(j)}(\mathbf{g}) \in \mathbb{R}^{N_Z^{(j)}}$.

Al netto della modifica effettuata, il processo di codifica dell'input da parte del reservoir rimane inalterato in entrambi i casi (si veda il paragrafo 1.5.2 e l'algoritmo 1 a pagina 41).

Il readout delle sotto-reti viene invece modificato considerando gli output-feedback come parte della codifica del grafo in input. In altri termini, la codifica viene arricchita da nuove informazioni o features che, calcolate dalle precedenti sotto-reti, vengono concatenate al vettore risultato del processo di encoding del reservoir.
Nel caso di trasduzioni structure-to-structure, l'uscita $\mathbf{z}^{(i)}(v) \in \mathbb{R}^{N_Z^{(i)}}$ della sotto-rete i-esima è quindi data da

$$\begin{aligned} \mathbf{z}^{(i)}(v) &= g_{\text{out}}(\mathbf{x}^{(i)}(v), \mathbf{z}^{(1)}(v), \ldots, \mathbf{z}^{(i-1)}(v)) \\ &= f_{\text{out}}(\mathbf{W}_{\text{out}}^{(i)} [\mathbf{x}^{(i)}(v), \mathbf{z}^{(1)}(v), \ldots, \mathbf{z}^{(i-1)}(v)]) \end{aligned} \tag{2.3}$$

con $[\mathbf{x}^{(i)}(v), \mathbf{z}^{(1)}(v), \ldots, \mathbf{z}^{(i-1)}(v)]$ concatenazione della codifica corrispondente al vertice v e degli output-feedback e con la matrice dei pesi del readout opportunamente modificata per adattarsi alla nuova dimensione dello spazio delle features, $\mathbf{W}_{\text{out}}^{(i)} \in \mathbb{R}^{N_Z^{(i)} \times (N_R^{(i)} + \sum_{j=1}^{i-1} N_Z^{(j)} + 1)}$.

Per le trasduzioni structure-to-element si procede in maniera simile, calcolando l'output relativo all'intero grafo $\mathbf{z}^{(i)}(\mathbf{g})$ come

$$\begin{aligned} \mathbf{z}^{(i)}(\mathbf{g}) &= g_{\text{out}}(\mathcal{X}(\mathbf{x}^{(i)}(\mathbf{g})), \mathbf{z}^{(1)}(\mathbf{g}), \ldots, \mathbf{z}^{(i-1)}(\mathbf{g})) \\ &= f_{\text{out}}(\mathbf{W}_{\text{out}}^{(i)} [\mathcal{X}(\mathbf{x}^{(i)}(\mathbf{g})), \mathbf{z}^{(1)}(\mathbf{g}), \ldots, \mathbf{z}^{(i-1)}(\mathbf{g})]) \end{aligned} \tag{2.4}$$

Poiché la natura del readout rimane sostanzialmente invariata rispetto al caso delle GraphESN, non sono necessari particolari accorgimenti per la fase

di learning, che può essere eseguita con le tecniche standard.

È importante sottolineare alcuni aspetti relativi alla formulazione data delle singole unità computazionali.

- La presenza degli output-feedback è opzionale (e.g. $\mathbf{W}_{\text{fof}}^{(ij)} = \mathbf{0}$). Questo fornisce la possibilità di implementare diverse varianti architetturali (si vedano i paragrafi 2.2.1 e 2.3) e permette di caratterizzare le singole unità computazionali come vere e proprie estensioni delle GraphESN standard. La presenza opzionale degli output-feedback determina anche il fatto che i tre modelli realizzati si generalizzino a vicenda, secondo la gerarchia descritta in precedenza nel paragrafo 2.1.

- Non è stato posto alcun vincolo particolare sulle caratteristiche di una singola unità computazionale in relazione al fatto che più sotto-reti debbano interagire fra loro. Benché ognuna sia predisposta per ricevere gli output-feedback dalle precedenti è dunque possibile che ciascuna sotto-rete differisca dalle altre nelle dimensioni del reservoir, del readout o dell'output. Questa caratteristica consente quindi l'impiego di sotto-reti eterogenee sia nelle dimensioni che nel sotto-task affrontato, permettendo la realizzazione di numerose varianti implementative.

- I pesi dei singoli output-feedback sono tutti indipendenti l'uno dall'altro. Ne risulta un'ampia gamma di possibilità nell'implementare politiche di feedback specifiche, ad esempio facendo in modo che alcune sotto-reti abbiano maggiore influenza sulle altre (i.e. abbiano pesi maggiori sulle connessioni di output-feedback).

- Gli output-feedback, agendo come input ausiliari, non modificano le condizioni per la contrattività del reservoir.

Nel seguito verrà descritto il funzionamento della rete nella sua totalità e sarà discusso in dettaglio il ruolo degli output-feedback.

2.2.1 Costruzione di una rete

Dopo aver definito come sono fatte le singole unità che formano la rete nel suo complesso, è possibile indicare quale sia l'approccio usato per combinarle.

Consideriamo una rete come formata da un *readout globale* al quale vengono collegate in ingresso le connessioni corrispondenti alle uscite di una o più sotto-reti, aggiunte incrementalmente. Al passo i-esimo, ovvero dopo aver aggiunto i sotto-reti, l'output della rete è dunque dato, nel caso di trasduzioni structure-to-structure, da:

$$\mathbf{y}^{(i)}(v) = f_{\text{out}}(\mathbf{W}_{\text{out}}\,[\mathbf{z}^{(1)}(v), \ldots, \mathbf{z}^{(i)}(v)]) \tag{2.5}$$

con $\mathbf{W}_{\text{out}} \in \mathbb{R}^{N_Y \times (\sum_{j=1}^{i} N_Z^{(i)} + 1)}$ matrice dei pesi delle connessioni del readout globale. Nel caso dell'apprendimento di trasduzioni structure-to-element l'output è calcolato, in maniera simile, come

$$\mathbf{y}^{(i)}(\mathbf{g}) = f_{\text{out}}(\mathbf{W}_{\text{out}}\,[\mathbf{z}^{(1)}(\mathbf{g}), \ldots, \mathbf{z}^{(i)}(\mathbf{g})]) \tag{2.6}$$

In entrambi i casi siamo in presenza di un unico livello di connessioni che può essere allenato attraverso tecniche efficienti, in analogia con quanto avviene nelle sotto-reti ed in accordo con l'approccio del Reservoir Computing.

Internamente alla rete, ogni nuova sotto-rete può essere collegata a tutte le altre sotto-reti esistenti: gli output-feedback possono andare al reservoir della nuova sotto-rete, al suo readout o ad entrambi, in accordo con quanto descritto nel paragrafo 2.2. Il modo in cui le sotto-reti sono connesse fra loro determina la differenza fra i modelli sperimentati, le cui topologie verranno descritte in seguito (si veda il paragrafo 2.3 a pagina 61).

Ogni sotto-rete viene allenata per *correggere l'errore* commesso dalla rete (i.e. dal readout globale). Riferendoci al caso di un task structure-to-element[1], indichiamo con $\mathbf{e}^{(i)}(\mathbf{g}) \in \mathbb{R}^{N_Y}$ l'errore commesso dalla rete al passo i-esimo (i.e. dopo aver aggiunto i sotto-reti) sull'input $\mathbf{g}$. La sotto-rete $(i+1)$-esima utilizza dunque $\mathbf{e}^{(i)}(\mathbf{g})$ come target nella propria fase di apprendimento. Una volta effettuato il training, la sotto-rete viene "congelata" ed aggiunta alla rete: il suo output diventa un nuovo input per il readout globale e può, eventualmente, essere usato come output-feedback per le sotto-reti successive. Ogni volta che una nuova unità computazionale allenata viene aggiunta alla rete si esegue un nuovo allenamento del readout globale, necessario a valutare se interrompere l'algoritmo o, in alternativa, a determinare i nuovi errori da correggere tramite ulteriori sotto-reti. L'allenamento del readout globale, il cui input ha una dimensionalità ridotta rispetto ai readout delle sotto-reti, ha tuttavia un costo computazionale contenuto. L'algoritmo procede dunque iterativamente aggiungendo nuove sotto-reti e modificando l'errore, finché non venga verificato un criterio di stop prefissato (e.g. l'errore di training non scenda al di sotto di una soglia prefissata).

[1] Il passaggio al caso di task structure-to-structure è banale e viene quindi tralasciato per chiarezza espositiva.

Algoritmo 2 GraphESN costruttiva. Caso structure-to-element.

```
init sub-network counter: i = 0
init error: e^(0)(g) = −y_target(g)
repeat
  for all 1 ≤ j ≤ pool_size do
    init j-th candidate
    connect candidate's output-feedback
    train candidate using e^(i)(g) as target
  end for
  select the winning candidate
  update sub-network counter: i = i + 1
  connect the winner (i.e. i-th sub-network) to the global readout
  train the global readout
  update error e^(i)(g)
until stop adding sub-networks
```

L'algoritmo 2 riassume il procedimento iterativo di costruzione della rete, mostrando anche come la fase di training delle sotto-reti possa avvalersi di un pool di candidate come avviene per la Cascade Correlation (si veda il paragrafo 1.2.1). In questo caso il pool di candidate può contenere sotto-reti che si differenziano, oltre che per l'esito dell'apprendimento, anche nell'iper-parametrizzazione o nell'inizializzazione dei pesi, in particolare dei pesi del reservoir che non vengono modificati dal training.

Poiché al primo passo non è disponibile alcuna informazione sull'errore commesso, la prima sotto-rete viene allenata considerando l'errore massimo: $\mathbf{e}^{(0)}(\mathbf{g}) = -\mathbf{y}_{\text{target}}(\mathbf{g}), \forall \mathbf{g} \in \mathcal{G}$. Tale scelta risponde ad un'esigenza ben precisa. Il segnale appreso da una sotto-rete gioca infatti un ruolo ambivalente rispetto alla risoluzione del task: da una parte serve al readout globale come indicazione degli errori da correggere, dall'altra sposta gli stati delle sotto-reti successive in modo da raggruppare i pattern potenzialmente già corretti (si veda il paragrafo 2.2.2). In quest'ottica è quindi importante che la prima sotto-rete

agisca in maniera analoga alle successive, seppur riferendosi ad un errore per certi versi artificiale.

È infine opportuno sottolineare che la strategia incrementale adottata permette di allenare il readout globale attraverso metodi iterativi (e.g. LMS, si veda il paragrafo 1.1.3). Tale pratica, comune nell'ambito delle Reti Neurali Artificiali, risulta infatti generalmente inapplicabile nel caso del Reservoir Computing a causa dell'alto numero di condizionamento delle matrici di input utilizzate per allenare quei readout che prendano i propri input direttamente da un reservoir [31], cosa che non avviene nel caso del readout globale, i cui input sono le uscite (precedentemente allenate) delle varie sotto-reti.

2.2.2 Output-feedback

Il ruolo degli output-feedback è di rilevanza centrale per i modelli proposti, in particolare rispetto al modo in cui questi possono influenzare le dinamiche del reservoir. Con riferimento al reservoir esteso dell'equazione (2.2) è infatti importante sottolineare come i valori in $\mathbf{z}^{(j)}(\mathbf{g})$ introducano *localmente* un'informazione *globale*, riferita all'intero grafo in input[2], e *supervisionata*, ottenuta cioè tramite il processo di apprendimento (i.e. gli output-feedback corrispondono agli output di altre sotto-reti, precedentemente allenate). Se per la sua natura Markoviana un reservoir ha la capacità di discriminare i vertici in base ai loro vicini (ed i vicini dei vicini, e così via, iterativamente), il reservoir esteso usufruisce quindi, sia in training che in test, anche di un'in-

[2]Nel caso di un task structure-to-structure, equazione (2.1), è forse più corretto parlare di informazione *contestuale* piuttosto che globale. Tale differenza nella terminologia è tuttavia irrilevante ai fini del discorso affrontato se non addirittura fuorviante se si considera che il processo di encoding di una GraphESN sfrutta di per sé informazioni contestuali, anche se di natura diversa da quelle qui riferite.

formazione che gli permette di "vedere oltre" la frontiera che determina lo stato corrente (si veda la figura 1.8) ed in base a questo di spostare gli stati in maniera consistente con il task affrontato, in funzione del target.

L'idea di sfruttare le caratteristiche dell'output per influenzare le dinamiche del reservoir è d'altra parte presente nell'ambito del Reservoir Computing sin dalla nascita delle ESN [29] e risponde all'esigenza di superare i limiti imposti dall'adozione di reservoir fissi, con dinamiche prefissate. Nonostante questo, ad oggi l'impiego di output-feedback nella loro formulazione originale comporta tecniche ed accorgimenti specifici [37, 64, 46] e presenta alcune limitazioni (si veda il paragrafo 1.4.1). I modelli proposti introducono dunque, in contrapposizione con i modelli esistenti di Reservoir Computing, un sistema di output-feedback stabile realizzato sfruttando le caratterstiche della strategia costruttiva.

Per comprenderne meglio la natura, la figura 2.1 mostra un esempio di come gli output-feedback possano modificare le dinamiche della rete. Con riferimento alla figura, si considerino (i) due sequenze in input $\mathbf{g}'$ e $\mathbf{g}''$, a cui siano associati target diversi (i.e. che debbano essere distinte dalla rete), che differiscono unicamente nell'ultimo vertice ed a cui sono associati valori di output-feedback distinti, indicati con r e q rispettivamente. Supponendo che lo spazio delle feature sia bidimensionale (i.e. $N_R = 2$), è possibile dare un'interpretazione geometrica ai vari input (e.g. $\mathbf{W}_{\text{in}}\,\text{a} = (0, 1)$), che supponiamo essere come è indicato in (ii). In (iii) sono mostrate le traiettorie corrispondenti all'encoding iterativo del primo vertice della sequenza, indicato con v' e v'' rispettivamente per i due input o semplicemente con v quando le traiettorie nello spazio degli stati coincidono per entrambi i casi.

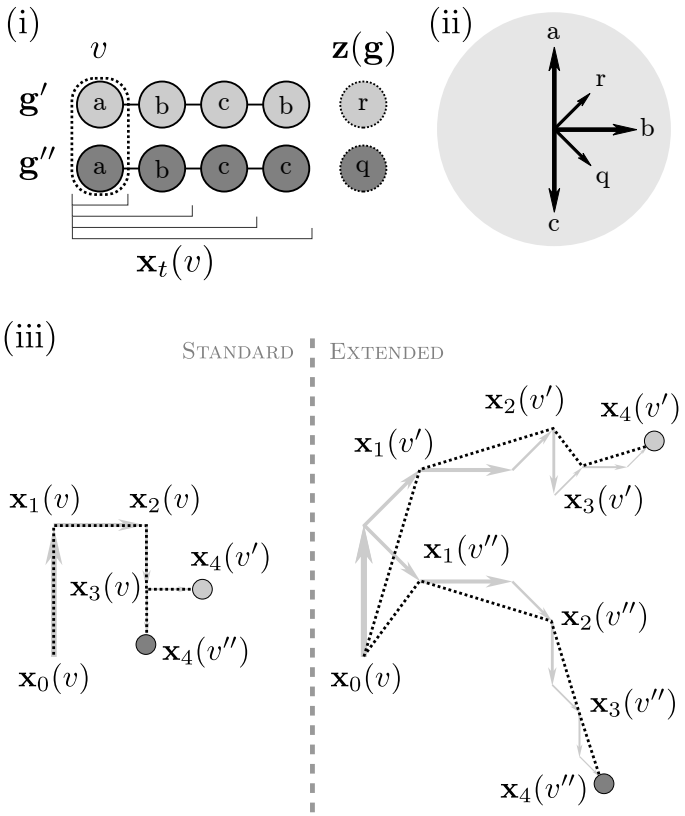


Figura 2.1: Effetto degli output-feedback sul reservoir esteso: esempio di encoding in uno spazio bidimensionale.
(i) Input; (ii) interpretazione geometrica degli input; (iii) traiettorie nel feature space, (*sx*) reservoir standard, (*dx*) reservoir esteso.

Nel caso di un un reservoir standard, a sinistra nella figura, i due input seguono la stessa traiettoria per le prime tre iterazioni. Le due sequenze sono dunque indistinguibili fino al quarto passo, ovvero fino a quando l'insieme dei vertici che determina lo stato $\mathbf{x}(v)$ è sufficientemente ampio da includere l'ultimo vertice. Si nota inoltre come, per effetto della natura Markoviana del reservoir, gli spostamenti delle variabili di stato si riducano con l'aumentare delle iterazioni di modo che l'ultima iterazione, quella determinante per discriminare $\mathbf{g}'$ e $\mathbf{g}''$ nel reservoir standard, mantenga comunque le due

codifiche $\mathbf{x}_4(v')$ e $\mathbf{x}_4(v'')$ molto vicine nello spazio degli stati.
Sulla destra sono invece mostrate le traiettorie nel caso in cui ai due input sia associato un output-feedback differente, tenuto in considerazione dal reservoir esteso nel corso del processo di encoding. La figura evidenzia come in questo caso le traiettorie si differenzino sin dalla prima iterazione, permettendo di discriminare le due sequenze ben prima che l'encoding di v' e v'' arrivi ad includere l'etichetta dell'ultimo vertice e con l'effetto di portare $\mathbf{x}_4(v')$ e $\mathbf{x}_4(v'')$ in punti molto più lontani all'interno del feature space rispetto a quanto avverrebbe con un reservoir standard.

Per effetto dello spostamento appena descritto, dunque, i pattern in input tenderanno a raggrupparsi nello spazio degli stati in maniera consistente con i propri valori degli output-feedback. È importante notare che, quando le varie sotto-reti sono allenate per riprodurre un segnale legato all'errore residuo commesso dalla rete (e.g. emulando l'errore o massimizzando la correlazione fra il proprio output e l'errore, si veda il paragrafo 1.2.1 a pagina 18), allora si avrà uno spostamento maggiore in quegli input per cui sia stato precedentemente riconosciuto un alto errore residuo, e che hanno dunque maggiore probabilità di essere stati corretti dalla rete nei passi precedenti (si veda anche il paragrafo 2.2.1). Informalmente potremmo dire che gli output-feedback, provenienti da altre unità computazionali, hanno in questo caso l'effetto di "mettere da parte" gli input i cui errori siano già stati corretti da altre sotto-reti, permettendo dunque al learning di concentrarsi maggiormente su una porzione di pattern per i quali la rete commette gli errori maggiori.

2.3 Modelli

Come detto in precedenza, i modelli sperimentati si distinguono nel modo in cui le varie sotto-reti vengono collegate fra loro.

Mantenendo il denominatore comune dell'approccio costruttivo, secondo i principi e le modalità già descritte, i tre modelli proposti si inseriscono in un percorso incrementale che guarda all'arricchimento della struttura nelle connessioni in modo da sfruttare l'introduzione di informazione supervisionata attraverso la presenza degli output-feedback.

GraphESN-CF

Il modello *GraphESN-CF* (i.e. GraphESN Constructive Flat) rappresenta l'estensione delle GraphESN al caso costruttivo: le singole sotto-reti vengono aggiunte incrementalmente ma non vi è alcuna connessione fra i loro output. Il modello è quindi formato da un unico strato di sotto-reti e dal readout globale, senza che sia presente alcun output-feedback.

La figura 2.2 nella pagina successiva mostra schematicamente la topologia del modello GraphESN-CF.

GraphESN-FW

Il modello *GraphESN-FW* (i.e. GraphESN Forward) arricchisce il precedente attraverso una struttura più complessa. In questo caso, infatti, il readout di ogni sotto-rete riceve in input gli output-feedback provenienti dalle sotto-reti precedenti. Il risultato è una struttura in cascata in cui, in maniera simile a quanto accade per la Cascade Correlation (si veda il paragrafo 1.2.1),

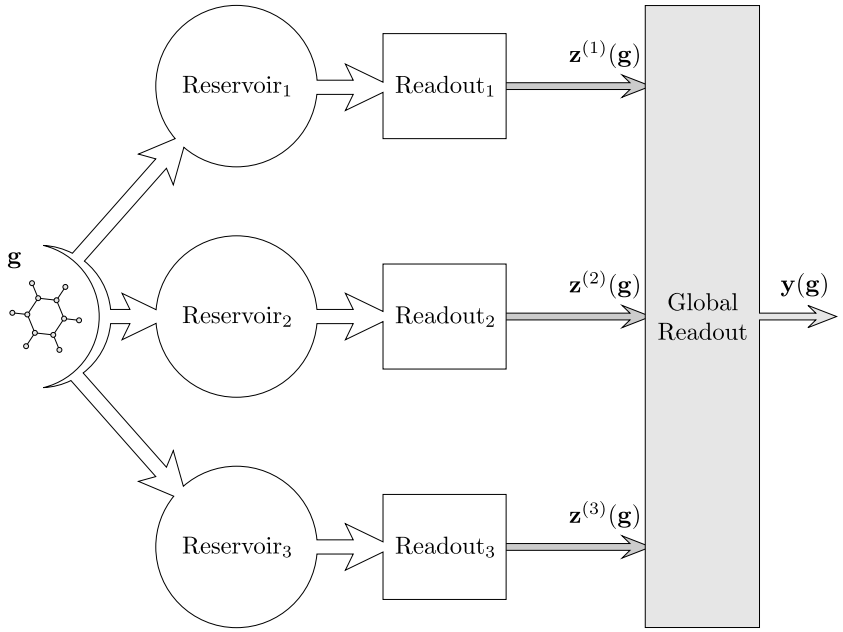


Figura 2.2: Modello GraphESN-CF. Topologia con tre sotto-reti.

ogni sotto-rete forma un livello a sé stante dipendente dai precedenti. Poiché gli output-feedback vanno da una sotto-rete a tutte le successive, le connessioni si sviluppano "in avanti".

La figura 2.3 a pagina 64 mostra la topologia del modello GraphESN-FW, con le connessioni in avanti.

Alcune osservazioni sono opportune riguardo alla presenza di output-feedback ed alla particolare scelta di una struttura a cascata in avanti. L'esistenza di output-feedback, infatti, determina implicitamente una relazione di dipendenza fra le varie unità computazionali (i.e. per poter svolgere la propria elaborazione, ogni sotto-rete necessita dell'output delle sotto-reti dalle quali riceve output-feedback). La struttura a cascata in avanti individua dunque un ordinamento parziale che mette il modello nelle condizioni di poter funzionare, determinando anche una strategia di esecuzione specifica (i.e. gli

output vengono calcolati in sequenza, a partire dalla prima sotto-rete in poi). Il fatto che gli output-feedback siano solo in avanti ha anche il vantaggio di mantenere inalterate tutte quelle sotto-reti per cui siano già state svolte delle elaborazioni, con l'effetto pratico di poter salvare i risultati ottenuti (e.g. l'encoding dei singoli input da parte dei reservoir) e risparmiare dunque risorse di calcolo.

GraphESN-FOF

L'ultimo modello proposto, *GraphESN-FOF* (i.e. GraphESN Forward Output-Feedback), e combina l'approccio incrementale, la struttura a cascata in avanti e l'uso degli output-feedback per spostare gli stati del reservoir come discusso nel paragrafo 2.2.2. In questo caso, dunque, le connessioni in avanti sono mandate sia al readout che al reservoir di tutte le sotto-reti successive.

La struttura delle connessioni del modello GraphESN-FOF è mostrata nella figura 2.4. La figura 2.5 a pagina 65 mostra invece i primi tre passi del processo incrementale di costruzione di una rete.

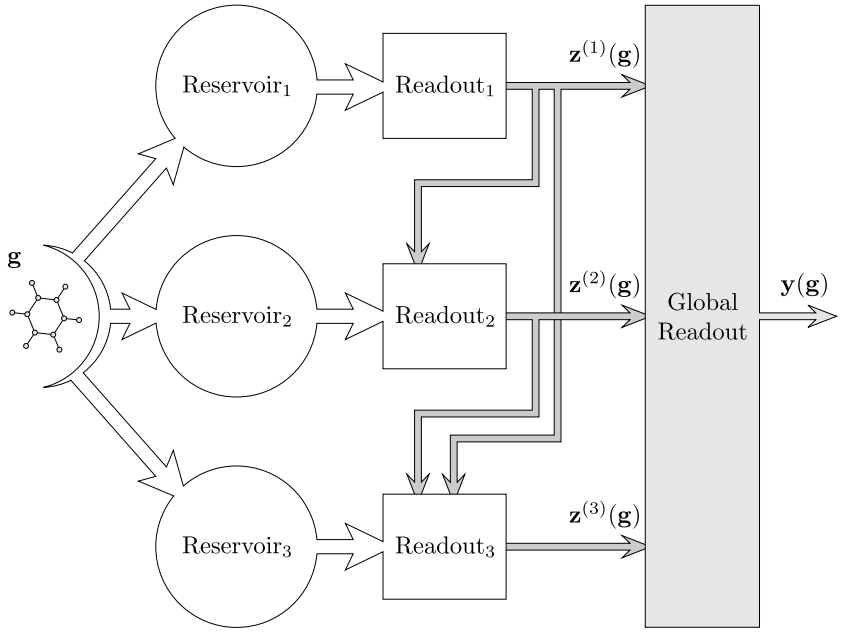


Figura 2.3: Modello GraphESN-FW. Topologia con tre sotto-reti.

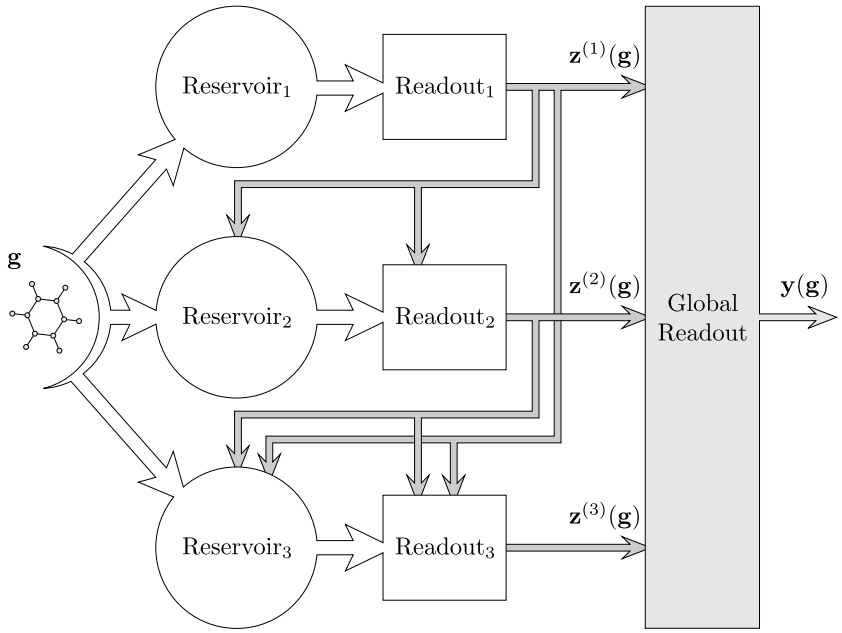
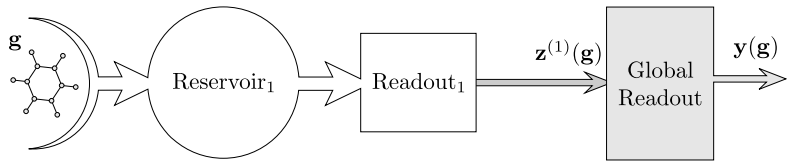
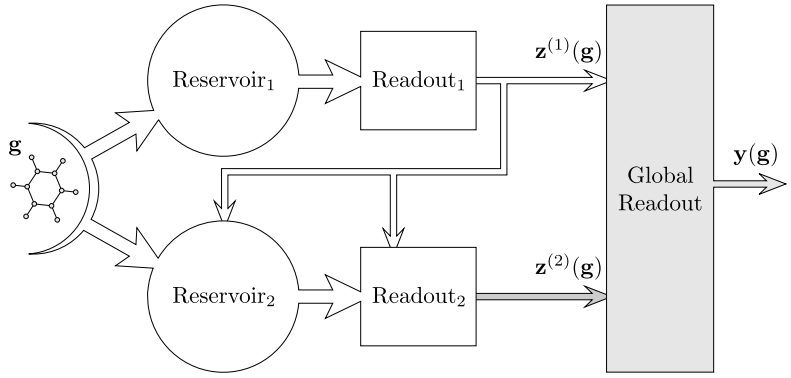


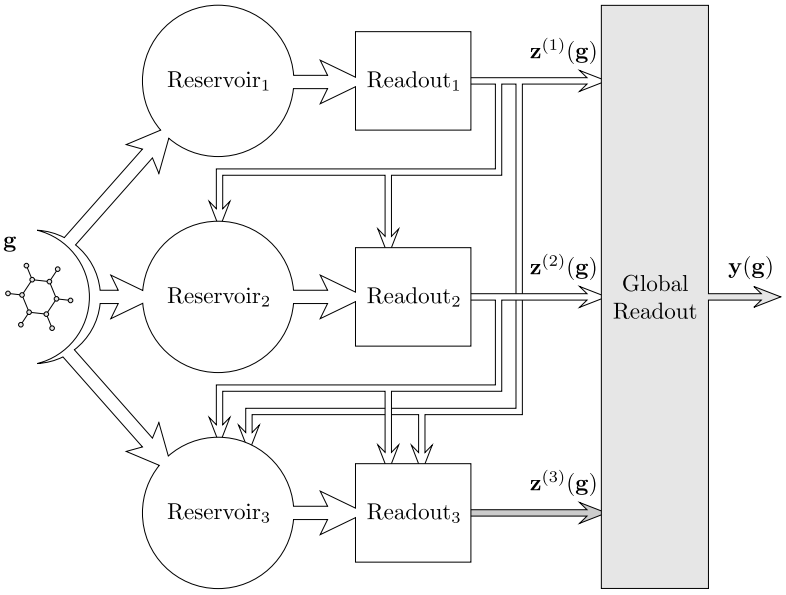
Figura 2.4: Modello GraphESN-FOF. Topologia con tre sotto-reti.



(a) Struttura della rete dopo una iterazione.



(b) Struttura della rete dopo due iterazioni.



(c) Struttura della rete dopo tre iterazioni.

Figura 2.5: Costruzione di una rete di tipo GraphESN-FOF. Dall'alto verso il basso, tre passi successivi di costruzione della rete. Ad ogni iterazione sono mostrate in grigio le connessioni soggette ad allenamento.

2.4 Costo computazionale

Consideriamo una rete di tipo GraphESN-FOF formata da *NSN* sotto-reti, ognuna con N_R unità nel reservoir. Assumiamo inoltre, secondo un'ipotesi tutt'altro che irrealistica, che la dimensione dell'output della rete e delle sotto-reti sia trascurabile (tipicamente $N_Y = N_Z = 1$). Di seguito viene riportato il costo computazionale delle fasi principali che caratterizzano l'evoluzione della rete (si veda l'algoritmo 2 a pagina 56). Il dettaglio del calcolo del costo computazionale è riportato nell'appendice A.

Encoding

Ogni sotto-rete deve far convergere il proprio reservoir per ogni input $\mathbf{g} \in \mathcal{G}$ e quindi ottenere l'encoding corrispondente nello spazio delle features. Grazie alle caratteristiche dei modelli sperimentati è sufficiente che ogni sotto-rete esegua questa fase una sola volta per ogni input.

Chiamando $MAXV$ il numero massimo di vertici di un grafo del dataset $\mathcal{G}$ (i.e. $MAXV = \max(\{|V(\mathbf{g})|, \mathbf{g} \in \mathcal{G}\})$ e $MAXIT$ il numero massimo di iterazioni consentite per la convergenza di un singolo reservoir, il costo computazionale complessivo del processo di encoding risulta essere

$$O(NSN\ MAXIT\ MAXV\ |\mathcal{G}|\ N_R^2) \tag{2.7}$$

Il costo totale della codifica degli input scala quindi quadraticamente rispetto alle dimensioni del reservoir. È opportuno tuttavia osservare che, nel caso dei modelli proposti, la possibilità di scomporre il task in sotto-task permette l'adozione di sotto-reti di dimensioni contenute.

Come nel caso delle GraphESN standard questa fase è identica sia per gli input usati nel training della rete che per il test.

Benché negli esperimenti riportati nel seguito (capitolo 3) siano stati utilizzati reservoir completamente connessi, sfruttando la presenza di sotto-reti di dimensioni contenute, il costo computazionale dell'encoding può essere ridotto assumendo una connettività sparsa nel reservoir. Nel caso in cui ogni unità del abbia al più M connessioni in ingresso, il costo dell'encoding diventa

$$O(NSN\ MAXIT\ MAXV\ |\mathcal{G}|\ N_R\ M) \tag{2.8}$$

e dipende dunque linearmente dalla dimensione dell'input e dei reservoir utilizzati.

La fase di encoding realizzata tramite i modelli proposti si caratterizza dunque, come nel caso delle GraphESN (si veda anche il paragrafo 1.5.2), per efficienza computazionale se confrontata con altri modelli per il trattamento di domini strutturati. Particolarmente interessante può essere in questo caso il confronto con tecniche basate su kernel: l'uso di kernel su grafi [14] comporta infatti un costo almeno quadratico rispetto al numero di vertici dell'input, e realizza un encoding non adattivo, secondo metriche prefissate. Con i modelli introdotti risulta invece possibile, ad un costo computazionale vantaggioso, realizzare la codifica dell'input tenendo conto anche dell'informazione supervisionata derivante dagli output-feedback.

I dettagli del calcolo per il costo computazionale dell'encoding sono riportati nel paragrafo A.1.

Training

Distinguiamo due distinte fasi di training, in accordo con la strategia costruttiva descritta in precedenza: il training di una singola sotto-rete e l'allenamento del readout-globale. Poiché il training riguarda in ogni caso l'adattamento dei pesi di un unico livello di connessioni, verso il readout, vari algoritmi possono essere efficacemente applicati. Nel seguito saranno considerate solo le strategie utilizzate nell'analisi sperimentale dei modelli.

Una sotto-rete può essere allenata per emulare l'errore commesso dal readout-globale. In questo caso, ricorrendo all'algoritmo di Ridge Regression (si veda il paragrafo 1.1.3) il costo computazionale complessivo è

$$O(NSN^4 + N_R\ NSN^3 + N_R^2\ NSN^2 + N_R^3\ NSN + NSN^3\ |\mathcal{G}| + NSN^2\ N_R\ |\mathcal{G}| + NSN\ N_R^2\ |\mathcal{G}|) \quad (2.9)$$

e scala dunque con il numero di sotto-reti elevato alla quarta e con il cubo della dimensione del reservoir, secondo una dipendenza critica nel determinare il vantaggio computazionale, come verrà discusso nel seguito.

Il readout-globale della rete può a sua volta essere allenato tramite Ridge Regression, con un costo complessivo

$$O(NSN^4 + NSN^3|\mathcal{G}|) \quad (2.10)$$

È tuttavia possibile allenare il readout-globale con Least Mean Squares (si veda il paragrafo 1.1.3). Sia $LMSIT$ il numero massimo di iterazioni necessarie

alla convergenza dell'algoritmo, il costo complessivo è

$$O(LMSIT\ |\mathcal{G}|\ NSN^2\ ) \tag{2.11}$$

In questo caso la dipendenza è quadratica rispetto al numero complessivo di sotto-reti.

Il calcolo completo del costo computazionale dell'allenamento delle sotto-reti tramite Ridge Regression è riportato nel paragrafo A.2, mentre i paragrafi A.3.1 e A.3.2 riportano il calcolo del costo per l'allenamento del readout globale tramite Ridge Regression e LMS rispettivamente.

Calcolo dell'output

Ogni volta che una nuova sotto-rete viene allenata ed aggiunta alla rete è necessario procedere con il calcolo dell'output relativo ad ognuno dei grafi nel dataset $\mathcal{G}$. Nell'intero processo di costruzione della rete il costo del calcolo dell'output, a livello delle sotto-reti, risulta

$$O(NSN\ N_R\ |\mathcal{G}| + NSN^2\ |\mathcal{G}|) \tag{2.12}$$

presentando una dipendenza quadratica rispetto al numero di sotto-reti e lineare rispetto alla dimensione del dataset e del reservoir. È opportuno in questo caso sottolineare come, per una generica sotto-rete, il calcolo dell'uscita dipenda dagli output-feedback delle sotto-reti precedenti, i cui valori sono precalcolati, e dal risultato della fase di encoding, anch'esso già calcolato durante la fase di allenamento.

Ad ogni incremento delle dimensioni della rete è inoltre necessario ottenere l'errore commesso dal readout-globale e quindi calcolare l'output della rete. Poiché il readout-globale prende in input unicamente gli output delle singole sotto-reti il costo complessivo è

$$O(NSN^2\ |\mathcal{G}|) \tag{2.13}$$

e scala linearmente con la dimensione del dataset e quadraticamente rispetto al numero di sotto-reti.

Il dettaglio del calcolo del costo computazionale per ottenere l'output delle sotto-reti e del readout globale è riportato nei paragrafi A.4 e A.5 rispettivamente.

Costo complessivo

In riferimento ad una GraphESN-FOF, con reservoir completamente connesso, che usi l'algoritmo Ridge Regression per l'apprendimento, è possibile riassumere il costo computazionale delle singole fasi come indicato nella tabella 2.1.

Tabella 2.1: Costo computazionale di GraphESN-FOF

Encoding	$O(NSN\ MAXIT\ MAXV\ \|\mathcal{G}\|\ N_R^2)$
Learning (SubNets)	$O(NSN^4 + N_R\ NSN^3 + N_R^2\ NSN^2 + N_R^3\ NSN$ $+NSN^3\ \|\mathcal{G}\| + NSN^2\ N_R\ \|\mathcal{G}\| + NSN\ N_R^2\ \|\mathcal{G}\|)$
Learning (Global)	$O(NSN^4 + NSN^3\|\mathcal{G}\|)$
Output (SubNets)	$O(NSN\ N_R\ \|\mathcal{G}\| + NSN^2\ \|\mathcal{G}\|)$
Output (Global)	$O(NSN^2\ \|\mathcal{G}\|)$

2.4.1 Considerazioni

I modelli di Reservoir Computing sono caratterizzati, in generale, per l'efficienza computazionale. La possibilità di limitare l'allenamento alle sole unità del readout, facendo affidamento sulle caratteristiche strutturali del reservoir per realizzare l'encoding, permette infatti l'impiego di algoritmi di apprendimento semplici e computazionalmente poco onerosi. Per poter efficacemente riflettere sul costo computazionale dei modelli proposti è dunque opportuno guardare a questo ambito, prendendo come riferimento il confronto con la GraphESN standard (si veda il paragrafo 1.5.2), che i modelli estendono.

Per le varie fasi di utilizzo di una GraphESN, assumendo anche in questo caso l'adozione di reservoir completamente connessi e formati da un numero totale di unità pari a quelle di tutte le sotto-reti di una GraphESN-FOF (i.e. $N'_R = N_R\, NSN$), possiamo considerare il costo computazionale riportato nella tabella 2.2.

Tabella 2.2: Costo computazionale di una GraphESN con $N'_R = N_R\, NSN$.

Encoding	$O((N_R\, NSN)^2\ MAXIT\ MAXV\ \lvert\mathcal{G}\rvert)$
Learning	$O((N_R\, NSN)^3 + (N_R\, NSN)^2\ \lvert\mathcal{G}\rvert)$
Output	$O(\lvert\mathcal{G}\rvert\ N_R\ NSN)$

Le differenze principali emergono, confrontando il costo delle GraphESN con quello dei modelli costruttivi, osservando la dipendenza del costo computazionale rispetto alle dimensioni del reservoir. La dipendenza quadratica del processo di encoding delle GraphESN rispetto a NSN diventa infatti lineare con l'adozione della strategia costruttiva. Il costo dell'apprendimento, che nelle GraphESN dipende dal cubo del prodotto di NSN ed N_R, è a sua

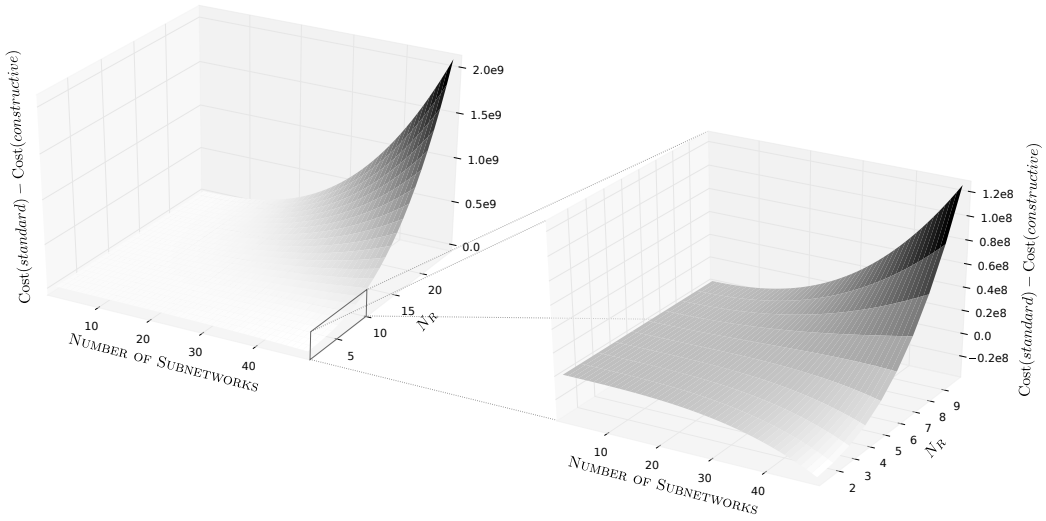


Figura 2.6: Vantaggio computazionale nell'uso dell'approccio costruttivo. Assi orizzontali: numero di sotto-reti e dimensioni del reservoir (N_R). Asse verticale: differenza fra costo del modello standard e costo del modello costruttivo.
(*sx*) $N_R \in [1, 25]$. (*dx*) Dettaglio della regione con minore vantaggio computazionale: $N_R \in [1, 10]$.

volta ridotto di due ordini di grandezza rispetto a *NSN*. Di contro, sia per l'apprendimento che per il calcolo dell'output, le reti costruttive presentano un costo fortemente dipendente dal numero di sotto-reti, che determina uno svantaggio nel caso in cui il valore di *NSN* domini sugli altri.

La figura 2.6 dà un'intuizione di quale sia il vantaggio dal punto di vista computazionale derivato dall'adozione dell'approccio costruttivo. A parità di dimensioni del dataset (i.e. $|\mathcal{G}| = 100$) ed al variare delle dimensioni dei reservoir e del numero delle sotto-reti, la figura mostra come varia la differenza fra il costo di una GraphESN standard equivalente (i.e. con reservoir di dimensione $N'_R = N_R\, NSN$) e di una GraphESN-FOF. I valori positivi corrispondono dunque ai casi in cui l'approccio costruttivo risulta più efficiente.

La figura evidenzia come con il crescere delle dimensioni dei reservoir ed all'aumentare del numero di sotto-reti, l'approccio costruttivo risulti in un vantaggio computazionale notevole. La possibilità di scomporre un problema in sotto-problemi si rispecchia infatti nella struttura complessiva della rete, che impiega, rispetto al caso delle GraphESN, più reservoir di dimensioni ridotte, il cui allenamento richiede un impiego minore di risorse di calcolo. A destra nella figura è evidenziata tuttavia l'esistenza di particolari condizioni per le quali l'impiego di una strategia costruttiva risulta essere sconveniente. Tale situazione si verifica in particolare nel caso in cui si usino molte sotto-reti di dimensioni estremamente ridotte. La motivazione risiede nel fatto che, in un simile scenario, il costo computazionale complessivo risulta dominato dal processo iterativo di costruzione della rete piuttosto che dalla dimensione dei reservoir trattati. In merito a quanto detto è tuttavia opportuno osservare che, per la natura stessa del Reservoir Computing, l'impiego di reservoir con un numero davvero esiguo di unità è da considerarsi poco realistico.

Un'ulteriore osservazione risulta particolarmente importante per valutare i modelli proposti dal punto di vista computazionale. Benché il confronto con una singola GraphESN lasci emergere i vantaggi dei modelli proposti, è infatti opportuno sottolineare come la strategia costruttiva comporti particolari benefici anche in termini di selezione del modello. Per determinare la corretta dimensione della rete, infatti, i modelli non costruttivi si affidano ad un approccio *trial and error* in cui ogni possibile variazione nella dimensione del reservoir viene allenata e valutata in maniera indipendente. Al contrario, l'approccio costruttivo permette di determinare la dimensione della rete in maniera automatica ed offre quindi l'opportunità di ridurre di molto il numero

di esperimenti necessari a determinare quali siano gli iperparametri più adatti per la risoluzione di un task.
Intuitivamente possiamo dunque dire che il costo speso per la costruzione incrementale di una singola rete costruttiva, corrisponda all'allenamento ed il test di svariate reti diverse nel caso delle GraphESN. Per fare un esempio possiamo pensare che l'allenamento di una GraphESN-FOF, formata da NSN sotto-reti di dimensione N_R, corrisponda, in termini computazionali, all'allenamento di NSN diverse GraphESN, con reservoir di dimensioni crescenti: $N'_R \in \{N_R, 2N_R, \ldots, NSN\ N_R\}$. Questo porta ad avere un costo complessivo per la selezione di una GraphESN pari a (si veda il paragrafo A.6)

$$O(NSN^4\ N_R^3 + NSN^3\ N_R^2\ MAXIT\ MAXV\ |\mathcal{G}| + NSN^3\ N_R^2\ |\mathcal{G}| + NSN^2\ N_R\ |\mathcal{G}|) \quad (2.14)$$

che scala con il prodotto fra N_R^3 e NSN^4 determinando una dipendenza estremamente svantaggiosa rispetto al caso dell'allenamento di una singola GraphESN-FOF, in cui i due fattori compaiono congiunti in una dipendenza al più quadratica.

Quanto detto evidenzia dunque come i modelli proposti si caratterizzino per efficienza dal punto di vista computazionale. La strategia costruttiva introdotta permette infatti di ridurre gli oneri del processo di apprendimento rispetto alle GraphESN, già di per sé contraddistinte dall'efficienza computazionale, rispondendo ad un'esigenza molto rilevante nell'ambito del trattamento dei domini strutturati. È infine opportuno sottolineare come l'utilizzo di uno schema di output-feedback permetta, nei modelli proposti, di

influenzare il processo di encoding in maniera supervisionata senza che questo comporti la perdita, in termini computazionali, dei vantaggi derivati dall'uso di un reservoir non adattivo.

2.5 Software

Un'implementazione software dei modelli descritti nel corso del capitolo è stata realizzata utilizzando il linguaggio `Python`[3] (versione 2.6).

Essendo un linguaggio dinamico, dalla sintassi semplice e fornito di un'ampia libreria standard, `Python` risulta particolarmente efficace per la prototipazione rapida. Questa caratteristica è stata considerata particolarmente desiderabile ai fini della realizzazione del lavoro svolto. Lo sviluppo dei modelli, così come quello del software, è infatti stato portato avanti in maniera incrementale, secondo un percorso guidato anche dall'esito di prove empiriche e sperimentali.

La scelta del linguaggio è risultata inoltre soddisfacente in termini di performance, nonostante il fatto che `Python` sia un linguaggio interpretato. L'utilizzo della libreria `scipy`[4] per l'algebra lineare ha infatti permesso di raggiungere prestazioni comparabili con quelle ottenibili realizzando gli stessi modelli in `MATLAB`[5], il cui utilizzo è ampiamente diffuso nell'ambito della realizzazione di modelli neurali.

Il software sviluppato sarà rilasciato con licenza open source.

[3] `http://python.org/`
[4] `http://www.scipy.org/`
[5] `http://www.mathworks.com/products/matlab/`

3 Risultati Sperimentali

Il capitolo illustra e discute i risultati sperimentali ottenuti applicando i modelli proposti su task relativi a dataset reali.

Il paragrafo 3.1 descrive i dataset utilizzati ed i corrispondenti task affrontati.

Il paragrafo 3.2 descrive i setting sperimentali e raccoglie i risultati ottenuti.

Il paragrafo 3.3 espone un'analisi critica dei risultati sperimentali.

3.1 Dataset

Le potenzialità dei modelli proposti sono state sperimentate su problemi reali appartenenti all'ambito della Chimica.

Pur senza addentrarsi nei dettagli è interessante sottolineare come il legame fra il Machine Learning e la Chimica vada recentemente rafforzandosi, in particolar modo in relazione all'apprendimento di dati su domini strutturati. L'enorme varietà di molecole naturalmente rappresentabili come grafi, i costi sperimentali elevati, l'esistenza di conoscenze non sempre codificabili o prone

a numerose eccezioni, fanno infatti della Chimica il banco di prova ideale per dei modelli in grado di gestire domini strutturati.

È dunque alla sfera della Chemioinformatica che vanno ascritti i task affrontati ed i dataset che verranno descritti nel seguito del paragrafo.

Prima di procedere, è importante sottolineare che nessuna specifica conoscenza pregressa è stata necessaria per preprocessare i dati, modificarne la struttura o estrapolarne informazioni numeriche. In altri termini, le informazioni utilizzate per allenare i modelli sono esattamente quelle riportate nei dataset: diversamente da quanto accade ricorrendo ad altri approcci, nessuna particolare assunzione riguardante il dominio trattato è stata fatta (e.g. la scelta di una metrica nell'uso di un kernel) né è stata praticata alcuna forma di feature-selection manuale (e.g. la selezione di indici topologici).

Predictive Toxicology Challenge

Il *Predictive Toxicology Challenge* (PTC) [26] ha fornito i dati per quattro task distinti. Il dataset consta di 417 molecole in formato *Structure Data Format* (SDF) di cui è riporta la carcinogenicità riferita a diversi tipi di roditori: topi maschi (MM), topi femmine (FM), ratti maschi (MR), ratti femmine (FR).

Ogni molecola è stata rappresentata come un grafo indiretto, con i vertici corrispondenti agli atomi e gli archi ai legami atomici. Il grado massimo riscontrato sull'intero dataset è $k = 4$. Il numero di vertici nei grafi in input risulta mediamente 25.7, variando da un minimo di 2 ad un massimo di 109.

L'etichetta numerica associata ad ogni vertice è stata ottenuta codificando il simbolo atomico corrispondente tramite un encoding binario 1-of-m e

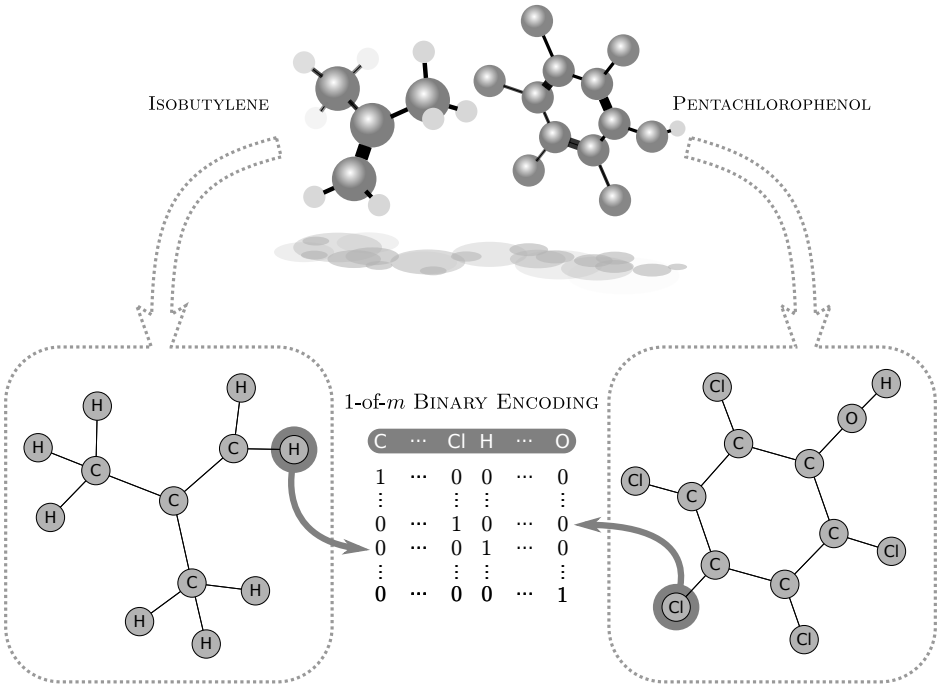


Figura 3.1: Rappresentazione dell'input in forma di grafo ed encoding del simbolo atomico.

concatenando due valori aggiuntivi presenti nel dataset, CHG e RAD, rispettivamente corrispondenti alla carica atomica ed il radical dell'atomo. La dimensione totale delle etichette di input è dunque 24. La figura 3.1 mostra l'esempio due molecole appartenenti al dataset PTC, rappresentate come grafi indiretti, e schematizza il processo di encoding del simbolo atomico.

Quattro task structure-to-element di classificazione binaria sono stati definiti per PTC, uno per ogni tipo di roditore [13], assegnando target $+1$ alle molecole attive e target -1 a quelle inattive.

Mutagenesis

La seconda serie di task affrontati si riferisce al dataset *Mutagenesis* [55] (Mutag), contenente 230 molecole nitroaromatiche in formato Progol[1], di cui è indicata la mutagenicità. Ogni composto nel dataset è descritto dalla sua struttura atomo-legame (AB), da due valori corrispondenti a misurazioni di proprietà chimiche della molecola (C) e due attributi strutturali precalcolati (PS). Riprendendo quanto fatto in [15], sono state dunque considerate le tre possibili descrizioni: AB, AB+C, AB+C+PS.

Ogni molecola del dataset è stata rappresentata come un grafo indiretto, con vertici ed archi corrispondenti ad atomi e legami rispettivamente, entrambi riportati nella descrizione AB dei singoli composti. Il grado massimo riscontrato sul dataset risulta essere $k = 4$. Il numero di vertici per input varia tra un minimo di 13 ad un massimo di 40, con una media di 25.6 vertici per ogni molecola.

L'etichetta numerica associata ad ogni vertice dell'input è stata ricavata codificando il simbolo atomico corrispondente attraverso un encoding binario 1-of-m e poi concatenando la carica parziale dell'atomo, il tipo atomico normalizzato in $[-1, 1]$ più gli altri valori globali contenuti nella descrizione C e PS, in accordo alla rappresentazione via via adottata. La dimensione delle etichette di input è dunque di 11, 13 e 15 per le descrizioni AB, AB+C e AB+C+PS rispettivamente.

Il task sul dataset è stato definito come una classificazione binaria, con il target a $+1$ per i grafi corrispondenti a composti mutageni e -1 altrimenti.

[1] `http://www.doc.ic.ac.uk/~shm/progol.html`

Bursi

Il dataset *Bursi* [35, 11] contiene 4204 molecole in formato SDF, di cui è riportata la mutagenicità. Ne risulta un task structure-to-element di classificazione binaria. Il dataset risulta all'origine suddiviso partizionato in un training-set, contenente 3367 molecole, ed un test-set con 837 composti.

Anche in questo caso le molecole sono state rappresentate come grafi indiretti, con vertici ed archi corrispondenti ad atomi e legami rispettivamente. Nell'input il numero di vertici varia tra un minimo di 4 ed un massimo di 417, con una media di 30.3 vertici per molecola.

L'etichetta numerica associata ai vertici è realizzata attraverso l'encoding binario 1-of-m del simbolo atomico. Le etichette hanno dunque dimensione 14. Il grado massimo riscontato sul dataset è $k = 4$.

Il target per il task di classificazione binaria è stato realizzato assegnando valore $+1$ alle molecole attive e -1 alle molecole non attive.

Angiotensin Converting Enzyme

Il dataset *Angiotensin Converting Enzyme* (ACE) [57] contiene 114 ACE-inibitori in formato SDF a cui è associato un valore reale che ne indica l'attività (i.e. plC_{50}) in un intervallo che varia tra 2.1 a 9.9. Il dataset definisce dunque un task structure-to-element di regressione di tipo *Quantitative Structure–Activity Relationship* (QSAR).

Come nei casi precedenti, le molecole sono state rappresentate come grafi indiretti, con i vertici corrispondenti agli atomi e gli archi corrispondenti ai

legami. Il grado massimo riscontrato sul dataset è $k = 4$, le dimensioni dei grafi variano tra 18 e 79, con una media di 42.5 vertici per input.

I valori dell'etichetta associata ad ogni vertice dell'input sono ricavati concatenando l'encoding binario del simbolo atomico al valore della carica atomica, CHG, presente nel dataset. Le etichette di input hanno dunque dimensione 8.

3.2 Esperimenti

Sui dataset descritti in precedenza sono stati svolti esperimenti che permettessero di confrontare i tre modelli proposti (si veda il paragrafo 2.3) con le GraphESN (paragrafo 1.5.2).

È bene precisare che, data la flessibilità dell'approccio introdotto, sono molte le diverse strategie applicabili per realizzare un singolo modello fra quelli descritti. Variazioni sono possibili nell'inizializzazione dei pesi, nella scelta delle candidate, nella tipologia di allenamento delle sotto-reti o nel task loro assegnato. Tutte le reti a cui fanno riferimento i risultati riportati nel seguito condividono dunque un setting sperimentale comune, che individua solo un piccolo frammento delle possibili varianti implementative. Prima di presentare i risultati sperimentali è quindi necessario descrivere il setting sperimentale adottato.

Come appartenenti all'ambito del Reservoir Computing, le reti considerate presentano una serie di connessioni i cui pesi non vengono modificati durante il learning: le connessioni input-reservoir e reservoir-reservoir, a cui si aggiungono, nei modelli proposti, anche le connessioni per gli output-feedback.

Nei modelli sperimentati i pesi sulle connessioni di input, contenuti nella matrice $\mathbf{W}_{\text{in}}^{(i)}$ per la sotto-rete i-esima, hanno valori random con distribuzione uniforme nell'intervallo $[-\lambda_{\text{in}}, \lambda_{\text{in}}]$, con λ_{in} parte degli iperparametri del modello. I pesi sulle connessioni del reservoir, matrice $\hat{\mathbf{W}}^{(i)}$, sono invece stati inizializzati secondo una distribuzione uniforme in $[-1, 1]$ e poi ridimensionati per ottenere il coefficiente di contrazione σ desiderato (si veda l'equazione (1.29) a pagina 40). In tutti gli esperimenti svolti si è ricorso a reservoir completamente connessi, sfruttando la possibilità di impiegare sotto-reti di dimensioni contenute, che non richiedessero eccessivi oneri computazionali. Alle connessioni di output-feedback, in $\mathbf{W}_{\text{fof}}^{(ij)}$ per ogni coppia di sotto-reti (i, j), sono stati assegnati pesi fissi con valore dato dall'iperparametro λ_{fof}.

Poiché tutti i dataset trattati riguardano trasduzioni structure-to-element, i modelli sperimentati ricorrono all'uso di una *state mapping function*, $\mathcal{X}$. In tutti i casi è stata utilizzata una funzione *mean state mapping* (si veda l'equazione (1.31) a pagina 43).

I modelli costruttivi proposti sono caratterizzati da due distinte fasi di apprendimento per ogni iterazione: una riguardante le singole sotto-reti e l'altra relativa al readout globale (si veda l'algoritmo 2 a pagina 56).
Nel corso degli esperimenti svolti le sotto-reti sono state allenate attraverso l'algoritmo di Ridge Regression (si veda il paragrafo 1.1.3) per emulare l'errore residuo commesso dalla rete. Il parametro di regolarizzazione λ_{r} utilizzato, uguale per ogni sotto-rete, rappresenta uno degli iperparametri dei modelli. La possibilità di usare un algoritmo di apprendimento che non dipendesse dal valore iniziale dei pesi e che non non rischiasse di convergere a minimi o massimi locali ha inoltre fatto propendere per tralasciare l'impiego di un pool

di sotto-reti candidate.

Il readout globale è invece stato allenato tramite Least Mean Squares (LMS) con l'iperparametro λ_{wd} a regolare il *weight decay* (si veda il paragrafo 1.1.3).

Per tutte le unità, sia nei reservoir che nei readout, la funzione di attivazione utilizzata è la *tangente iperbolica* (tanh). L'errore residuo, utilizzato per l'allenamento delle sotto-reti, è dunque quello ottenuto dalla differenza fra l'attivazione delle unità di output ed il valore del target

$$\begin{aligned} \mathbf{e}^{(i)}(\mathbf{g}) &= \mathbf{y}^{(i)}(\mathbf{g}) - \mathbf{y}_{\text{target}}(\mathbf{g}) \\ &= \tanh(\mathbf{W}_{\text{out}}\,[\mathbf{z}^{(1)}(\mathbf{g}), \ldots, \mathbf{z}^{(i)}(\mathbf{g})]) - \mathbf{y}_{\text{target}}(\mathbf{g}) \end{aligned} \tag{3.1}$$

Ne risulta un errore distribuito nell'intervallo $(-2, 2)$. Due fattori influenzano la scelta di adottare tale criterio nel determinare l'errore, e dunque il target delle sotto-reti: da una parte la pratica ha lasciato emergere una minore efficacia dell'uso di errori discreti (e.g. $\mathbf{e}^{(i)}(\mathbf{g}) \in \{-1, 1\}$ oppure $\mathbf{e}^{(i)}(\mathbf{g}) \in \{-1, 0, 1\}$), dall'altra la necessità di usare un'arcotangente iperbolica, $\tanh^{-1}$, per per poter applicare la Ridge Regression nelle sotto-reti ha reso indispensabile la presenza di un limite, sia superiore che inferiore, sui valori di errore. Per essere efficacemente usato come target per le sotto-reti, dunque, il segnale di errore è stato ad ogni iterazione dimezzato e portato nell'intervallo $(-1, 1)$.

Nel corso degli esperimenti svolti le reti sono state fatte crescere finché non si sia verificata una delle seguenti condizioni: l'errore sul training-set (i.e. misclassification-rate o errore quadratico medio), $\mathbf{e}^{(i)}(\mathcal{G}_{\text{tr}})$, sia risultato inferiore ad una soglia prefissata, δ_{err}, specifica per ogni task, oppure la variazione dell'errore di training sia risultata, in due iterazioni successive,

inferiore in valore assoluto ad un parametro δ_{var}. È inoltre stato fissato un numero massimo di sotto-reti, δ_{size}. In formula, il processo di costruzione della rete viene dunque interrotto quando si verifica la seguente condizione

$$\mathbf{e}^{(i)}(\mathcal{G}_{\text{tr}}) < \delta_{\text{err}} \ \vee \ \left| \frac{\mathbf{e}^{(i)}(\mathcal{G}_{\text{tr}}) - \mathbf{e}^{(i-1)}(\mathcal{G}_{\text{tr}})}{\mathbf{e}^{(i)}(\mathcal{G}_{\text{tr}})} \right| < \delta_{\text{var}} \ \vee \ i = \delta_{\text{size}}$$

Per confrontare i modelli introdotti con le GraphESN, son stati svolti su queste ultime esperimenti che prevedessero l'impiego di iperparametri simili. L'allenamento delle GraphESN è stato realizzato tramite Ridge Regression e, per ovviare al fatto che in questo caso il numero di unità totali dovesse essere fissato a priori, sono state prese in considerazione diverse GraphESN facendo variare la dimensione del reservoir.

3.2.1 Predictive Toxicology Challenge

La valutazione dei modelli sui quattro task PTC è stata fatta attraverso una *doppia k-fold cross-validation* (si veda il paragrafo 1.1.2) con un ciclo esterno di 5 fold ed un ciclo interno di 5 fold. La suddivisione del dataset è stata realizzata attraverso *stratificazione* (si veda il paragrafo 1.1.2), in modo da garantire la stessa distribuzione di esempi positivi e negativi in ogni sotto-insieme. Ogni iperparametrizzazione è stata testata su 5 istanziazioni del modello, in modo da variare i valori dei pesi random assegnati alle connessioni.

Per ognuno dei modelli proposti sono state considerate due distinte configurazioni per la dimensione del reservoir delle sotto-reti: $N_R \in \{50, 30\}$. Il task structure-to-element di classificazione binaria prevede una dimensione delle etichette di input $N_U = 24$ e un singolo valore di uscita $N_Y = N_Z = 1$.

Tabella 3.1: Iperparametri usati per la model selection sui 4 task PTC.

λ_{in}	λ_{fof}	λ_{r}	λ_{wd}
$\{1.0, 0.1\}$	$\{1.0, 2.0\}$	$\{0.01, 0.1, 0.2\}$	$\{0.0, 0.01\}$

Tabella 3.2: Iperparametri per la model selection di GraphESN sui 4 task PTC.

N_R	λ_{in}	λ_{r}
$\{500, 200, 100, 50\}$	$\{1.0, 0.1\}$	$\{0.01, 0.1, 0.2\}$

Per la convergenza del reservoir è stata adottata una soglia $\epsilon = 10^{-5}$. Il coefficiente di contrazione utilizzato è $\sigma = 1$.

Per interrompere la costruzione delle reti si è fissato $\delta_{\text{var}} = 0.01$ e δ_{tr}, riferito al miscalssification-rate (i.e. rapporto fra input non classificati correttamente ed input totali), uguale a 0.29, 0.33, 0.37, 0.29 per i task FR, FM, MR, MM rispettivamente. Il numero massimo di sotto-reti è stato fissato a $\delta_{\text{size}} = 15$ nei casi in cui $N_R = 50$ e $\delta_{\text{size}} = 20$ con $N_R = 30$.

Nell'allenamento del readout globale tramite LMS è stato utilizzato un learning-rate $\eta = 10^{-3}$.

La model selection ha coinvolto gli iperparametri λ_{in}, λ_{fof}, λ_{r}, λ_{wd}, fatti variare secondo i valori riportati nella tabella 3.1. Per realizzare la selezione del modello e la validazione dei risultati sono dunque state allenate 24200 differenti istanziazioni di modelli costruttivi.

Sulle GraphESN la selezione del modello è stata eseguita facendo variare i valori di N_R, λ_{in}, λ_{r} come indicato nella tabella 3.2.

La tabella 3.3 descrive i risultati sperimentali ottenuti dai modelli sui quattro task PTC. I valori riportati rappresentano l'accuratezza percentuale di test (i.e. percentuale di grafi nel test-set classificata correttamente) mediata

Tabella 3.3: Accuratezza media dei modelli e deviazione standard, in percentuale, sui 4 task PTC.

Model	N_R	FR	FM	MR	MM
GraphESN		67.7 (±0.1)	60.7 (±0.4)	56.7 (±0.9)	67.1 (±0.1)
GraphESN-CF	50	67.3 (±0.6)	62.8 (±0.8)	57.9 (±0.7)	65.0 (±0.5)
GraphESN-CF	30	67.2 (±0.7)	63.2 (±0.8)	58.1 (±0.4)	65.8 (±0.7)
GraphESN-FW	50	67.1 (±1.0)	63.6 (±0.7)	58.4 (±0.7)	65.4 (±1.6)
GraphESN-FW	30	67.2 (±1.1)	63.3 (±1.0)	57.4 (±1.5)	64.6 (±1.5)
GraphESN-FOF	50	68.3 (±1.1)	62.5 (±1.2)	57.2 (±1.3)	66.6 (±1.7)
GraphESN-FOF	30	67.9 (±1.5)	62.8 (±1.6)	57.4 (±1.7)	65.4 (±1.6)

Tabella 3.4: Numero di sotto-reti dei modelli sui quattro task PTC. Numero massimo, numero minimo, media e moda.

Model	Max	Min	Avg	Mode
GraphESN-CF	15	2	3.9	3
GraphESN-FW	20	2	5.2	3
GraphESN-FOF	20	2	4.6	3

sulle 5 fold e la deviazione standard media. Le tabelle B.1-B.12 (si veda l'appendice B a pagina 124) riportano i risultati in dettaglio, fold per fold, dei vari modelli. Dai dati si riscontra come i modelli introdotti risultino in grado, in tre dei quattro casi, di migliorare la performance predittiva ottenuta tramite un approccio non costruttivo. In due dei quattro casi esaminati (i.e. FM, FR) il miglioramento offerto dai modelli costruttivi risulta sistematico e consistente se si considera che il dataset è particolarmente affetto da rumore.

La tabella 3.9 riporta i dati relativi alle dimensioni raggiunte delle reti nell'affrontare i quattro task del dataset. Si nota come in alcuni casi le reti crescano abbastanza da raggiungere la dimensione massima imposta δ_{size}, suggerendo l'eventuale adozione di una soglia meno vincolante.

La figura 3.2 a pagina 88 mostra alcuni esempi di curve di apprendimento

Tabella 3.5: Accuratezza media dei modelli e deviazione standard, in percentuale, di metodi basati su kernel applicati ai 4 task PTC.

Method	FR	FM	MR	MM
MG-Kernel	70 (±1)	65 (±1)	63 (±1)	69 (±1)
OA-Kernel	70 (±1)	65 (±1)	63 (±1)	68 (±1)
EM-Kernel	69 (±1)	65 (±1)	61 (±2)	67 (±1)

riferite all'applicazione su PTC del più generale dei modelli proposti (i.e. GraphESN-FOF). L'errore riportato è il misclassification-rate, motivo per cui le curve di test, calcolate su pochi dati, risultano avere un andamento caratterizzato da forti pendenze. Come mostrato in figura, il processo incrementale di costruzione della rete risulta efficace nell'aumentare la capacità di generalizzazione della rete, ma non evita completamente il verificarsi di situazioni di overfitting (grafico al centro).

L'accuratezza di predizione ottenuta tramite i modelli proposti risulta inoltre comparabile con quelle raggiunta applicando sugli stessi task dei modelli basati su kernel allo stato dell'arte nel trattamento dei domini strutturati [13], riportati nella tabella 3.5. Nel confronto è inoltre opportuno considerare che i modelli kernel-based prevedono, rispetto ai modelli proposti, l'impiego di maggiori risorse di calcolo ed operano attraverso l'impiego di metriche fissate a priori ed indipendenti dal task, in maniera simile a quanto avviene per i reservoir che non sfruttino output-feedback.

3.2.2 Mutagenesis

Le performance dei modelli sui tre task relativi al dataset Mutagenesis sono state calcolate ricorrendo ad una *doppia k-fold cross-validation* (si veda

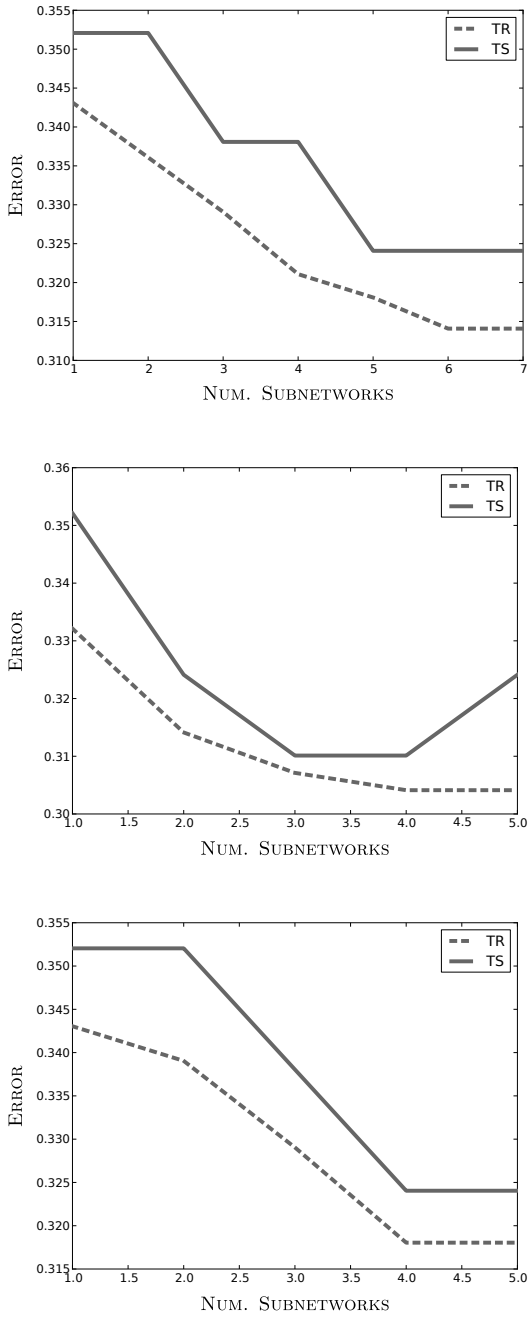


Figura 3.2: GraphESN-FOF. Curve di apprendimento sul dataset PTC. Errore di misclassificazione commesso nel corso della costruzione della rete. Linea tratteggiata: errore di training. Linea piena: errore di test.

il paragrafo 1.1.2) con un ciclo esterno di 10 fold ed un ciclo interno di 5 fold. La suddivisione del dataset è stata realizzata attraverso *stratificazione* (si veda il paragrafo 1.1.2). Per avere una stima più attendibile al variare dell'assegnazione iniziale dei pesi, 5 distinte ripetizioni sono state effettuate per ogni iperparametrizzazione testata.

Come nel caso di PTC, per ognuno dei modelli proposti sono state considerate due distinte configurazioni per la dimensione del reservoir delle sotto-reti: $N_R \in \{50, 30\}$. Il task structure-to-element di classificazione binaria, $N_Y = N_Z = 1$, prevede una dimensione delle etichette di input variabile, in base alla rappresentazione adottata: N_U vale 11, 13, 15 per AB, AB+C, AB+C+PS rispettivamente.

La soglia di convergenza del reservoir è stata fissata a $\epsilon = 10^{-5}$. Il coefficiente di contrazione σ è stato in questo caso selezionato attraverso model selection.

Per interrompere la costruzione delle reti si è fissato $\delta_{\text{var}} = 0.01$ e $\delta_{\text{tr}} = 0.11$, riferito al miscalssification-rate. Come per PTC, il numero massimo di sotto-reti è stato fissato a $\delta_{\text{size}} = 15$ nei casi in cui $N_R = 50$ e $\delta_{\text{size}} = 20$ con $N_R = 30$.

Il learning-rate utilizzato per l'allenamento del readout globale tramite LMS è $\eta = 10^{-3}$.

La model selection è stata realizzata variando gli iperparametri λ_{in}, λ_{fof}, λ_{r}, λ_{wd} e σ, secondo i valori riportati nella tabella 3.6. Selezione e test del modello hanno dunque coinvolto, per i tre task e limitatamente ai casi costruttivi, l'allenamento di un totale di 72300 diverse istanze di modello.

Sulle GraphESN la selezione degli iperparametri è stata eseguita facendo

Tabella 3.6: Iperparametri usati per la model selection sui 3 task Mutagenesis.

λ_{in}	λ_{fof}	λ_{r}	λ_{wd}	σ
$\{1.0, 0.1\}$	$\{1.0, 2.0\}$	$\{0.001, 0.01, 0.1\}$	$\{0.0, 0.01\}$	$\{1.0, 2.0\}$

Tabella 3.7: Iperparametri usati per la model selection di GraphESN sui 3 task Mutagenesis.

N_R	λ_{in}	λ_{r}
$\{500, 200, 100, 50\}$	$\{1.0, 0.1\}$	$\{0.01, 0.1, 0.2\}$

variare i valori di N_R, λ_{in}, λ_{r} come indicato nella tabella 3.7.

Nella tabella 3.8 sono riportati i risultati sperimentali ottenuti sui tre task Mutagenesis. I valori riportati rappresentano l'accuratezza percentuale di test (i.e. percentuale di grafi nel test-set classificata correttamente) mediata sulle 10 fold e la deviazione standard media. Il dettaglio dei risultati sulle singole fold è riportato nelle tabelle B.13-B.21 (si veda l'appendice B a pagina 124). Benché l'accuratezza nella predizione ottenuta dai modelli proposti risulti inferiore rispetto a quella raggiungibile con l'utilizzo di altri approcci (e.g. GNN [50]), i risultati sperimentali evidenziano come i modelli costruttivi rappresentino una valida alternativa ed un miglioramento rispetto alle GraphESN.

Tabella 3.8: Accuratezza media dei modelli e deviazione standard, in percentuale, sui 3 task Mutagenesis.

Model	N_R	AB	AB+C	AB+C+PS
GraphESN		75.2 ($\pm$0.8)	76.5 ($\pm$0.8)	80.3 ($\pm$0.8)
GraphESN-CF	50	79.0 ($\pm$2.3)	78.1 ($\pm$1.8)	79.4 ($\pm$0.5)
GraphESN-CF	30	79.6 ($\pm$2.5)	76.0 ($\pm$1.5)	79.7 ($\pm$0.2)
GraphESN-FW	50	80.5 ($\pm$2.8)	76.3 ($\pm$2.0)	79.3 ($\pm$1.8)
GraphESN-FW	30	79.7 ($\pm$2.8)	76.7 ($\pm$2.5)	80.6 ($\pm$1.9)
GraphESN-FOF	50	79.3 ($\pm$3.6)	76.0 ($\pm$2.8)	79.9 ($\pm$1.9)
GraphESN-FOF	30	76.8 ($\pm$3.7)	77.0 ($\pm$2.4)	80.0 ($\pm$2.5)

Tabella 3.9: Numero di sotto-reti dei modelli sui tre task Mutagenesis. Numero massimo, numero minimo, media e moda.

Model	Max	Min	Avg	Mode
GraphESN-CF	17	2	3.9	2
GraphESN-FW	15	2	4.6	2
GraphESN-FOF	18	2	5.5	4

Nella tabella 3.9 sono riportati alcuni dati sulle dimensioni raggiunte delle reti nell'affrontare i tre task del dataset. In questo caso la tabella evidenzia la presenza di molte reti di dimensioni minime, suggerendo l'adozione di un criterio di stop più sofisticato per interrompere la costruzione della rete. La condizione di arresto utilizzata guarda infatti al solo errore di training commesso al passo precedente e, nel caso di task complessi, è ipotizzabile che questo possa non essere sufficiente a determinare l'effettivo andamento del processo di learning. Criteri di stop più sofisticati potrebbero, ad esempio, usare un validation-set per determinare la dimensione ottimale della rete.

La figura 3.3 nella pagina successiva mostra un esempio di alcune curve di apprendimento che caratterizzano l'applicazione dei modelli sul dataset. Il valore di errore indicato è il misclassification-rate. La figura evidenzia come l'errore di test diminuisca al decrescere dell'errore di training, e dunque come il processo di apprendimento costruttivo abbia effettivamente la capacità di migliorare la capacità predittiva del modello. Si riscontra tuttavia uno scostamento piuttosto ampio fra l'errore commesso sui dati training e quello commesso sui dati di test, che suggerisce l'opportunità di perfezionare i risultati ottenuti adottando ulteriori criteri di regolarizzazione, che riducano la complessità della rete.

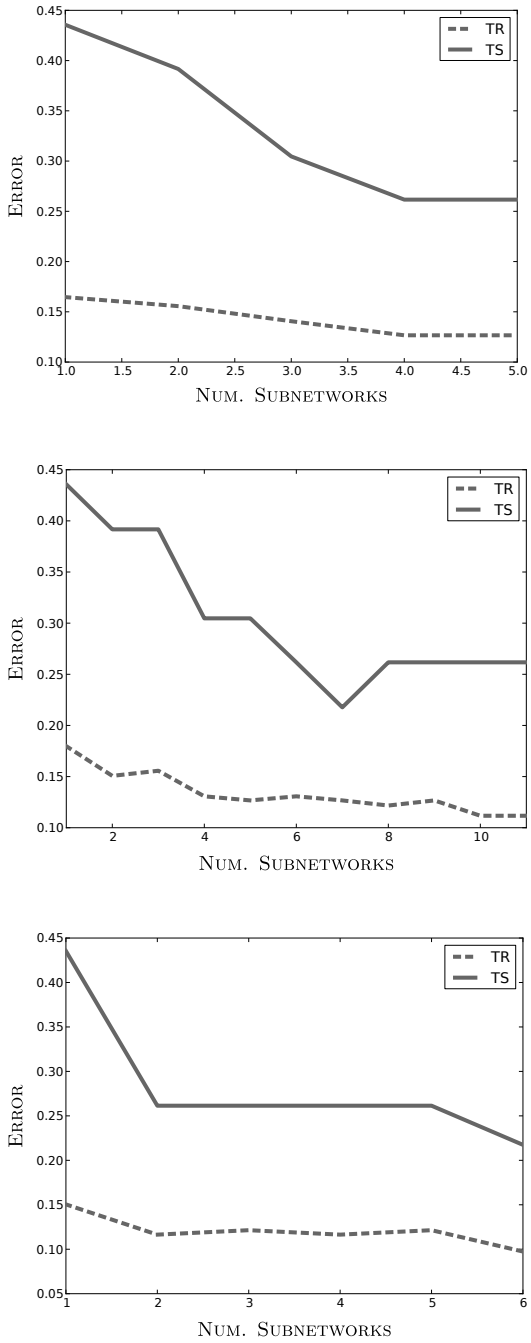


Figura 3.3: GraphESN-FOF. Curve di apprendimento sul dataset Mutagenesis. Errore di misclassificazione commesso nel corso della costruzione della rete.
Linea tratteggiata: errore di training. Linea piena: errore di test.

3.2.3 Bursi

Sul dataset Bursi è stato possibile sfruttare una suddivisione preesistente del dataset in dati di training e dati di test. La selezione del modello è stata dunque realizzata suddividendo il training-set in 5 fold tramite stratificazione e realizzando una *k-fold cross-validation* con dei validation-set per selezionare gli iperparametri (si veda il paragrafo 1.1.2). Successivamente si è usato il partizionamento originale del dataset, in trainig-set e test-set, per testare le capacità predittive dei modelli. La suddivisione in fold è avvenuta ricorrendo a *stratificazione* (si veda il paragrafo 1.1.2). Per ogni iperparametrizzazione testata sono state effettuate 5 ripetizioni distinte, in modo da variare l'assegnazione iniziale dei pesi.

Sul dataset sono stati testati modelli costruttivi con sotto-reservoir di dimensione $N_R = 100$. Il task structure-to-element di classificazione binaria $N_Y = N_Z = 1$ prevede una dimensione delle etichette di input $N_U = 14$.

La soglia di convergenza del reservoir utilizzata è $\epsilon = 10^{-5}$, mentre il coefficiente di contrazione è stato fissato a $\sigma = 1.0$.

Per interrompere la costruzione delle reti si è fissato $\delta_{\text{var}} = 0.01$ e $\delta_{\text{tr}} = 0.15$, riferito al miscalssification-rate, con un numero massimo di sotto-reti consentito $\delta_{\text{size}} = 10$.

Il learning-rate $\eta = 10^{-3}$ è stato utilizzato per l'allenamento del readout globale tramite LMS.

Data la dimensione del dataset, ed il relativo onere computazionale, la model selection su Bursi ha interessato i soli iperparametri λ_{fof} e λ_{r}, mantenendo fissi i parametri $\lambda_{\text{wd}} = 0$ per l'apprendimento nel readout globale e

Tabella 3.10: Iperparametri usati per la model selection su Bursi.

λ_{fof}	λ_{r}
$\{1.0, 2.0\}$	$\{0.0, 0.001, 0.01, 0.1\}$

Tabella 3.11: Iperparametri usati per la model selection di GraphESN su Bursi.

N_R	λ_{r}
$\{500, 200, 100\}$	$\{0.0, 0.001, 0.01, 0.1\}$

$\lambda_{\text{in}} = 1.0$ per lo scaling dei pesi dall'input. I valori degli iperparametri coinvolti nella model selection sono riportati nella tabella 3.10. Le performance su GraphESN sono invece state calcolate facendo variare i parametri N_R e λ_{r} in accordo ai valori riportati nella tabella 3.11. Gli esperimenti sul dataset hanno dunque richiesto l'allenamento complessivo di 680 diverse reti.

I risultati sperimentali ottenuti sono riportati nella tabella 3.12, con i valori ad indicare l'accuratezza percentuale e la deviazione standard riscontrate nel classificare i dati sia di training che di test. Dai dati emerge come in questo caso l'approccio costruttivo permetta di superare sistematicamente le performance predittive ottenute tramite GraphESN. Risulta inoltre come il miglioramento delle performance segua l'arricchimento della connettività progressivo realizzato dai tre modelli costruttivi.

Tabella 3.12: Accuratezza media in training e test dei modelli e deviazione standard, in percentuale, sul dataset Bursi.

Model	N_R	TR	TS
GraphESN		77.9 ($\pm$0.2)	76.2 ($\pm$0.2)
GraphESN-CF	100	78.3 ($\pm$0.4)	76.7 ($\pm$0.7)
GraphESN-FW	100	78.4 ($\pm$0.8)	77.1 ($\pm$0.7)
GraphESN-FOF	100	79.8 ($\pm$0.5)	78.0 ($\pm$0.9)

Tabella 3.13: Numero di sotto-reti dei modelli su Bursi. Numero massimo, numero minimo, media e moda.

Model	Max	Min	Avg	Mode
GraphESN-CF	5	2	3.2	2
GraphESN-FW	8	2	4.2	4
GraphESN-FOF	10	4	6.8	7

La tabella 3.9 riporta i dati sulle dimensioni raggiunte delle reti nell'affrontare il task. Si nota come il modello GraphESN-FOF riesca a procedere più a lungo nella costruzione della rete, probabilmente grazie ad una maggiore capacità di fitting che permette all'errore sul trainig-set di variare maggiormente, senza determinare l'interruzione del processo di costruzione della rete.

La figura 3.4 a pagina 98 mostra un esempio dell'andamento del training di una rete di tipo GraphESN-FOF applicata al dataset, evidenziando il progresso della capacità predittiva della rete nel corso del procedimento di costruzione della rete.

3.2.4 Angiotensin Converting Enzyme

Sul dataset ACE non è stato svolto un processo di selezione degli iperparametri ottimali dei modelli, limitando l'analisi sperimentale ad una singola iperparametrizzazione, scelta sulla base di valutazioni empiriche. La valutazione della performance del modello è stata svolta attraverso *k-fold cross-validation* su 10 fold (si veda il paragrafo 1.1.2), ripetendo il test 5 volte in modo da variare l'inizializzazione dei pesi di reservoir.

Il dataset ACE ha permesso di applicare i modelli ad un task structure-to-element di regressione, con $N_Y = N_Z = 1$. Le etichette associate ai vertici

dell'input risultano avere dimensione $N_U = 8$.

Per interrompere il processo di encoding è stata usata la soglia $\epsilon = 10^{-5}$, fissando un coefficiente di contrazione per ogni reservoir $\sigma = 1.0$.

Il processo di costruzione della rete è stato interrotto fissando $\delta_{\text{var}} = 0.01$ e senza tenere conto di alcun errore minimo sul training-set, $\delta_{\text{tr}} = -1.0$, con un numero massimo di sotto-reti consentito pari a $\delta_{\text{size}} = 30$.

L'allenamento del readout globale è stato svolto attraverso LMS con learning-rate $\eta = 10^{-3}$ e senza applicare alcun fattore di weight-decay, $\lambda_{\text{wd}} = 0$. L'apprendimento nelle sotto-reti è stato realizzato attraverso una Ridge Regression con fattore di regolarizzazione $\lambda_{\text{r}} = 0.1$.

Per l'inizializzazione dei pesi sono stati fissati i parametri $\lambda_{\text{in}} = 1.0$ e $\lambda_{\text{fof}} = 2.0$. Il test è stato svolto utilizzando sotto-reti di dimensione $N_R = 30$.

Per poter effettuare un confronto, l'allenamento su GraphESN è stato eseguito mantenendo, laddove abbia un senso, la stessa iperparametrizzazione usata per i modelli costruttivi. La dimensione del reservoir in questo caso è però stata fatta variare $N_R \in \{90, 150, 300, 450\}$. Nel seguito viene riportato solo il risultato migliore ottenuto (i.e. $N_R = 300$), in termini di performance predittiva.

La tabella 3.14 mostra i risultati sperimentali ottenuti, indicando l'errore quadratico medio e la deviazione standard mediati sulle 10 fold. Benché non sia stata effettuata una selezione rigorosa degli iperparametri ottimali per il task, è importante sottolineare come i risultati ottenuti siano comparabili con quelli riportati in letteratura relativamente all'applicazione di altri modelli per l'apprendimento di trasduzioni strutturali. La tabella 3.15 indica le performance su ACE ottenute attraverso modelli basati su kernel allo stato

Tabella 3.14: Errore quadratico medio in training e test dei modelli e deviazione standard sul dataset ACE.

Model	MSE (TR)	MSE (TS)
GraphESN	1.83 (±0.01)	2.13 (±0.03)
GraphESN-CF	2.03 (±0.07)	2.37 (±0.15)
GraphESN-FW	1.71 (±0.02)	2.02 (±0.06)
GraphESN-FOF	1.60 (±0.07)	2.03 (±0.06)

Tabella 3.15: Errore quadratico medio dei modelli e deviazione standard di metodi basati su kernel sul dataset ACE.

Kernel	MSE
DotProduct	1.60 (±0.59)
2D-LAP(CK)	1.67 (±0.63)
2D-LAP(OA)	1.81 (±0.65)
3D-LAP(CK)	2.04 (±0.78)
3D-LAP(OA)	1.91 (±0.71)
XD-LAP(CK)	1.90 (±0.71)
XD-LAP(OA)	1.99 (±0.71)
MACCS keys	2.02 (±0.69)
Marginalized GK	1.74 (±0.72)
Pharmacophore	2.24 (±0.82)
OAK	1.49 (±0.68)
Tanimoto DFS	1.85 (±0.64)

dell'arte nel trattamento di domini strutturati [27].

La figura 3.5 a pagina 99 riporta una campione delle curve di apprendimento ottenute utilizzando una rete di tipo GraphESN-FOF sul dataset. I grafici dimostrano l'efficacia del processo di apprendimento, mostrando tuttavia come la mancanza della scelta di un setting specifico su ogni fold risulti in un comportamento non uniforme dei modelli sulle varie fold.

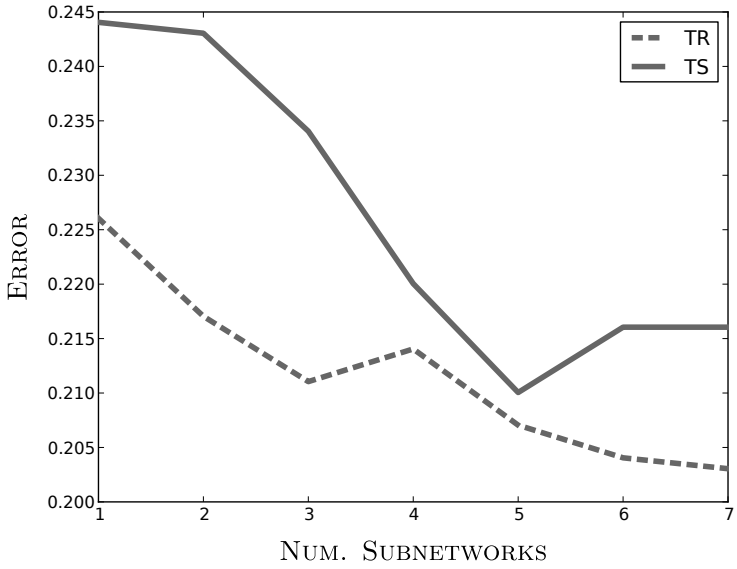


Figura 3.4: GraphESN-FOF. Curve di apprendimento sul dataset Bursi. Errore di misclassificazione commesso nel corso della costruzione della rete. Linea tratteggiata: errore di training. Linea piena: errore di test.

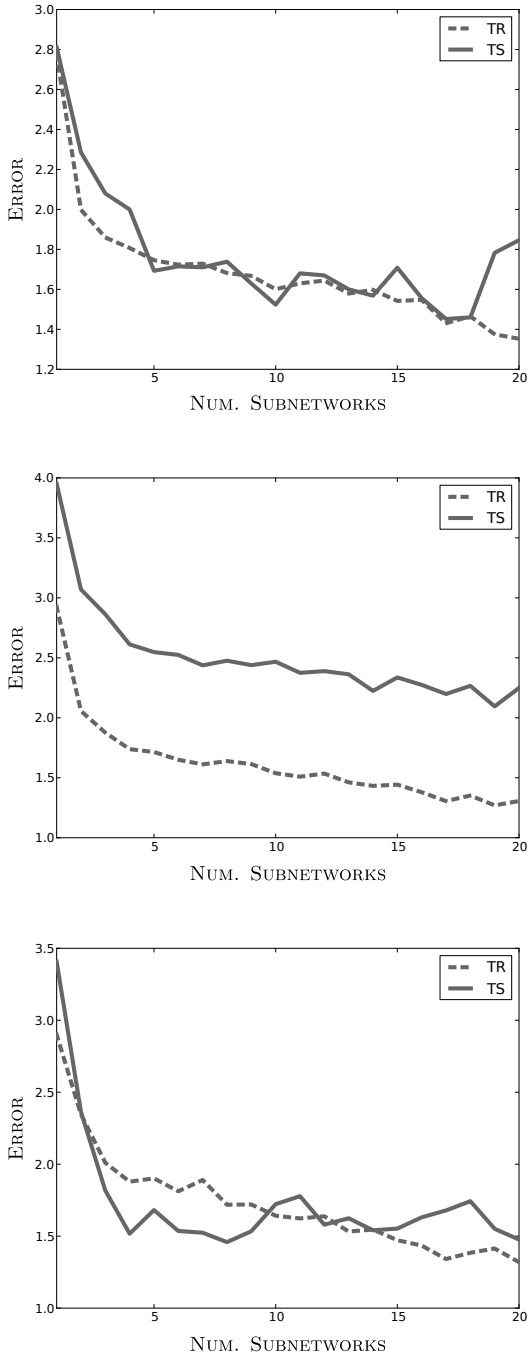


Figura 3.5: GraphESN-FOF. Curve di apprendimento sul dataset ACE. Errore quadratico medio (MSE) commesso nel corso della costruzione della rete.
Linea tratteggiata: errore di training. Linea piena: errore di test.

3.3 Considerazioni

Guardando ai risultati emerge piuttosto chiaramente come la strategia costruttiva, seppur meno onerosa dal punto di vista computazionale, possa rappresentare un potenziamento delle GraphESN. Sia la fase di training che la selezione del modello si avvalgono infatti dell'approccio costruttivo per ridurre il costo computazionale senza che questo determini una perdita nella capacità predittiva della rete: i risultati ottenuti mostrano infatti come i modelli proposti siano in grado di superare nella maggior parte dei casi le performance raggiunte da GraphESN, raggiungendo risultati comparabili negli altri casi. In quest'ottica i modelli introdotti si configurano come uno strumento utile nel trattamento dei domini strutturati, in grado di soddisfare efficacemente i vincoli imposti dalla presenza di risorse di calcolo o di tempo limitate.

Guardando alla sola analisi delle performance, tuttavia, alcune caratteristiche dei modelli proposti rimangono poco chiare. Si nota ad esempio come l'introduzione e lo sfruttamento di un meccanismo stabile di output-feedback, nel modello GraphESN-FOF, non corrisponda ad un miglioramento sistematico dell'accuratezza predittiva della rete. Ulteriori analisi sperimentali hanno tuttavia permesso di verificare l'effetto dei segnali di output-feedback sui reservoir delle sotto-reti. L'organizzazione dello spazio degli stati dei reservoir è stata investigata attraverso la Principal Component Analysis (PCA) [34]. La figura 3.6 a pagina 102 mostra le prime due componenti principali dello spazio degli stati dei reservoir di sotto-reti appartenenti ad una GraphESN-FOF. I punti dei plot corrispondono ai grafi in input, colorati in base al

target loro assegnato e disposti nello spazio delle features secondo l'encoding corrispondente. Lo stesso procedimento e lo stesso setting sperimentale sono stati utilizzati per realizzare la figura 3.7 a pagina 103, che mostra lo spazio degli stati dei reservoir di una rete di tipo GraphESN-CF, in cui non sono presenti connessioni di output-feedback. Dal confronto delle due figure emerge chiaramente come il meccanismo di output-feedback abbia l'effetto di modificare le dinamiche del processo di encoding, organizzando gli input nello spazio degli stati in maniera coerente con il task affrontato ed aumentando progressivamente la distanza fra input appartenenti a classi distinte.
L'analisi della PCA è stata ripetuta sul dataset ACE, corrispondente ad un task di regressione. La figura 3.8 a pagina 104 mostra lo spazio degli stati di una rete GraphESN-FOF, mentre la figura 3.9 si riferisce ad una rete GraphESN-CF con lo stesso setting. Gli input sono in questo caso colorati in scale di grigio in accordo al valore target corrispondente, che varia in un intervallo continuo. Anche in questo caso le figure evidenziano come la presenza di output-feedback abbia l'effetto di organizzare la rappresentazione dell'input all'interno dello spazio degli stati in maniera consistente con il task. Al contrario, nei due casi visti, l'assenza di output-feedback fa sì che i reservoir delle varie sotto-reti si comportino in maniera sostanzialmente simile l'uno all'altro, con variazioni dell'encoding che dipendono unicamente dalla diversa inizializzazione random dei pesi sulle connessioni ricorrenti.
Dall'analisi della PCA sugli stati dei reservoir risulta quindi ben evidente come la presenza degli output-feedback introduca effettivamente dell'informazione supervisionata all'interno di reservoir non adattivi.

L'efficacia dell'adozione di uno schema di output-feedback è inoltre confer-

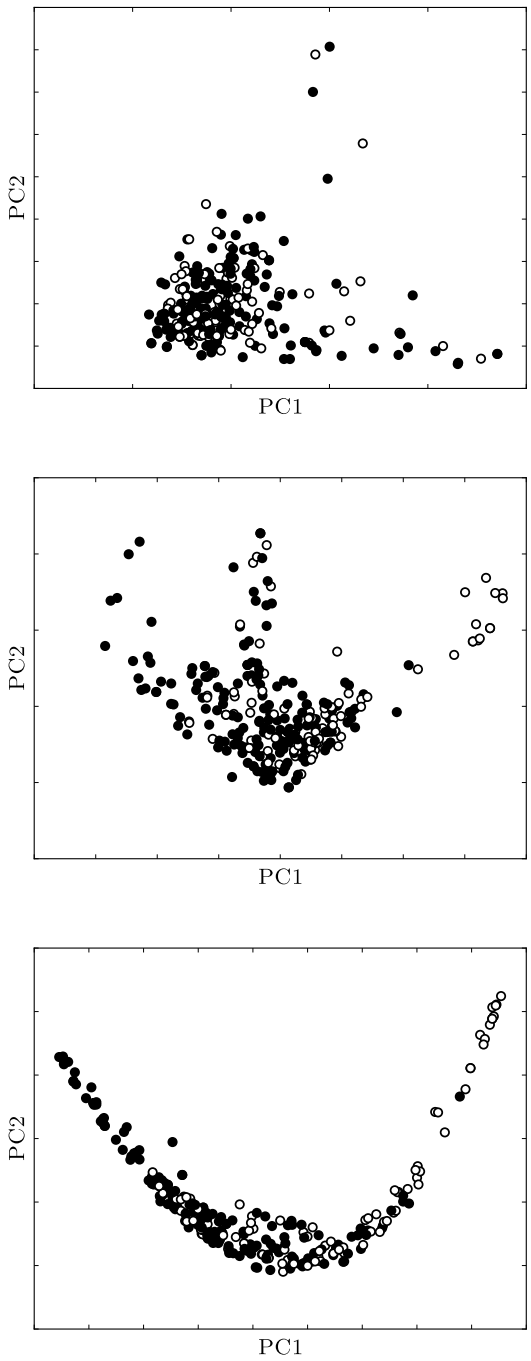


Figura 3.6: Plot delle prime due componenti principali dello spazio degli stati dei reservoir nelle sotto-reti. Modello GraphESN-FOF applicato al task PTC-FR.
In nero: gli input con target -1. In bianco: gli input con target $+1$.
Reservoir riportati: (*alto*) sotto-rete 1; (*centro*) sotto-rete 5; (*basso*) sotto-rete 9.

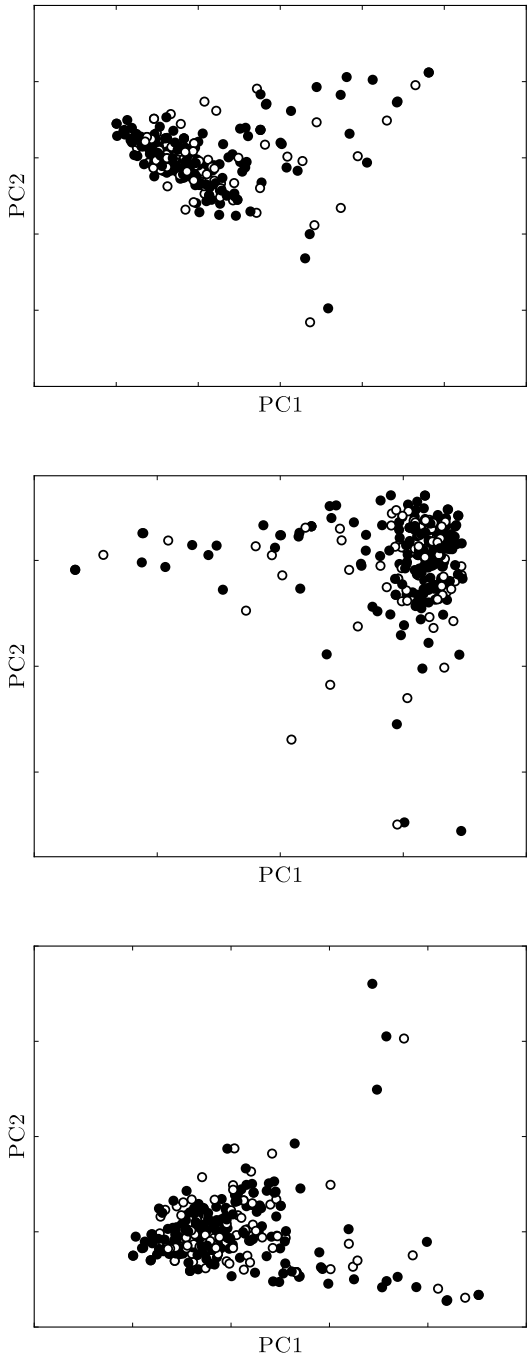


Figura 3.7: Plot delle prime due componenti principali dello spazio degli stati dei reservoir nelle sotto-reti. Modello GraphESN-CF applicato al task PTC-FR.
In nero: gli input con target -1. In bianco: gli input con target $+1$.
Reservoir riportati: (*alto*) sotto-rete 1; (*centro*) sotto-rete 4; (*basso*) sotto-rete 8.

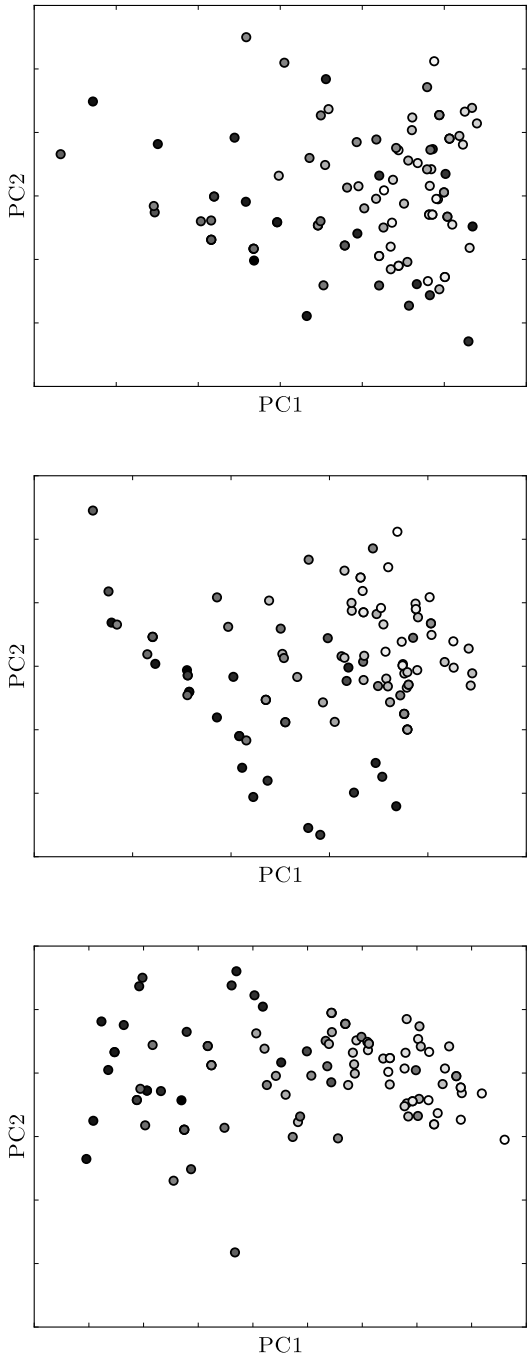


Figura 3.8: Plot delle prime due componenti principali dello spazio degli stati dei reservoir nelle sotto-reti. Modello GraphESN-FOF applicato al task ACE.
In scuro gli input con valori target più alti.
Reservoir riportati: (*alto*) sotto-rete 1; (*centro*) sotto-rete 4; (*basso*) sotto-rete 9.

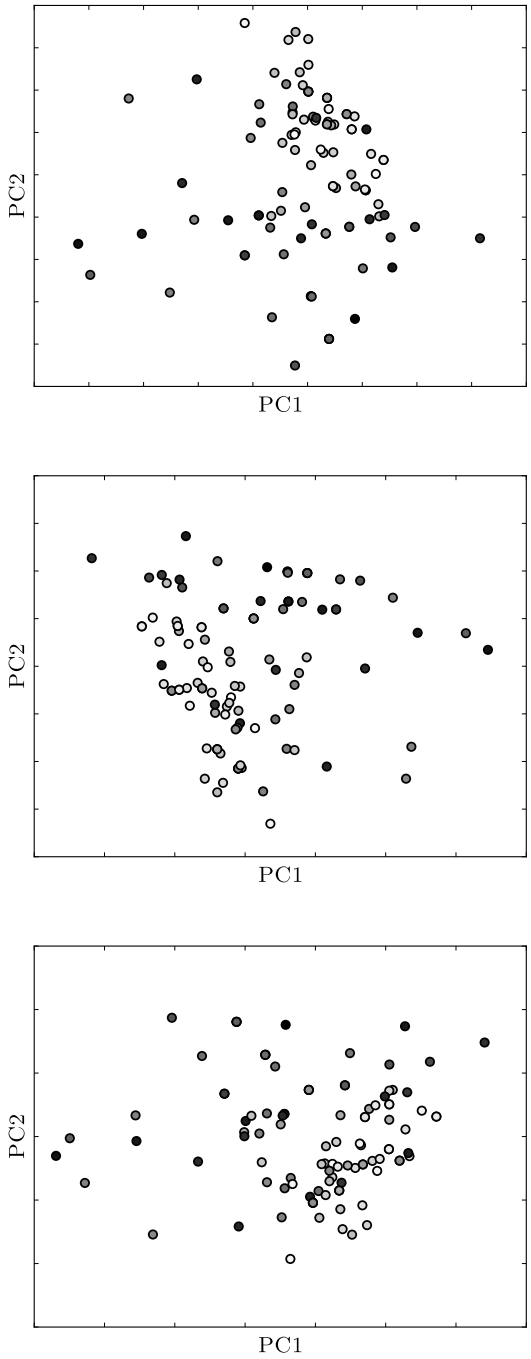


Figura 3.9: Plot delle prime due componenti principali dello spazio degli stati dei reservoir nelle sotto-reti. Modello GraphESN-CF applicato al task ACE.
In scuro gli input con valori target più alti.
Reservoir riportati: (*alto*) sotto-rete 1; (*centro*) sotto-rete 5; (*basso*) sotto-rete 9.

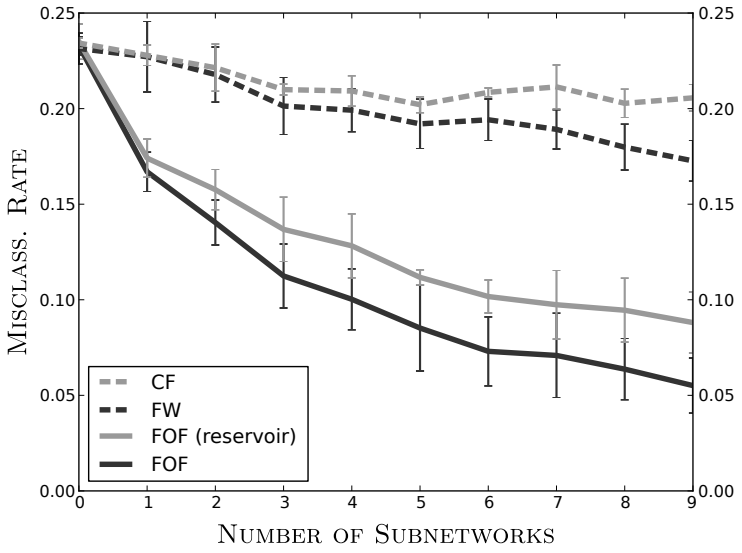


Figura 3.10: Confronto fra i modelli: errore di training (i.e. misclassification rate) e deviazione standard al crescere della rete. Task PTC-FM, $N_R = 100$.

mata, secondo un approccio diverso, dall'analisi della capacità di fitting dei modelli. La figura 3.10 mostra il confronto fra l'errore commesso, a parità di iperparametrizzazione ed al crescere della rete. Oltre ai tre proposti è stato in questo caso considerato un quarto modello, consistente in una GraphESN-FOF con connessioni di output-feedback che andassero unicamente verso i reservoir delle sotto-reti successive, e non anche verso il readout. Tenendo in considerazione il solo errore commesso sul training-set, ovvero la semplice capacità di fitting dei modelli, risulta evidente dal grafico come l'introduzione di informazione supervisionata attraverso gli output-feedback determini una accresciuta capacità di approssimazione da parte del modello.

Dalle analisi supplementari risulta dunque come l'introduzione degli output-feedback rappresenti effettivamente una importante risorsa tesa a potenziare l'encoding delle reti, superando in parte i limiti legati all'uso di reservoir non

adattivi. A fronte di questo, le osservazioni sia qualitative che quantitative sui risultati sperimentali suggeriscono l'impiego di forti meccanismi di regolarizzazione e di criteri di stop sofisticati, che impediscano alla rete di accrescere eccessivamente la propria complessità.

Sul piano pratico, gli esperimenti lasciano invece emergere un altro aspetto critico. Poiché per la natura del Reservoir Computing la dimensione del reservoir è estremamente importante ai fini del funzionamento della rete, è infatti necessario stabilire a priori quali siano le dimensioni dei reservoir delle sotto-reti. Nel farlo ci si trova di fronte ad una sorta di trade-off:

- Con N_R troppo piccolo i reservoir delle sotto-reti perdono la capacità di espandere l'input su uno spazio degli stati ad alta dimensionalità, riducendo la capacità di apprendimento delle sotto-reti e della rete nel suo complesso. In questo scenario, inoltre, si rischia di perdere il vantaggio computazionale, pagando il prezzo di un numero molto alto di iterazioni necessarie alla costruzione della rete (si veda il paragrafo 2.4 a pagina 66). D'altra parte l'utilizzo di sotto-reservoir di piccole dimensioni offre l'opportunità di avere maggiore controllo sulla crescita della rete, che avverrà in maniera poco discontinua.

- Con N_R troppo grande si ottengono al contrario singole sotto-reti con una grande capacità di apprendimento che però più facilmente metteranno la rete nella condizione di incappare in situazioni di overfitting. L'aggiunta di una singola sotto-rete determina infatti in questo scenario un aumento elevato, e più difficilmente controllabile, della complessità del

modello, rendendo complesso il compito di apprendere dinamicamente una topologia che sia effettivamente adeguata al task affrontato.

Pur consentendo la determinazione automatica della topologia della rete, dunque, l'approccio costruttivo necessita di accorgimenti specifici che permettano di controllare la complessità della rete, attraverso esperienza empirica, criteri di stop sofisticati e l'adozione di meccanismi di regolarizzazione che possano evitare il verificarsi di situazioni di overfitting.

4 Conclusioni

Nel corso della tesi sono stati presentati dei nuovi modelli di Reti Neurali Ricorsive per l'apprendimento supervisionato di trasduzioni strutturali su grafi. I modelli proposti estendono le GraphESN attraverso un approccio costruttivo, innovativo nell'ambito del Reservoir Computing, che consente di costruire la rete progressivamente. Ereditando dalle GraphESN la capacità di apprendere trasduzioni strutturali generiche attraverso algoritmi di apprendimento efficienti, i modelli presentati sfruttano la strategia costruttiva per offrire soluzioni originali ad alcuni dei problemi aperti nell'ambito del Reservoir Computing.

L'approccio costruttivo consente nei modelli proposti di determinare in maniera automatica (i.e. guidata dal task affrontato) il numero di unità che compongono la rete nel suo complesso e la sua topologia. Questa caratteristica permette di non dover fissare a priori la dimensione e la struttura della rete, in contrapposizione con quanto avviene per i modelli di Reservoir Computing esistenti, che fanno affidamento su reservoir prefissati. Tale vantaggio acquista rilievo se si considera che la dimensione del reservoir risulta essere un fattore

estremamente importante per caratterizzare la complessità della rete. La determinazione automatica del numero di unità ha quindi anche l'effetto di ridurre il costo computazionale necessario alla selezione del modello.

Sfruttando la strategia costruttiva è stato inoltre realizzato un meccanismo stabile di output-feedback, usato per modificare le dinamiche dei reservoir. Nel corso del procedimento incrementale di costruzione della rete, infatti, i modelli costruttivi possono avvalersi delle informazioni già apprese in precedenza che, attraverso uno schema di connessioni fra le sotto-reti, sono state introdotte all'interno del processo di codifica dell'input. Anche in presenza di reservoir prefissati e non soggetti ad alcun tipo di adattamento, dunque, i modelli proposti risultano in grado di sfruttare, durante il procedimento di encoding, delle informazioni supervisionate, direttamente dipendenti dal task. Lo schema di output-feedback individua quindi una possibile strategia per affrontare uno dei problemi aperti caratteristici del Reservoir Computing: la presenza di reservoir fissati a priori e non adattivi, con dinamiche interne unicamente guidate dalla natura degli input e non legate anche al problema affrontato.

Le soluzioni proposte sono state sviluppate secondo un processo incrementale, da cui sono derivati tre distinti modelli, ognuno in grado di generalizzare i precedenti. La strategia costruttiva è stata introdotta attraverso il modello GraphESN-CF, in grado di determinare automaticamente le dimensioni di una rete con topologia flat (i.e. con un unico livello di sotto-reti). Il modello GraphESN-FW ha esteso il precedente introducendo delle connessioni fra le sotto-reti, in modo da realizzare una rete di tipo Reservoir Computing con topologia multilayer non lineare. Con il modello GraphESN-FOF si è infine ulteriormente estesa la struttura delle connessioni fra le sotto-reti, sfruttando

gli output-feedback per guidare il processo di encoding coerentemente con il task trattato.

I modelli realizzati si caratterizzano per una notevole flessibilità ed efficienza dal punto di vista computazionale. L'opportunità di suddividere un task in sotto-problemi consente di ampliare la gamma di strategie utilizzabili nell'allenamento e permette di ricorrere a strumenti caratteristici dell'approccio costruttivo, come l'impiego di un pool. Oltre a questo, la decomposizione del problema offre la possibilità di impiegare sotto-reti di dimensioni contenute, specializzate a risolvere solo un sotto-task specifico, determinando così un notevole vantaggio computazionale nella realizzazione dell'allenamento della rete nel suo complesso. Anche guardando al solo ambito del Reservoir Computing, di per sé caratterizzato da oneri computazionali estremamente ridotti, i modelli realizzati permettono quindi di guadagnare in efficienza senza che questo comporti perdite in accuratezza di predizione.

L'efficacia dei modelli realizzati è stata testata su problemi reali appartenenti all'ambito della Chemioinformatica. Gli esperimenti svolti hanno mostrato nei modelli proposti la capacità di ottenere un'accuratezza nelle predizioni comparabile, ed in molti casi migliore, rispetto ai modelli non costruttivi. Il ruolo degli output-feedback è stato indagato attraverso un approccio sia quantitativo che qualitativo, mettendo in luce come l'introduzione di informazione supervisionata all'interno dei reservoir sia effettivamente in grado di influenzarne le dinamiche in maniera coerente con il task affrontato, in netta contrapposizione con quanto avviene in assenza di output-feedback. L'analisi sperimentale ha anche fatto emergere alcuni aspetti critici nei modelli proposti. L'impiego di output-feedback verso i reservoir non risulta

infatti in maniera sistematica in un aumento della capacità di generalizzazione dei modelli, suggerendo quindi l'adozione di meccanismi di regolarizzazione forti, tesi ad evitare situazioni di overfitting. Inoltre, benché le dimensioni della rete siano determinate in maniera automatica, il numero di unità che compongono i reservoir delle sotto-reti risulta essere un fattore importante per le capacità dei modelli. La determinazione di un valore adeguato per tale parametro può tuttavia essere affrontata tramite un processo di selezione che coinvolga l'esperienza empirica o, eventualmente, attraverso il processo di model selection.

L'aver indirizzato un approccio innovativo nel campo del Reservoir Computing fornisce anche spunti per ulteriori sperimentazioni. La strategia costruttiva proposta è estremamente flessibile ed è ragionevole ipotizzare che altre varianti topologiche, oltre a quelle realizzate, possano essere oggetto di investigazione. I modelli sperimentati sfruttano infatti uno schema di connessioni fra le sotto-reti molto semplice, che può facilmente essere modificato in modo da implementare, ad esempio, politiche di propagazione degli output-feedback che tengano conto della topologia (e.g. collegando una sotto-rete solo alle sotto-reti più prossime) o delle performance (e.g. assegnando alle connessioni pesi differenti in base alle performance ottenute dalle singole sotto-reti). Altrettanto importante e variegata è anche la gamma degli algoritmi di learning o dei sotto-task utilizzabili per l'allenamento delle sotto-reti, anch'essa possibile oggetto di ulteriori indagini così come l'impiego di sotto-reti eterogenee (e.g. nella dimensione del reservoir) selezionate tramite un pool.

È infine opportuno sottolineare che parte dei contenuti discussi nel corso

della tesi sono stati riportati nell'articolo *Constructive Reservoir Computation with Output Feedbacks for Structured Domains* [19], che verrà discusso nel corso del XX European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning.

A

Dettaglio del costo computazionale

Consideriamo una rete di tipo GraphESN-FOF formata da *NSN* sotto-reti con reservoir completamente connessi di dimensione N_R. Indichiamo con N_Z la dimensione dell'output delle sotto-reti e con N_Y quella del readout globale, entrambe trascurabili visto che tipicamente $N_Y = N_Z = 1$. Sia N_U la dimensione delle etichette associate ad ogni vertice dell'input e $|\mathcal{G}|$ la dimensione del dataset (i.e il numero totale di input).

Di seguito, nei paragrafi A.1-A.5 viene riportato il calcolo del costo computazionale relativo alle varie fasi di utilizzo del modello: encoding dei dati, allenamento delle sotto-reti, allenamento del readout-globale, calcolo dell'output.

Il paragrafo A.6 riporta invece il calcolo del costo computazionale per la model selection di una GraphESN standard.

A.1 Encoding dei dati

Consideriamo un singolo passo del processo di encoding nella rete i-esima.

$$\mathbf{x}_t^{(i)}(v) = f(\mathbf{W}_{\text{in}}^{(i)}\,\mathbf{u}(v) + \sum_{w\in\mathcal{N}(v)} \hat{\mathbf{W}}^{(i)}\,\mathbf{x}_{t-1}^{(i)}(w) + \sum_{j=1}^{i-1} \mathbf{W}_{\text{fof}}^{(ij)}\,\mathbf{z}^{(j)}(v)) \qquad \text{(A.1)}$$

con $\mathbf{W}_{\text{in}}^{(i)} \in \mathbb{R}^{N_R\times N_U}$, $\hat{\mathbf{W}}^{(i)} \in \mathbb{R}^{N_R\times N_R}$, $\mathbf{W}_{\text{fof}}^{(ij)} \in \mathbb{R}^{N_R\times N_Z}$.

Indicando con $MAXV$ il numero massimo di vertici di un grafo di input, il costo per l'encoding di un grafo $\mathbf{g}$ è dato dalla somma dei costi:

- $\mathbf{W}_{\text{in}}^{(i)}\,\mathbf{u}(v)$, ha costo $O(N_R\ N_U)$ e viene calcolato una sola volta per ogni vertice del'input.
- $\sum_{j=1}^{i-1} \mathbf{W}_{\text{fof}}^{(ij)}\,\mathbf{z}^{(j)}(v)$, ha costo $O(N_R\ N_Y\ (i-1))$ e viene calcolato una sola volta per ogni vertice dell'input.
- $\sum_{w\in\mathcal{N}(v)} \hat{\mathbf{W}}^{(i)}\,\mathbf{x}_{t-1}^{(i)}(w)$, ha costo $O(N_R^2)$ e viene calcolato per ogni vertice ad ogni iterazione.

Chiamando $MAXIT$ il numero massimo di iterazioni, il costo dell'encoding di un singolo grafo è dunque dato da

$$O(N_R\ N_U\ MAXV + N_R\ N_Y\ (i-1)\ MAXV + N_R^2\ MAXV\ MAXIT)) \qquad \text{(A.2)}$$

che, ripetuto per $|\mathcal{G}|$ volte, ovvero per ogni grafo nel dataset, è

$$\begin{aligned} &O(|\mathcal{G}|\ (N_R\ N_U\ MAXV + N_R\ N_Y\ (i-1)\ MAXV + N_R^2\ MAXV\ MAXIT)) \\ &\simeq O(|\mathcal{G}|\ N_R^2\ MAXV\ MAXIT) \end{aligned} \qquad \text{(A.3)}$$

che scala quadraticamente con la dimensione del reservoir, rendendo trascurabile la presenza delle connessioni di output-feedback.

Per l'intera rete, ovvero considerando tutte le *NSN* sottoreti, il costo complessivo del processo di encoding è dato da

$$O(NSN\ |\mathcal{G}|\ N_R^2\ MAXV\ MAXIT) \tag{A.4}$$

e scala linearmente con la dimensione del dataset e dei grafi e quadraticamente con la dimensione del reservoir.

Considerando l'uso di reservoir sparsi, con al più M connessioni in ingresso ad ogni unità, il costo computazionale dell'encoding può essere ulteriormente ridotto a

$$O(NSN\ |\mathcal{G}|\ N_R\ M\ MAXV\ MAXIT) \tag{A.5}$$

che dipende linearmente dalla dimensione del reservoir e da M.

A.2 Training delle sotto-reti

Consideriamo il caso in cui la rete i-esima sia allenata tramite Ridge Regression. In formula:

$$\mathbf{W}_{\text{out}}^{\text{T}} = (\mathbf{X}^T\mathbf{X} + \lambda_{\text{r}}\mathbf{I})^{-1}\mathbf{X}^T\mathbf{Y}_{\text{target}} \tag{A.6}$$

con $\mathbf{X} \in \mathbb{R}^{|\mathcal{G}|\times(N_R+N_Z(i-1))}$ matrice che contiene tutti gli input del readout (i.e. per ogni grafo, la codifica ottenuta dal sotto-reservoir più gli output-feedback provenienti dalle sotto-reti precedenti), $\mathbf{Y}_{\text{target}} \in \mathbb{R}^{|\mathcal{G}|\times N_Y}$

contenente i valori target (i.e. un vettore di uscita per ogni input) e $\mathbf{I} \in \mathbb{R}^{(N_R+N_Z(i-1))\times(N_R+N_Z(i-1))}$ matrice identità.

Chiamiamo per semplicità $N_F(i) = N_R + N_Z\,(i-1)$, ad indicare il numero di *features* in input al readout i-esimo. I singoli passaggi nel calcolo della Ridge Regression hanno il seguente costo:

- $\mathbf{X}^T\mathbf{X}$, prodotto di matrici: $O(N_F(i)^2\ |\mathcal{G}|)$.
- $(\mathbf{X}^T\mathbf{X} + \lambda_{\mathrm{r}}\mathbf{I})^{-1}$, inversione: $O(N_F(i)^3)$.
- $\mathbf{X}^T\mathbf{Y}_{\mathrm{target}}$, prodotto di matrici: $O(N_F(i)\ |\mathcal{G}|\ N_Z)$.
- $(\mathbf{X}^T\mathbf{X} + \lambda_{\mathrm{r}}\mathbf{I})^{-1}\mathbf{X}^T\mathbf{Y}_{\mathrm{target}}$, prodotto di matrici: $O(N_F(i)^2 N_Z)$.

Il costo complessivo per l'allenamento della sotto-rete i-esima è quindi dato da:

$$\begin{aligned} &O(N_F(i)^3 + N_F(i)^2\ |\mathcal{G}| + N_F(i)^2 N_Z + N_F(i)\ |\mathcal{G}|\ N_Z) \\ &\simeq O(N_F(i)^3 + N_F(i)^2\ |\mathcal{G}| + N_F(i)^2 + N_F(i)\ |\mathcal{G}|\ ) \\ &\simeq O(N_F(i)^3 + N_F(i)^2\ |\mathcal{G}|) \end{aligned} \tag{A.7}$$

Nel passaggio da una singola sotto-rete alla rete nel suo complesso bisogna tener conto del fatto che N_F è funzione di i (i.e.$N_F(i) = N_R + N_Z\,(i-1)$). Il costo dell'allenamento di *NSN* sotto-reti è quindi dato da

$$\begin{aligned} &O(\textstyle\sum_{i=1}^{NSN}(N_F(i)^3 + N_F(i)^2\ |\mathcal{G}|)) \\ &= O(\textstyle\sum_{i=1}^{NSN} N_F(i)^3 + \sum_{i=1}^{NSN} N_F(i)^2\ |\mathcal{G}|) \end{aligned} \tag{A.8}$$

Per semplicità, consideriamo le due sommatorie separatamente.

$$
\begin{aligned}
&O(\sum_{i=1}^{NSN} N_F(i)^3) = O(\sum_{i=1}^{NSN}(N_R + N_Z(i-1))^3) \\
&= O(\sum_{i=1}^{NSN}(N_R^3 + 3N_R^2\ N_Z\ (i-1) + N_Z^3\ (i-1)^3 + 3N_R\ N_Z^2\ (i-1)^2) \\
&= O(NSN\ N_R^3 + 3N_R^2\ N_Z \sum_{i=1}^{NSN}(i-1) + N_Z^3 \sum_{i=1}^{NSN}(i-1)^3 + 3N_R\ N_Z^2 \sum_{i=1}^{NSN}(i-1)^2) \\
&\simeq O(NSN\ N_R^3 + N_R^2\ \sum_{i=1}^{NSN}(i-1) + \sum_{i=1}^{NSN}(i-1)^3 + N_R\ \sum_{i=1}^{NSN}(i-1)^2) \\
&\simeq O(NSN\ N_R^3 + N_R^2\ NSN^2 + NSN^4 + N_R\ NSN^3)
\end{aligned}
\tag{A.9}
$$

la seconda parte dell'equazione (A.8) è invece data da

$$
\begin{aligned}
&O(\sum_{i=1}^{NSN} N_F(i)^2\ |\mathcal{G}|) \\
&= O(|\mathcal{G}| \sum_{i=1}^{NSN}(N_R + N_Z\ (i-1))^2) \\
&= O(|\mathcal{G}| \sum_{i=1}^{NSN}(N_R^2 + 2N_R\ N_Z\ (i-1) + N_Z^2(i-1)^2)) \\
&= O(|\mathcal{G}|(NSN\ N_R^2 + 2N_R\ N_Z \sum_{i=1}^{NSN}(i-1) + N_Z^2 \sum_{i=1}^{NSN}(i-1)^2) \\
&\simeq O(|\mathcal{G}|(NSN\ N_R^2 + N_R\ NSN^2 + NSN^3)) \\
&= O(NSN\ N_R^2\ |\mathcal{G}| + N_R\ NSN^2\ |\mathcal{G}| + NSN^3\ |\mathcal{G}|)
\end{aligned}
\tag{A.10}
$$

Mettendo insieme le equazioni (A.9) e (A.10) si ottiene il costo complessivo

dell'allenamento delle sotto-reti tramite Ridge Regression:

$$O(NSN^4 + N_R\ NSN^3 + N_R^2\ NSN^2 + NSN\ N_R^3 + NSN^3\ |\mathcal{G}| + NSN^2\ N_R\ |\mathcal{G}| + NSN\ N_R^2\ |\mathcal{G}|) \quad \text{(A.11)}$$

A.3 Training del readout globale

Consideriamo due possibili casi di allenamento per il readout globale: l'applicazione dell'algoritmo diretto di Ridge Regression o l'impiego dell'algoritmo iterativo LMS.

A.3.1 Ridge Regression

Nel caso in cui il readout globale sia allenato tramite Ridge Regresison, la formula ricalca quella riportata nell'equazione (A.6), in cui però la dimensione degli input è data solo dagli output delle sotto-reti (i.e. senza alcun input proveniente da un reservoir), quindi $\mathbf{X} \in \mathbb{R}^{|\mathcal{G}| \times N_Z\,(i-1)}$.
Chiamando $N_F(i) = N_Z\,(i-1)$, il costo computazionale per l'allenamento della rete al passo i-esimo è dato da

$$O(N_F(i)^3 + N_F(i)^2\ |\mathcal{G}|) \quad \text{(A.12)}$$

Il costo complessivo dell'allenamento del readout globale è quindi:

$$\begin{aligned} & O(N_F(i)^3 + N_F(i)^2 \ |\mathcal{G}|) \\ & = O(\sum_{i=1}^{NSN}(N_Z^3 \ (i-1)^3 + (N_Z^2 \ (i-1)^2 \ |\mathcal{G}|) \\ & = O(N_Z^3 \ \sum_{i=1}^{NSN}(i-1)^3 + N_Z^2 \ |\mathcal{G}| \ \sum_{i=1}^{NSN}(i-1)^2) \\ & \simeq O(NSN^4 + NSN^3|\mathcal{G}|) \end{aligned} \tag{A.13}$$

A.3.2 LMS

In questo caso l'algoritmo è iterativo. Chiamiamo *MAXIT* il numero massimo di iterazioni necessarie perché l'algoritmo converga. L'allenamento del readout globale, dopo aver aggiunto i sotto-reti, è dato da

$$O(MAXIT \ |\mathcal{G}| \ (i-1) \ N_Y^2) \tag{A.14}$$

Complessivamente, il costo necessario all'allenamento del readout globale della rete tramite LMS è quindi

$$\begin{aligned} & O(\sum_{i=1}^{NSN}(MAXIT \ |\mathcal{G}| \ (i-1) \ N_Y^2)) \\ & = O(MAXIT \ N_Y^2 \ |\mathcal{G}| \ \sum_{i=1}^{NSN}(i-1)) \\ & \simeq O(MAXIT \ N_Y^2 \ |\mathcal{G}| \ NSN^2) \\ & \simeq O(MAXIT \ |\mathcal{G}| \ NSN^2) \end{aligned} \tag{A.15}$$

A.4 Calcolo dell'output delle sotto-reti

Per il calcolo dell'output tutti i valori di input — sia provenienti dal reservoir che gli output-feedback — sono calcolati in precedenza. Il calcolo dell'uscita della rete i-esima per il grafo $\mathbf{g}$ corrisponde dunque alla moltiplicazione

$$\mathbf{W}_{\text{out}}^{(i)} \left[\mathcal{X}(\mathbf{x}^{(i)}(\mathbf{g})), \mathbf{z}^{(1)}(\mathbf{g}), \ldots, \mathbf{z}^{(i-1)}(\mathbf{g})\right] \equiv \mathbf{W}_{\text{out}}^{(i)} \, \mathbf{v}_{\text{feat}} \tag{A.16}$$

con $\mathbf{W}_{\text{out}}^{(i)} \in \mathbb{R}^{N_Z \times (N_R + N_Z\,(i-1))}$ ed il vettore $\mathbf{v}_{\text{f}} \in \mathbb{R}^{N_R + N_Z\,(i-1)}$ ad indicare la concatenazione delle features in input al readout (i.e. encoding del reservoir più output-feedback).

Il costo per il calcolo dell'uscita della sotto-rete i-esima per tutti i grafi del dataset è dunque

$$\begin{aligned} &O(|\mathcal{G}|\ N_Z(N_R + N_Z\,(i-1)) \\ &= O(|\mathcal{G}|\ N_Z\ N_R + |\mathcal{G}|\ N_Z^2, (i-1)) \end{aligned} \tag{A.17}$$

Considerando tutte le NSN sotto-reti, si ottiene il costo complessivo:

$$\begin{aligned} &O(\sum_{i=1}^{NSN}(|\mathcal{G}|\ N_Z\ N_R + |\mathcal{G}|\ N_Z^2, (i-1))) \\ &= O(NSN\ |\mathcal{G}|\ N_Z N_R + |\mathcal{G}|\ N_Z^2 \sum_{i=1}^{NSN}(i-1)) \\ &\simeq O(NSN\ |\mathcal{G}|\ N_R + |\mathcal{G}|\ NSN^2) \end{aligned} \tag{A.18}$$

A.5 Calcolo dell'output del readout globale

Questo caso differisce dal calcolo dell'output delle sotto-reti unicamente perché l'input del readout globale è formato dai soli output-feedback, senza che vi sia alcun segnale proveniente da un reservoir.
Il costo computazionale complessivo per il calcolo degli output della rete è quindi dato da

$$O(|\mathcal{G}|\ NSN^2) \tag{A.19}$$

e scala linearmente con la dimensione del dataset e quadraticamente con il numero di sotto-reti.

A.6 Model selection di GraphESN

Assumiamo l'uso di una GraphESN con reservoir completamente connessi. Siamo interessati al costo complessivo di una model selection che preveda la variazione delle dimensioni del reservoir: $N'_R \in \{N_R, 2N_R, \ldots, NSN\ N_R\}$.

Il costo del processo di encoding di una singola rete con reservoir di dimensioni N_R è in questo caso

$$O(N_R^2\ MAXIT\ MAXV\ |\mathcal{G}|) \tag{A.20}$$

Facendo variare la dimensione del reservoir, il costo complessivo diventa

$$\begin{aligned} &O(N_R^2\ MAXIT\ MAXV\ |\mathcal{G}| \sum_{i=1}^{NSN} i^2) \\ &\simeq O(N_R^2\ MAXIT\ MAXV\ |\mathcal{G}| NSN^3) \end{aligned} \tag{A.21}$$

L'apprendimento, tramite Ridge Regression, ha invece per una singola rete costo complessivo di

$$O(N_R^3 + N_R^2 \ |\mathcal{G}|) \tag{A.22}$$

variando $N'_R \in \{N_R, 2N_R, \ldots, NSN\ N_R\}$ si ha dunque

$$\begin{aligned} &O(N_R^3 \sum_{i=1}^{NSN} i^3 + N_R^2 \ |\mathcal{G}| \sum_{i=1}^{NSN} i^2) \\ &\simeq O(N_R^3 \ NSN^4 + N_R^2 \ |\mathcal{G}| \ NSN^3) \end{aligned} \tag{A.23}$$

Infine, il calcolo dell'output per ogni input del dataset costa

$$O(N_R \ |\mathcal{G}|) \tag{A.24}$$

che nel caso di una model selection completa diventa

$$O(N_R \ |\mathcal{G}| \sum_{i=1}^{NSN} i) \simeq O(N_R \ |\mathcal{G}| NSN^2) \tag{A.25}$$

Il costo complessivo della selezione di un modello di tipo GraphESN, ottenuto variando la dimensione del reservoir $N'_R \in \{N_R, 2N_R, \ldots, NSN\ N_R\}$ è dunque:

$$\begin{aligned} O(&NSN^4 \ N_R^3 + NSN^3 \ N_R^2 \ MAXIT \ MAXV \ |\mathcal{G}| \\ &+ NSN^3 \ N_R^2 \ |\mathcal{G}| + NSN^2 \ N_R \ |\mathcal{G}|) \end{aligned} \tag{A.26}$$

B

Dettaglio dei risultati sperimentali

In questa appendice sono riportate per esteso le performance ottenute sulle singole fold nei casi in cui sia stata realizzata una double k-fold cross-validation (i.e. task dei dataset PTC e Mutagenesis). In questo caso, infatti, il processo di model selection determina un'iperparametrizzazione vincente su ogni fold (si veda il paragrafo 1.1.2).

B.1 PTC-FR

Tabella B.1: Dettaglio delle performance (in training e test) di GraphESN-CF sul task PTC-FR. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	67.1 (±0.1)	65.9 (±0.6)	66.6 (±0.2)	64.8 (±0.0)
2	68.6 (±0.1)	70.9 (±0.7)	68.5 (±0.3)	70.3 (±1.1)
3	69.0 (±0.0)	66.3 (±0.7)	69.0 (±0.0)	66.6 (±0.7)
4	67.6 (±0.0)	65.7 (±0.0)	68.6 (±0.3)	66.6 (±0.7)
5	69.5 (±0.2)	67.7 (±1.1)	69.5 (±0.1)	67.7 (±1.1)

Tabella B.2: Dettaglio delle performance (in training e test) di GraphESN-FW sul task PTC-FR. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	67.4 (±0.7)	66.2 (±0.9)	68.3 (±0.6)	67.6 (±0.9)
2	68.5 (±0.6)	66.9 (±2.1)	68.8 (±0.1)	66.9 (±2.5)
3	69.4 (±0.0)	66.9 (±0.6)	69.3 (±0.2)	66.6 (±0.7)
4	69.3 (±0.1)	66.6 (±0.7)	69.3 (±0.1)	67.1 (±0.0)
5	69.5 (±0.2)	69.1 (±0.7)	69.5 (±0.2)	67.7 (±1.5)

Tabella B.3: Dettaglio delle performance (in training e test) di GraphESN-FOF sul task PTC-FR. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	71.3 (±0.3)	68.2 (±1.7)	70.4 (±0.5)	67.0 (±3.3)
2	69.5 (±0.3)	70.9 (±1.1)	69.3 (±0.4)	70.3 (±1.7)
3	69.5 (±0.2)	66.3 (±0.7)	69.6 (±0.2)	66.9 (±0.6)
4	70.3 (±0.8)	67.4 (±1.1)	71.5 (±0.8)	66.9 (±1.4)
5	69.7 (±0.1)	68.6 (±0.9)	70.0 (±0.7)	68.3 (±0.6)

B.2 PTC-FM

Tabella B.4: Dettaglio delle performance (in training e test) di GraphESN-CF sul task PTC-FM. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	63.4 (±0.3)	68.3 (±1.7)	63.5 (±0.1)	69.4 (±1.1)
2	67.9 (±0.3)	57.1 (±0.9)	67.3 (±0.1)	56.9 (±1.1)
3	67.1 (±0.1)	59.1 (±0.7)	66.7 (±0.2)	60.3 (±0.6)
4	66.7 (±0.4)	62.9 (±0.0)	66.0 (±0.1)	63.1 (±0.6)
5	64.3 (±0.0)	66.4 (±0.6)	64.1 (±0.2)	66.4 (±0.6)

Tabella B.5: Dettaglio delle performance (in training e test) di GraphESN-FW sul task PTC-FM. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	63.9 (±0.4)	69.7 (±1.1)	63.9 (±0.3)	68.6 (±1.6)
2	67.5 (±0.1)	57.7 (±0.7)	67.7 (±0.4)	57.1 (±0.0)
3	66.5 (±0.4)	59.7 (±0.6)	67.0 (±0.4)	62.3 (±0.7)
4	65.8 (±0.8)	64.3 (±1.3)	65.5 (±0.4)	63.4 (±0.7)
5	64.0 (±0.8)	66.7 (±0.0)	64.3 (±0.5)	65.2 (±1.8)

Tabella B.6: Dettaglio delle performance (in training e test) di GraphESN-FOF sul task PTC-FM. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	63.3 (±0.4)	67.7 (±1.9)	63.5 (±0.3)	67.7 (±2.9)
2	67.7 (±0.4)	57.1 (±0.9)	66.5 (±1.7)	57.4 (±0.6)
3	67.3 (±0.3)	60.0 (±0.9)	67.1 (±0.4)	60.6 (±1.9)
4	65.5 (±0.3)	64.0 (±1.1)	65.4 (±0.4)	64.3 (±0.9)
5	64.1 (±0.6)	63.5 (±1.4)	63.9 (±0.1)	63.8 (±1.6)

B.3 PTC-MR

Tabella B.7: Dettaglio delle performance (in training e test) di GraphESN-CF sul task PTC-MR. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	58.8 (±0.1)	59.1 (±0.7)	58.8 (±0.3)	60.0 (±0.9)
2	58.9 (±0.3)	59.7 (±0.6)	58.8 (±0.1)	59.7 (±0.6)
3	61.1 (±0.3)	53.8 (±1.5)	60.9 (±0.4)	54.7 (±0.6)
4	60.2 (±0.1)	58.8 (±0.0)	60.3 (±0.2)	58.8 (±0.0)
5	58.8 (±0.4)	58.2 (±0.7)	58.0 (±0.1)	57.4 (±0.0)

Tabella B.8: Dettaglio delle performance (in training e test) di GraphESN-FW sul task PTC-MR. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	59.0 (±0.2)	60.3 (±0.6)	58.0 (±0.6)	58.0 (±1.9)
2	59.5 (±0.0)	59.4 (±0.7)	59.2 (±0.4)	59.4 (±0.7)
3	60.9 (±0.4)	55.3 (±1.5)	61.2 (±0.2)	53.5 (±3.8)
4	60.0 (±0.3)	58.8 (±0.0)	59.9 (±0.4)	58.8 (±0.0)
5	59.6 (±0.9)	57.9 (±0.7)	58.2 (±1.0)	57.4 (±1.3)

Tabella B.9: Dettaglio delle performance (in training e test) di GraphESN-FOF sul task PTC-MR. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	59.8 (±0.3)	60.3 (±1.1)	59.4 (±0.4)	59.7 (±1.1)
2	60.5 (±0.5)	54.3 (±2.0)	62.3 (±1.1)	56.9 (±2.3)
3	61.3 (±0.3)	56.5 (±1.5)	61.5 (±0.3)	55.3 (±1.5)
4	59.9 (±0.2)	58.8 (±0.0)	60.4 (±0.4)	58.5 (±1.1)
5	64.3 (±1.0)	56.2 (±1.7)	62.0 (±1.9)	56.5 (±2.7)

B.4 PTC-MM

Tabella B.10: Dettaglio delle performance (in training e test) di GraphESN-CF sul task PTC-MM. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	66.4 (±0.2)	65.9 (±1.1)	66.0 (±0.5)	66.2 (±0.9)
2	66.7 (±0.1)	60.3 (±0.0)	66.8 (±0.0)	61.2 (±0.7)
3	66.6 (±0.3)	67.2 (±0.0)	66.2 (±0.1)	67.2 (±0.0)
4	67.1 (±0.6)	66.0 (±0.6)	66.7 (±0.4)	67.2 (±0.9)
5	67.5 (±0.4)	65.8 (±0.7)	67.8 (±0.4)	67.3 (±1.2)

Tabella B.11: Dettaglio delle performance (in training e test) di GraphESN-FW sul task PTC-MM. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	66.3 (±0.5)	69.7 (±2.0)	67.6 (±1.0)	67.1 (±1.2)
2	67.2 (±0.3)	61.8 (±1.3)	67.8 (±0.5)	63.2 (±1.3)
3	68.0 (±0.6)	65.4 (±0.6)	67.7 (±1.2)	66.0 (±1.1)
4	68.6 (±0.6)	61.8 (±2.8)	68.6 (±0.8)	60.0 (±2.2)
5	68.2 (±0.7)	68.2 (±1.4)	67.6 (±0.2)	66.7 (±1.9)

Tabella B.12: Dettaglio delle performance (in training e test) di GraphESN-FOF sul task PTC-MM. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	69.9 (±1.0)	69.7 (±3.0)	67.9 (±1.2)	67.4 (±1.4)
2	66.8 (±0.4)	61.2 (±1.2)	67.5 (±1.2)	61.2 (±1.2)
3	68.6 (±1.9)	66.9 (±1.1)	69.3 (±1.7)	65.1 (±2.0)
4	70.3 (±0.5)	69.0 (±1.5)	68.3 (±1.5)	66.9 (±2.0)
5	68.8 (±0.5)	66.1 (±1.5)	67.6 (±1.1)	66.7 (±1.4)

B.5 Mutag-AB

Tabella B.13: Dettaglio delle performance (in training e test) di GraphESN-CF sul task Mutag-AB. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	86.8 (±1.0)	77.4 (±3.3)	86.2 (±0.7)	75.7 (±2.1)
2	84.3 (±0.4)	91.3 (±0.0)	84.3 (±0.8)	89.6 (±2.1)
3	85.5 (±0.3)	60.9 (±2.7)	84.3 (±1.3)	62.6 (±2.1)
4	83.2 (±0.7)	80.9 (±2.1)	83.0 (±0.2)	83.5 (±3.3)
5	85.9 (±0.7)	84.3 (±2.1)	84.8 (±1.3)	86.1 (±1.7)
6	84.2 (±0.6)	95.7 (±0.0)	82.3 (±1.6)	95.7 (±0.0)
7	84.5 (±0.3)	75.7 (±3.5)	84.0 (±1.1)	79.1 (±3.3)
8	85.8 (±0.2)	66.1 (±1.7)	85.1 (±0.8)	67.8 (±3.5)
9	82.7 (±0.9)	85.2 (±4.4)	83.6 (±0.7)	87.0 (±3.9)
10	87.0 (±0.4)	72.2 (±3.5)	83.9 (±0.7)	68.7 (±3.3)

Tabella B.14: Dettaglio delle performance (in training e test) di GraphESN-FW sul task Mutag-AB. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	87.3 (±0.5)	76.5 (±2.1)	87.6 (±0.2)	75.7 (±2.1)
2	85.1 (±0.8)	89.6 (±2.1)	85.4 (±0.8)	83.5 (±1.7)
3	87.8 (±0.4)	61.7 (±5.1)	86.7 (±2.1)	60.0 (±3.3)
4	85.5 (±0.4)	76.5 (±3.5)	85.3 (±0.4)	76.5 (±3.5)
5	85.6 (±1.5)	85.2 (±2.1)	85.8 (±1.9)	87.0 (±0.0)
6	85.0 (±0.3)	95.7 (±0.0)	84.5 (±1.2)	97.4 (±2.1)
7	85.7 (±0.8)	75.7 (±5.9)	85.4 (±0.8)	78.3 (±4.8)
8	86.7 (±0.4)	73.0 (±1.7)	85.9 (±0.6)	67.8 (±4.4)
9	86.0 (±0.3)	93.9 (±2.1)	85.3 (±0.6)	93.0 (±3.5)
10	87.0 (±0.5)	77.4 (±3.3)	87.1 (±0.2)	78.3 (±2.7)

Tabella B.15: Dettaglio delle performance (in training e test) di GraphESN-FOF sul task Mutag-AB. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	88.3 (±0.6)	78.3 (±0.0)	86.8 (±0.4)	73.9 (±0.0)
2	88.7 (±1.9)	84.3 (±3.5)	87.1 (±1.9)	80.9 (±2.1)
3	89.1 (±0.8)	75.7 (±2.1)	87.1 (±0.7)	64.3 (±3.3)
4	86.8 (±1.2)	74.8 (±5.1)	85.5 (±0.6)	77.4 (±3.3)
5	87.8 (±1.5)	82.6 (±4.8)	87.3 (±1.9)	86.1 (±1.7)
6	86.2 (±1.9)	87.8 (±5.8)	85.8 (±0.9)	89.6 (±3.5)
7	87.6 (±0.8)	74.8 (±1.7)	88.0 (±2.2)	73.0 (±1.7)
8	87.7 (±1.2)	73.0 (±1.7)	86.6 (±1.5)	67.8 (±7.6)
9	84.1 (±2.5)	87.0 (±7.8)	87.0 (±1.5)	84.3 (±9.4)
10	87.8 (±0.5)	74.8 (±3.3)	87.7 (±0.6)	70.4 (±4.3)

B.6 Mutag-AB+C

Tabella B.16: Dettaglio delle performance (in training e test) di GraphESN-CF sul task Mutag-AB+C. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	84.2 (±0.8)	80.9 (±2.1)	85.4 (±0.4)	78.3 (±0.0)
2	81.6 (±0.6)	87.0 (±0.0)	80.9 (±0.5)	77.4 (±1.7)
3	85.7 (±0.6)	63.5 (±4.4)	84.8 (±0.5)	60.0 (±1.7)
4	80.7 (±0.0)	73.9 (±0.0)	80.6 (±0.4)	73.9 (±0.0)
5	85.9 (±0.6)	75.7 (±2.1)	84.9 (±0.7)	76.5 (±2.1)
6	82.1 (±0.5)	92.2 (±3.3)	78.6 (±0.4)	94.8 (±1.7)
7	80.9 (±0.2)	78.3 (±0.0)	80.4 (±0.5)	72.2 (±2.1)
8	86.1 (±0.4)	67.0 (±2.1)	82.3 (±0.7)	70.4 (±1.7)
9	84.1 (±1.0)	87.0 (±0.0)	81.1 (±0.4)	80.9 (±2.1)
10	87.1 (±0.4)	75.7 (±3.5)	86.6 (±0.6)	75.7 (±2.1)

Tabella B.17: Dettaglio delle performance (in training e test) di GraphESN-FW sul task Mutag-AB+C. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	84.9 (±0.7)	78.3 (±0.0)	85.0 (±0.9)	78.3 (±0.0)
2	83.0 (±1.2)	83.5 (±1.7)	83.3 (±1.2)	84.3 (±2.1)
3	86.4 (±0.7)	62.6 (±3.5)	85.7 (±0.5)	60.9 (±4.8)
4	83.4 (±1.0)	78.3 (±0.0)	82.9 (±1.2)	77.4 (±1.7)
5	84.3 (±1.7)	73.9 (±0.0)	86.9 (±2.0)	75.7 (±2.1)
6	82.4 (±0.7)	94.8 (±3.3)	81.6 (±1.9)	93.9 (±4.4)
7	81.1 (±0.2)	75.7 (±3.5)	80.5 (±0.7)	80.0 (±3.5)
8	83.6 (±0.5)	67.0 (±2.1)	84.3 (±1.2)	69.6 (±2.7)
9	82.1 (±1.2)	77.4 (±4.3)	83.2 (±0.6)	73.9 (±0.0)
10	85.8 (±0.7)	71.3 (±2.1)	85.2 (±1.2)	73.0 (±3.3)

Tabella B.18: Dettaglio delle performance (in training e test) di GraphESN-FOF sul task Mutag-AB+C. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	87.5 (±1.8)	79.1 (±1.7)	86.3 (±1.8)	76.5 (±2.1)
2	87.2 (±1.5)	79.1 (±3.3)	85.3 (±2.2)	80.0 (±2.1)
3	88.4 (±0.9)	61.7 (±1.7)	88.6 (±1.3)	69.6 (±3.9)
4	82.8 (±1.3)	76.5 (±2.1)	80.7 (±0.7)	73.9 (±0.0)
5	84.0 (±0.7)	73.9 (±0.0)	83.3 (±1.1)	74.8 (±1.7)
6	84.7 (±2.5)	93.0 (±4.4)	83.1 (±0.8)	91.3 (±4.8)
7	81.0 (±0.7)	74.8 (±5.1)	81.4 (±0.5)	74.8 (±1.7)
8	84.1 (±3.2)	69.6 (±4.8)	85.3 (±1.6)	70.4 (±4.3)
9	85.7 (±1.3)	77.4 (±3.3)	79.9 (±0.6)	82.6 (±0.0)
10	85.6 (±1.3)	74.8 (±1.7)	87.0 (±2.5)	76.5 (±3.5)

B.7 Mutag-AB+C+PS

Tabella B.19: Dettaglio delle performance (in training e test) di GraphESN-CF sul task Mutag-AB+C+PS. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	85.9 (±0.5)	82.6 (±0.0)	85.5 (±0.3)	82.6 (±0.0)
2	80.7 (±0.9)	82.6 (±0.0)	81.0 (±0.2)	82.6 (±0.0)
3	84.3 (±0.7)	60.9 (±0.0)	84.6 (±0.4)	60.9 (±0.0)
4	80.3 (±0.4)	78.3 (±0.0)	83.4 (±0.5)	82.6 (±0.0)
5	86.6 (±1.0)	78.3 (±0.0)	84.8 (±0.9)	78.3 (±0.0)
6	83.5 (±0.5)	97.4 (±2.1)	79.2 (±0.4)	95.7 (±0.0)
7	82.9 (±0.2)	78.3 (±0.0)	82.6 (±0.0)	78.3 (±0.0)
8	85.4 (±0.5)	74.8 (±3.3)	84.5 (±0.4)	74.8 (±1.7)
9	84.2 (±0.4)	78.3 (±0.0)	84.5 (±0.3)	78.3 (±0.0)
10	85.6 (±0.6)	82.6 (±0.0)	86.0 (±0.5)	82.6 (±0.0)

Tabella B.20: Dettaglio delle performance (in training e test) di GraphESN-FW sul task Mutag-AB+C+PS. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	85.4 (±0.8)	82.6 (±0.0)	85.7 (±0.6)	84.3 (±2.1)
2	81.8 (±0.9)	80.0 (±2.1)	81.1 (±0.6)	80.9 (±2.1)
3	84.3 (±0.4)	61.7 (±1.7)	82.8 (±0.7)	66.1 (±1.7)
4	85.5 (±1.8)	80.9 (±3.5)	83.0 (±0.2)	81.7 (±1.7)
5	86.9 (±0.9)	78.3 (±0.0)	87.5 (±1.6)	75.7 (±2.1)
6	79.7 (±1.2)	96.5 (±1.7)	80.1 (±0.8)	96.5 (±1.7)
7	82.1 (±0.0)	78.3 (±0.0)	82.0 (±0.2)	78.3 (±0.0)
8	86.9 (±1.7)	72.2 (±2.1)	81.0 (±0.9)	79.1 (±1.7)
9	82.7 (±0.5)	83.5 (±3.3)	84.7 (±0.8)	82.6 (±3.9)
10	87.1 (±1.5)	79.1 (±3.3)	85.9 (±1.3)	80.9 (±2.1)

Tabella B.21: Dettaglio delle performance (in training e test) di GraphESN-FOF sul task Mutag-AB+C+PS. (*sx*) $N_R = 50$; (*dx*) $N_R = 30$.

	$N_R = 50$		$N_R = 30$	
Fold	TR (stdev)	TS (stdev)	TR (stdev)	TS (stdev)
1	86.6 (±1.5)	83.5 (±1.7)	84.8 (±0.2)	82.6 (±0.0)
2	82.8 (±2.4)	80.0 (±2.1)	82.6 (±1.2)	80.0 (±2.1)
3	83.7 (±0.6)	68.7 (±1.7)	85.1 (±1.7)	62.6 (±2.1)
4	85.1 (±1.9)	81.7 (±1.7)	83.1 (±1.2)	80.9 (±2.1)
5	87.1 (±0.6)	77.4 (±1.7)	86.5 (±1.0)	77.4 (±1.7)
6	80.3 (±0.9)	98.3 (±2.1)	79.5 (±0.8)	96.5 (±1.7)
7	81.8 (±0.7)	76.5 (±2.1)	82.1 (±0.7)	76.5 (±2.1)
8	86.0 (±0.8)	74.8 (±1.7)	87.1 (±2.0)	78.3 (±4.8)
9	83.3 (±0.7)	77.4 (±1.7)	85.3 (±2.7)	85.2 (±5.9)
10	85.7 (±0.9)	80.9 (±2.1)	87.1 (±1.8)	80.0 (±2.1)

Bibliografia

[1] Pierre Baldi e Gianluca Pollastri. The principled design of large-scale recursive neural network architectures–DAG-RNNs and the protein structure prediction problem. *J. Mach. Learn. Res.*, 4:575–602, December 2003.

[2] Yoshua Bengio, Paolo Frasconi, e Patrice Simard. The problem of learning long-term dependencies in recurrent networks. In *Neural Networks, 1993., IEEE International Conference on*, volume 3, pp. 1183–1188, 1993.

[3] Carlo Bertinetto, Celia Duce, Alessio Micheli, Roberto Solaro, Antonina Starita, e Maria Rosaria Tiné. Evaluation of hierarchical structured representations for QSPR studies of small molecules and polymers by recursive neural networks. *Journal of Molecular Graphics and Modelling*, 27(7):797–802, 2009.

[4] Anna Maria Bianucci, Alessio Micheli, Alessandro Sperduti, e Antonina Starita. Application of cascade correlation networks for structures to chemistry. *Appl. Intell.*, 12(1-2):115–145, 2000.

[5] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.

[6] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–314, 1989.

[7] Kenji Doya. Bifurcations of recurrent neural networks in gradient descent learning. *IEEE Transactions on Neural Networks*, 1:75–80, 1993.

[8] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[9] Scott E. Fahlman e Christian Lebiere. The Cascade-Correlation learning architecture. In *Advances in Neural Information Processing Systems 2*, pp. 524–532. Morgan Kaufmann, 1990.

[10] Chrisantha Fernando e Sampsa Sojakka. Pattern recognition in a bucket. In *Advances in Artificial Life*, Lecture Notes In Computer Science, pp. 588–597. Springer Berlin / Heidelberg, 2003.

[11] Thomas Ferrari e Giuseppina Gini. An open source multistep model to predict mutagenicity from statistical analysis and relevant structural alerts. *Chemistry Central Journal*, 4(Suppl 1):S2, 2010.

[12] Paolo Frasconi, Marco Gori, e Alessandro Sperduti. A general framework for adaptive processing of data structures. *IEEE Transactions on Neural Networks*, 9:768–786, 1998.

[13] Holger Fröhlich, Jörg K. Wegner, Florian Sieker, e Andreas Zell. Optimal assignment kernels for attributed molecular graphs. In *Proceedings of the 22nd international conference on Machine learning*, ICML '05, pp. 225–232, New York, NY, USA, 2005. ACM.

[14] Holger Fröhlich, Jörg K. Wegner, e Andreas Zell. Assignment kernels for chemical compounds. In *IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on Neural Networks*, volume 2, pp. 913–918, July 2005.

[15] Claudio Gallicchio e Alessio Micheli. Graph Echo State Networks. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN) 2010*, pp. 1–8, July 2010.

[16] Claudio Gallicchio e Alessio Micheli. TreeESN: a preliminary experimental analysis. In *Proceedings of the ESANN 2010*, pp. 333–338, d–side, 2010.

[17] Claudio Gallicchio e Alessio Micheli. Architectural and markovian factors of Echo State Networks. *Neural Networks*, 24(5):440–456, 2011.

[18] Claudio Gallicchio e Alessio Micheli. Exploiting vertices states in GraphESN by weighted nearest neighbor. In *ESANN*, 2011.

[19] Claudio Gallicchio, Alessio Micheli, e Giulio Visco. Constructive reservoir computation with output feedbacks for structured domains. In *Proceedings of the ESANN 2012*, 2012. To appear.

[20] Lowell H. Hall e Lemont B. Kier. *The Molecular Connectivity Chi Indexes and Kappa Shape Indexes in Structure-Property Modeling*, pp. 367–422. John Wiley & Sons, Inc., 2007.

[21] Barbara Hammer, Alessio Micheli, e Alessandro Sperduti. Universal approximation capability of cascade correlation for structures. *Neural Computation*, 17(5):1109–1159, 2005.

[22] Barbara Hammer, Alessio Micheli, e Alessandro Sperduti. Adaptive contextual processing of structured data by recursive neural networks: A survey of computational properties. In *Perspectives of Neural-Symbolic Integration*, pp. 67–94. 2007.

[23] Barbara Hammer e Peter Tiňo. Recurrent neural networks with small weights implement definite memory machines. *Neural Ccomputation*, 15:1897–1929, 2003.

[24] Trevor Hastie, Robert Tibshirani, e Jerome H. Friedman. *The Elements of Statistical Learning*. Springer, corrected edizione, July 2003.

[25] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Macmillan, New York, 1994.

[26] Christoph Helma, Ross D. King, Stefan Kramer, e Ashwin Srinivasan. The predictive toxicology challenge 2000-2001. *Bioinformatics*, 17(1):107–108, 2001.

[27] Georg Hinselmann, Nikolas Fechner, Andreas Jahn, Matthias Eckert, e Andreas Zell. Graph kernels for chemical compounds using topological

and three-dimensional local atom pair environments. *Neurocomputing*, 74(1-3):219–229, 2010.

[28] Guang-Bin Huang, Qin-Yu Zhu, e Chee-Kheong Siew. Extreme Learning Machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, December 2006.

[29] Herbert Jaeger. The "echo state" approach to analyzing and training recurrent neural networks. Technical Report GMD Report 148, German National Research Center for information Technology, 2001. `http://www.faculty.iu-bremen.de/hjaeger/pubs/EchoStatesTechRep.pdf`.

[30] Herbert Jaeger. Short term memory in Echo State Networks. Technical Report GMD Report 152, German National Research Center for information Technology, 2001. `http://www.faculty.iu-bremen.de/hjaeger/pubs/STMEchoStatesTechRep.pdf`.

[31] Herbert Jaeger. Reservoir riddles: Suggestions for Echo State Network research (extended abstract of invited talk). In *Proceedings of the International Joint Conference on Neural Networks 2005*, pp. 1460–1462, August 2005.

[32] Herbert Jaeger e Harald Haas. Harnessing nonlinearity: Predicting chaotic systems and saving energy in wireless communication. *Science*, 304(5667):78–80, April 2004.

[33] Herbert Jaeger, Wolfgang Maass, e Jose Principe. Editorial: Special issue on Echo State Networks and Liquid State Machines. *Neural Networks*, 20(3):287–289, 2007.

[34] Ian T. Jolliffe. *Principal Component Analysis*. Springer, second edition, October 2002.

[35] Jeroen Kazius, Ross McGuire, e Roberta Bursi. Derivation and validation of toxicophores for mutagenicity prediction. *Journal of Medicinal Chemistry*, 48(1):312–320, 2005.

[36] Enno Littmann e Helge Ritter. Learning and generalization in cascade network architectures. *Neural Comput.*, 8:1521–1539, October 1996.

[37] Mantas Lukoševičius. Echo State Networks with trained feedbacks. Technical Report No. 4, Jacobs University Bremen, 2007. `http://wwwback.jacobs-university.de/imperia/md/content/groups/research/techreports/tfbesn_iubtechreport.pdf`.

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

[39] Wolfgang Maass, Thomas Natschläger, e Henry Markram. Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Comput*, 14(11):2531–60, 2002.

[40] Mario Martelli. *Introduction to Discrete Dynamical Systems and Chaos*. Wiley, 1999.

[41] Alessio Micheli. Neural Network for Graphs: a contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

[42] Alessio Micheli, Diego Sona, e Alessandro Sperduti. Contextual processing of structured data by recursive cascade correlation. *IEEE Transactions on Neural Networks*, 15:1396–1410, 2003.

[43] Alessio Micheli, Alessandro Sperduti, Antonina Starita, e Anna Maria Bianucci. Analysis of the internal representations developed by neural networks for structures applied to quantitative structure-activity relationship studies of benzodiazepines. *Journal of Chemical Information and Computer Sciences*, 41(1):202–218, 2001.

[44] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[45] Lutz Prechelt. Investigation of the CasCor family of learning algorithms. *Neural Networks*, 10:885–896, 1996.

[46] Danil Prokhorov. Echo State Networks: appeal and challenges. In *Neural Networks, 2005. IJCNN '05. Proceedings. 2005 IEEE International Joint Conference on*, volume 3, pp. 1463–1466 vol. 3, 2005.

[47] Felix R. Reinhart e Jochen J. Steil. Reservoir regularization stabilizes learning of Echo State Networks with output feedback. In *European Symposium on Artificial Neural Networks*, pp. 59–64, April 2011.

[48] Dana Ron, Yoram Singer, e Naftali Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. In *Machine Learning*, pp. 117–149, 1996.

[49] David E. Rumelhart, Geoffrey E. Hinton, e Ronald J. Williams. *Learning internal representations by error propagation*, pp. 318–362. MIT Press, Cambridge, MA, USA, 1986.

[50] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, e Gabriele Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, January 2009.

[51] Benjamin Schrauwen, David Verstraeten, e Jan Van Campenhout. An overview of Reservoir Computing: theory, applications and implementations. In *Proceedings of the 15th European Symposium on Artificial Neural Networks*, pp. 471–482, April 2007.

[52] Frank J. Śmieja. Neural network constructive algorithms: trading generalization for learning efficiency? *Circuits Syst. Signal Process.*, 12:331–374, February 1993.

[53] Qingsong Song e Zuren Feng. Effects of connectivity structure of complex Echo State Network on its prediction performance for nonlinear time series. *Neurocomputing*, 73(10-12):2177 – 2185, 2010.

[54] Alessandro Sperduti e Antonina Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8:714–735, 1997.

[55] Ashwin Srinivasan, Stephen H. Muggleton, Ross D. King, e Michael J. E. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237, pp. 217–232, 1994.

[56] Jochen J. Steil. Backpropagation-Decorrelation: online recurrent learning with O(N) complexity. In *Proc. IJCNN*, volume 1, pp. 843–848, July 2004.

[57] Jeffrey J. Sutherland, Lee A. O'Brien, e Donald F. Weaver. A comparison of methods for modeling quantitative structure-activity relationships. *Journal of Medicinal Chemistry*, 47(22):5541–5554, 2004.

[58] Peter Tiňo, Michal Čerňanský, e Ľubica Beňušková. Markovian architectural bias of recurrent neural networks. *IEEE Transactions on Neural Networks*, 15(1):6–15, 2004.

[59] Ah Chung Tsoi e Andrew Back. Discrete time recurrent neural network architectures: A unifying review. *Neurocomputing*, 15(3-4):183–223, 1997.

[60] Vladimir Vapnik. Principles of risk minimization for learning theory. In *NIPS*, pp. 831–838, 1991.

[61] David Verstraeten, Benjamin Schrauwen, Michiel D'Haene, e Dirk Stroobandt. An experimental unification of reservoir computing methods. *Neural Networks*, 20(3):391–403, April 2007.

[62] Paul J. Werbos. Backpropagation Through Time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, August 2002.

[63] Ronald J. Williams e David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1(2):270–280, 1989.

[64] Francis Wyffels, Benjamin Schrauwen, e Dirk Stroobandt. Stable output feedback in reservoir computing using ridge regression In *Proceedings of the 18th International Conference on Artificial Neural Networks*. A cura di V. Kurkova, R. Neruda, e J. Koutnik, pp. 808–817, Prague, 2008. Springer.