Università degli Studi di Padova

Dipartimento di Ingegneria
dell'Informazione
Corso di Laurea in Ingegneria
dell'Informazione

Modelli di reti neurali: multilayer perceptron e radial basis function

Laureando:
Giacomo Da Broi

Relatore:
Prof. Luca Schenato

Anno accademico 2011/2012

Indice

1 Introduzione

1.1 Breve introduzione storica

Le reti neurali ebbero origine da un'idea di due scienziati di nome McCulloch e Pitts, la quale fu esposta in un articolo negli anni '40. Essa consisteva nel mostrare che una rete di neuroni aveva capacità di computazione, e poteva imitare qualsiasi macchina computazionale. Il primo hardware modellato come una rete neurale arrivò però solo nel 1958, grazie a Rosenblatt. Questa macchina fu chiamata Percettrone, era formata da un array di fotorecettori che facevano da input, e da dei potenziometri che venivano settati tramite l'algoritmo di apprendimento della stessa. L'algoritmo alterava i pesi nei collegamenti tra i neuroni in funzione della differenza tra l'output ottenuto e quello desiderato.
Negli anni '60 il mondo scientifico, spinto dall'entusiasmo e agitato dalla nuova scoperta, si lanciò in numerose ricerche riguardanti la computazione neurale finchè non iniziarono a comparire problemi che non potevano essere risolti da questo tipo di algoritmi. In particolare Marvin Minsky e Seymour Papert misero in evidenza che il Percettrone poteva solamente riconoscere funzioni linearmente separabili, quindi l'attrazione per la computazione neurale venne accantonata.
Negli anni '80 però, ricomparve l'interesse per questo argomento grazie al fisico Hopfield, il quale dimostrò una stretta correlazione tra le reti neurali e il *vetro di spin*. Un'altra importante innovazione fu la scoperta di algoritmi di apprendimento basati sull'errore di backpropagation, che aggiravano molte delle limitazioni del Percettrone. Infine gli scienziati avevano a disposizione elaboratori di gran lunga più potenti rispetto a quelli di dieci anni prima.
Tutto ciò fece quindi esplodere un grande interesse per la computazione neurale la cui teoria fu espressa in termini più chiari negli anni '90, ed ora molto diffusa nell'elettronica di consumo.

1.2 Introduzione alle reti neurali

Le reti neurali sono un utile strumento per risolvere vari problemi, ad esempio di *pattern recognition*, analisi dei dati e di controllo. Sono molto veloci e possono trovare una soluzione tramite un insieme di esempi.
Si studiano quindi reti di neuroni idealizzati da modelli matematici. Questo tipo di approccio è utile in vari campi, come per esempio la biologia, dato che utilizzando queste reti si può tentare di capire come funziona la computazione e la memorizzazione nei neuroni. Dal punto di vista ingegneristico invece si vorrebbe creare macchine in grado di imparare. Per quanto riguarda la memoria un approccio di tipo neurale può essere utile, poiché una memoria di tipo "a indirizzo" non è associativa, cioè se conosciamo solo un pezzo di indirizzo non possiamo risalire al dato desiderato (anche esistono alcuni algoritmi in grado di ricostruire un indirizzo se vengono perdute piccole parti dello stesso), invece il nostro cervello ha una memoria associativa, infatti se ricorda vagamente il viso di una persona può risalire a questa. La memoria biologica è anche molto più robusta, infatti è possibile risalire alle informazioni anche se ci vengono comunicate con degli errori.
Tra tutti i tipi di reti neurali che sono state studiate, le più importanti e utilizzate anche in applicazioni pratiche sono le reti *multilayer perceptron* e *radial basis function*, le quali rientrano nella classificazione di reti *feedforward*.
Una rete neurale di tipo *feedforward* può essere vista come una trasformazione non lineare, la quale è governata da alcuni parametri detti pesi, i cui valori vengono regolati da una serie di esempi dati al sistema. Il processo in cui si assegna i valori ai suddetti parametri è detto *training*, e può avvenire in modi diversi a seconda dell'algoritmo usato. Dopo aver assegnato i pesi in maniera consona, i nuovi dati potranno essere elaborati velocemente.
Il grosso vantaggio di questo tipo di approccio è la possibilità di risolvere problemi la cui soluzione tramite modelli matematici standard sarebbe molto difficile.
Le reti neurali hanno però anche alcuni svantaggi, quindi si possono utilizzare per risolvere problemi solamente in presenza di determinati requisiti:
-deve essere disponibile un grande numero di dati per il *training*, infatti una limitazione delle reti neurali è la necessità di avere una grande quantità di dati per addestrare la rete;
-è difficile trovare un modello adeguato per la soluzione del problema in questione;
-i dati devono essere analizzati in maniera rapida;
-il metodo di processo dei dati deve essere abbastanza robusto, anche in presenza di un certo livello di rumore nei segnali di input.

1.3 Cellule neuronali e modello matematico

Il neurone è un'unità cellulare costituente del tessuto nervoso, è capace di ricevere e inviare impulsi elettrici. Il cervello umano contiene circa 10^{11} neuroni, i quali servono a trasmettere informazioni alle varie parti del corpo. Il neurone è formato da una parte centrale (soma), da cui dipartono dei prolungamenti che sono detti dentriti e assoni, i primi sono gli "input" del neurone, i secondi fungono da "output" (fig.1).
I collegamenti fra neuroni sono effettuati dalle sinapsi, le quali trasmettono gli impulsi tra gli assoni e gli alberi denditrici, e collegano una cellula a molte altre. Nel cervello sono quindi presenti un grandissimo numero di collegamenti possibili, che riescono a dare a un essere umano una potenza di calcolo elevatissima.

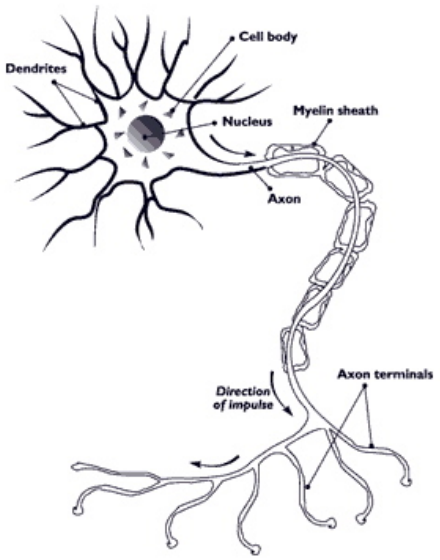

fig 1: Schema del neurone

L'impulso elettrico si trasmette lungo l'assone. La depolarizzazione della membrana innesca un processo fisiologico che determina il rilascio di neurotrasmettitori nello spazio sinaptico, permettendo la comunicazione fra cellula e cellula. La natura eccitatoria o inibitoria, dipende dal neurotrasmettitore e quindi cresce o decresce la probabilità che il neurone seguente invii un impulso. Ogni sinapsi ha associato un peso che determina la forza dell'impulso proviente dal neurone precedente, quindi possiamo vedere come input di una cellula neuronale la somma pesata dei segnali provenienti dagli assoni delle altre cellule.

Da questa struttura biologica si può modellizzare i neuroni come delle funzioni non lineari che trasformano degli input $(x_1, x_2, ..., x_i, ...x_n)$ in un output y. Come nelle cellule biologiche a ogni input è associato un peso $(w_1, ...w_n)$, in più spesso c'è un altro parametro w_0 detto bias, il quale può essere visto come il peso dell'input x_0 che viene posto costante a 1. Possiamo ora definire l'attivazione del neurone come la somma pesata dei vari input:

$$a = \sum_{k=i}^{N} w_i x_i + w_0 \tag{1.1}$$

oppure fissando $x_0 = 1$ possiamo avere una formula più compatta:

$$a = \sum_{i=0}^{N} w_i x_i \tag{1.2}$$

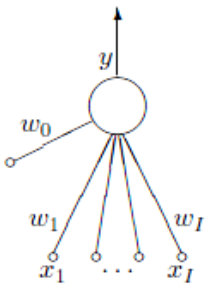

fig 2: Modello matematico del neurone

I pesi possono essere anche negativi, a seconda della sinapsi (che come detto può essere sia eccitatoria che inibitoria). Il modello non è completo solo con l'attivazione, è presente anche una funzione di attivazione $y = f(a)$, che da l'output finale del neurone. Si può scegliere questa funzione in molti modi:

-funzione lineare:

$$a = \sum_{i=0}^{N} w_i x_i \tag{1.3}$$

-funzione sigmoidea nella forma di funzione logistica, quindi di equazione:

$$y(a) = \frac{1}{1 + e^{-a}} \tag{1.4}$$

-funzione tangente iperbolica:

$$y\,(a) = tanh\,(a) \tag{1.5}$$

-funzione soglia:

$$y\left(a\right)=\begin{cases}1, & \text{se } a>0\\ -1, & \text{se } a\leq 0(1.6)\end{cases}$$

Viene un usata spesso la funzione sigmoidea perché garantisce la non linearità e fa rimanere i segnali entro un certo intervallo. Più specificatamente la funzione logistica è molto usata perché possiede una relazione interessante cioè che $y(a)' = y(a)(1-y(a))$. Invece la tangente iperbolica, che è riscrivibile come $tanh = \frac{e^a - e^- a}{e^a + e^{-a}}$ possiede la relazione $y(a)' = 1 + y(a)^2$.

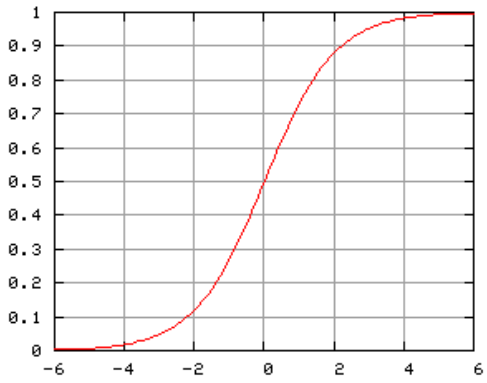

fig 3: Funzione logistica

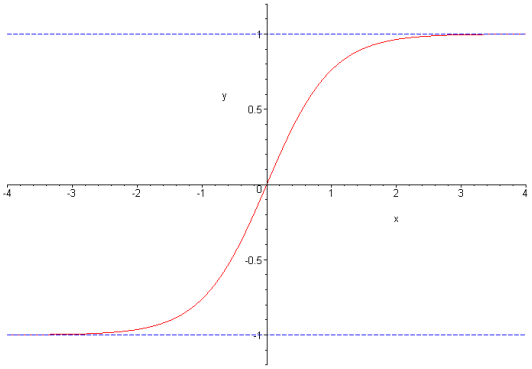

fig 4: Tangente iperbolica

Mettendo assieme varie unità rappresentate con questo modello è possibile rappresentare una classe molto ampia di funzioni, che possono essere usate per risolvere vari problemi. Tutto ciò, unito a un algoritmo di apprendimento che modifica i pesi in maniera consona, da vita alla rete neurale.

2 Reti neurali artificiali

2.1 Multilayer perceptron

2.1.1 Architettura della rete

Il *perceptron* (o percettrone in italiano) è una rete formata da un numero m di neuroni. Se consideriamo d come il numero di input, l'output di questo tipo di rete sarà quindi dato da:

$$y_j = y\left(\sum_{i=0}^{d} w_{ji}x_i\right) \tag{2.7}$$

dove x_i sono gli input e w_{ji} sono i pesi di ogni input combinati con ogni output. Con questa architettura spesso si usano funzioni di attivazione del tipo funzione soglia. Però questo tipo di rete è abbastanza limitante, quindi vengono introdotti altri livelli di neuroni e viene sostituita la funzione di attivazione con la funzione logistica (altrimenti sarebbe difficile trovare un algoritmo di apprendimento). In questo modo, essendo la funzione logistica differenziabile, si possono usare le regole del calcolo differenziale nell'algoritmo di apprendimento, e si da quindi vita a una rete detta *multilayer perceptron*. Per esempio prendiamo in esame una rete formata da due livelli di unità di elaborazione, sempre con d ingressi e con m uscite per il primo strato, ma aggiungiamo anche c uscite al secondo livello. Le unità del secondo livello sono dette unità nascoste, poiché le loro funzioni di attivazione non sono direttamente accessibili dall'esterno. Gli output finali della rete sono quindi dati da:

$$z_k = z\left(\sum_{j=0}^{m} w'_{kj}y_j\right) \tag{2.8}$$

dove z_k è l'output finale, w'_{kj} sono i pesi per ogni unità di elaborazione e y_j il segnale inviato dalle unità nascoste. Il *bias* sia nel caso del *single layer*, che nel caso del *multilayer* l'abbiamo come coefficiente rispettivamente dell'ingresso

x_0 e y_0 posti uguali a 1. La funzione di questo termine è importante, perché ci assicura che la mappa della rete non sarà lineare. Unendo quindi le equazioni sopracitate possiamo arrivare al risultato finale:

$$z_k = z\left(\sum_{j=0}^{m} w'_{kj} y\left(\sum_{i=0}^{d} w'_{ji} x_i\right)\right) \tag{2.9}$$

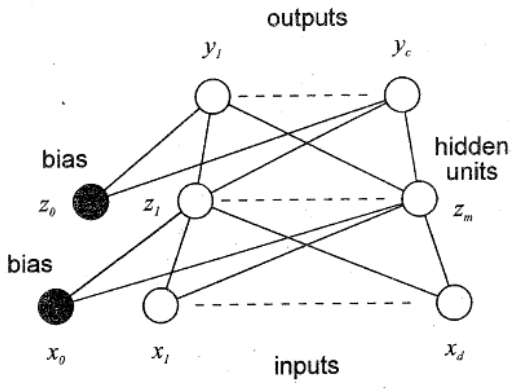

fig 5: Multilayer perceptron a 2 livelli

Per quanto riguarda le funzioni di attivazione se fossero entrambe lineari potremmo solo effettuare un prodotto tra matrici, quindi la funzione delle unità nascosta viene presa non lineare e anche differenziabile, così da aprire molte possibilità per l'utilizzo della rete. Quindi, per i vantaggi esposti nel capitolo precendente, viene usata la funzione sigmoidale sia nella forma *tanh*, sia come funzione logistica. La funzione di attivazione dell'uscita invece di più libera scelta, può essere sia lineare che non lineare a seconda dell'utilizzo della rete.
Una rete a più livelli ha una capacità di approssimazione molto più ampia di quella a un livello solo, infatti con un numero sufficiente di unità nascoste possiamo rappresentare una qualsiasi funzione continua sul dominio delle variabili d'ingresso.

2.1.2 Algoritmo di apprendendimento del MLP

Nell'addestramento di una rete neurale il nostro obiettivo è, dato un insieme di vettori di input x e dei target t, far imparare alla nostra rete la relazione da cui sono legati *input* e *output*. Così facendo se introduco (dopo l'addestramento) un input alla rete, l'output sarà molto vicino al target.
Per capire come funziona l'algoritmo di apprendimento del MLP introduciamo prima lo spazio dei pesi, che è lo spazio di dimensione dei parametri

della rete. Questo spazio ha dimensione n, dove n è il numero dei pesi nella rete. Sugli assi cartesiani ci sono appunto i pesi $(w_1, w_2, ..., w_n)$, e quindi a ogni punto del piano corrisponde una funzione della rete ben precisa. Infatti modificando i pesi si cambia anche la mappa della rete.
Introdotto lo spazio dei pesi possiamo prendere in esame la funzione d'errore, la quale è una funzione appartenente allo spazio dei pesi che misura quanto la rete è affidabile nella risoluzione del problema in esame. Il compito dell'algoritmo di apprendimento è quello di minimizzare questa funzione, quindi di trovare il punto nello spazio dei pesi in cui la funzione ha il punto di minimo globale, oppure in alcuni casi può bastare anche un punto di minimo locale.
Per trovare il minimo di una funzione, il metodo più naturale è utilizzare il calcolo differenziale, quindi derivare la funzione d'errore rispetto ai pesi trovando il vettore gradiente $\nabla E(w)$ e agire iterativamente su di esso. Consideriamo ora un vettore di *input* $(x_1^q, x_2^q, ...x_d^q)$ ed un vettore di *target* t^q. Prendiamo quindi come errore la somma dei quadrati dei residui (che è una delle funzioni più utilizzate). Il residuo è definito come $r_{qk} = y_k(x^q, w) - t^q$, quindi la funzione d'errore sarà espressa in questo modo:

$$E = \frac{1}{2} \sum_{q=1}^{n} \sum_{k=1}^{c} \{y_k(x^q, w) - t^q\}^2 \tag{2.10}$$

Adesso che abbiamo a disposizione la funzione d'errore, data la derivata rispetto ai pesi che vanno dal livello nascosto all'*output*, essa può essere scritta nella forma:

$$\frac{\partial E^q}{\partial w'_{kj}} = \frac{\partial E^q}{\partial a'_k} \frac{\partial a'_k}{\partial w'_{kj}} \tag{2.11}$$

dove a'_k è l'attivazione del *layer* finale quindi

$$a'_k = \sum_{j=0}^{m} w'_{kj} z_j \tag{2.12}$$

Ora definiamo

$$\delta'_k = \frac{\partial E^q}{\partial a'_k} \tag{2.13}$$

quindi l'equazione (2.11) diventa

$$\frac{\partial E^q}{\partial w'_{kj}} = \delta'_k z_j \tag{2.14}$$

Combinando le equazioni (2.11), (2.12) e (2.13) ottengo

$$\delta'_k = z\left(a'_k\right)\left\{y_k - t_k\right\} \tag{2.15}$$

Dato che la (2.15) è una differenza tra valori di *output* e di *target*, quindi viene chiamata errore.
Per quanto riguarda il primo livello di unità di elaborazione, iniziamo col scrivere l'attivazione delle unità nascoste

$$z_j = g(a_j), a_j = \sum_{i=0}^{d} w_{ji} x_i \tag{2.16}$$

Operando in modo analogo al caso precedente otteniamo

$$\frac{\partial E^q}{\partial w'_{ji}} = \delta_k x_i \tag{2.17}$$

Possiamo trovare un'espressione per δ_j

$$\delta_j = \frac{\partial E^q}{\partial a_j} = \sum_{k=1}^{c} \frac{\partial E^q}{\partial a'_k} \frac{\partial a'_k}{\partial a_j} \tag{2.18}$$

Unendo le equazioni (2.12), (2.13) e (2.16) otteniamo

$$\delta_j = g'\left(a_j\right) \sum_{k=1}^{c} w'_{kj} \delta'_k \tag{2.19}$$

Sostanzialmente l'errore all'unità di processo j, è data dalla somma degli errori δ'_k negli *output* moltiplicati per il loro peso, da qui il nome backpropagation error.

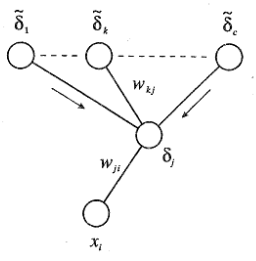

fig 6: Diffusione del backpropagation error

Quindi riassumendo quanto detto finora, i passi per valutare la derivata della funzione d'errore sono:

-per ogni *input* e *target* nell'insieme di dati che abbiamo a disposizione, dobbiamo calcolare l'attivazione delle unità nascoste e di *output*, usando rispettivamente le equazioni (2.16) e (2.12);
-trovare poi l'errore con l'equazione (2.15);
-valutare l'errore delle unità nascoste con l'equazione (2.18);
-valutare la derivata della funzione d'errore per la coppia in questione con le equazioni (2.17) e (2.14);
-ripetere il procedimento per tutti i dati che si hanno a disposizione.
Ora che abbiamo a disposizione il vettore gradiente, siamo in grado di minimizzare la funzione d'errore. Per iniziare bisogna scegliere un punto nello spazio dei pesi da cui partire (potrebbe essere scelto anche a caso), in seguito viene aggiornato il vettore dei pesi muovendosi a piccoli passi nella direzione in cui il gradiente diminuisce più rapidamente. Iterando questo procedimento troveremo una sequenza di vettori dei pesi, le cui componenti saranno date da

$$w_{kj}^{\tau+1} = w_{kj}^{\tau} - \eta \frac{\partial E}{\partial w_{kj}}_{w^{\tau}} \tag{2.20}$$

dove η è un parametro che è detto fattore di apprendimento. Teoricamente quindi, iterando questo procedimento dovremmo trovare una sequenza di pesi che converge in un punto in cui la funzione d'errore risulta avere un minimo. η è un parametro delicato, poiché se viene preso troppo piccolo le variazioni saranno molto lente e quindi l'ottimizzazione avverrà in maniera poco efficiente. Ma se viene preso molto grande possono aver luogo oscillazioni divergenti che ci impediscono di portare a termine il procedimento con successo.

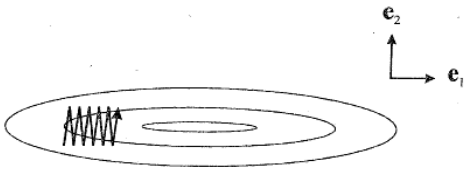

fig 7: Un esempio di oscillazioni nella ricerca del minimo in una superficie d'errore

L'ideale sarebbe poter cambiare il fattore di apprendimento a seconda della situazione in cui ci troviamo, e ciò è possibile introducendo un parametro nuovo μ, detto Momento, nell'equazione

$$\Delta w_{ji}^{\tau} = -\eta \frac{\partial E}{\partial w_{kj}}_{w^{\tau}} + \mu_{ji}^{\tau-1} \tag{2.21}$$

Il parametro μ si può cambiare man mano ed è una costante nell'insieme $0 < \mu \leq 1$. La funzione esatta di questo parametro è aumentare il fattore di apprendimento quando nella funzione si presenta una bassa curvatura, e di diminuirlo nelle zone di alta curvatura in modo da evitare oscillazioni.
Per un metodo alternativo possiamo anche tenere conto che l'errore è la somma di vari termini corrispondenti al *traning set*, quindi al posto di procedere direttamente sulla funzione d'errore complessiva, possiamo agire separatamente sui diversi *pattern*.
Per quanto riguarda la conclusione dell'algoritmo non ci sono metodi fissi per decidere quando bloccarlo, dato che le funzioni d'errore non lineari possono essere molto complesse. Vengono utilizzati vari criteri per fare ciò, per esempio bloccare l'algoritmo dopo un certo numero di iterazioni, dopo un certo tempo d'esecuzione oppure l'errore supera un certa soglia.

2.2 Radial basis function (RBF)

2.2.1 Architettura della rete

L'architettura RBF è una tipologia di rete alternativa alla multilayer perceptron, la quale si basa sull'idea di una funzione y(x) che può essere ottenuta dalla combinazione di lineare di varie funzioni base. Sostanzialmente abbiamo N funzioni base corrispondenti ognuna a un punto del nostro insieme di dati, la funzione in questione sarà della forma $\phi(\|x - x^q\|)$. Quindi l'*output* della mappa sarà come detto una combinazione lineare delle suddette funzioni

$$y_k = \sum_{q=1}^{n} w'_{kq} \phi_q(x) \tag{2.22}$$

La funzione ϕ si può scegliere in molti modi, la scelta più comune è la gaussiana quindi

$$\phi_k = exp\left(-\frac{\|x - x^q\|^2}{2\sigma^2}\right) \tag{2.23}$$

dove σ^2 è la varianza della gaussiana. Questo tipo di rete da un'interpolazione esatta, quindi si richiede che il *target* sia uguale alla mappa dei valori di *input*, quindi:

$$y_k(x^q) = t_k^q, k = 1, ..., n \tag{2.24}$$

Nelle applicazioni però non siamo interessati a un'interpolazione esatta, dato che in questo modo spesso si ottengono funzioni fortemente oscillatorie, le quali sono da evitare. Siamo quindi interessati a un'interpolazione approssimata. Quindi basta prendere un sottoinsieme $m < n$ di funzioni presenti nel nostro set di dati, quindi la nostra uscita sarà

$$y_k = \sum_{j=1}^{m} w'_{kj}\phi_j(x), m < n \tag{2.25}$$

anche in questo caso le funzioni ϕ saranno gaussiane di media μ_j e varianza σ^2, e anche in questo tipo di rete è presente un bias, che coincide con il coefficiente di $\phi_0 = 1$.

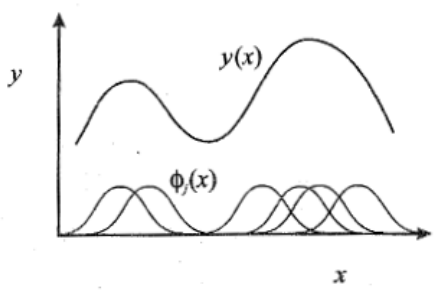

fig 8: La funzione y approssimata come combinazione lineare di funzioni

Questo tipo di rete può essere rappresentata con livello di input $(x_1, ..., x_d)$, uno strato nascosto rappresentato dalle funzioni di base $(\phi_1, ...\phi_m)$ e l'uscita con una funzione d'attivazione lineare.

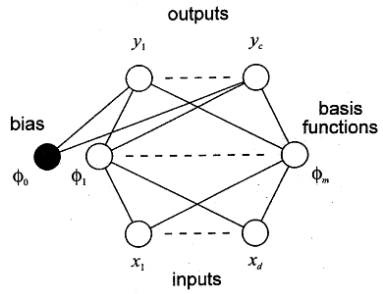

fig 9: Architettura tipo di una rete radial basis function

Anche qui la rete può approssimare (scegliendo un numero opportuno di unità nascoste) una qualsiasi funzione continua, e per addestrarla si possono usare algoritmi simili a quelli del *multilayer perceptron*. Con questa architettura però possiamo anche avere un diverso approccio, cioè possiamo prima ottimizzare le funzioni base, e in seguito i pesi nella rete.

2.2.2 Addestramento della rete

Come detto precedentemente, l'addestramento di una rete di RBF può essere affrontato con un approccio diverso rispetto a una MLP. Decidiamo di distinguere nettamente i ruoli del primo e del secondo strato, così da dividere il processo di apprendimento in due parti: la prima consiste nel determinare i parametri della funzione di base, la seconda di trovare (con le funzioni base già fissate) i pesi del secondo strato.
Per la stima dei parametri delle funzioni base ci sono vari metodi. Il più diffuso consiste nel porre i centri μ_j delle funzioni base uguali a un sottoinsieme di punti appartenenti ai dati di addestramento, e porre le varianze uguali per esempio alla distanza media tra i punti del suddetto sottoinsieme.
Un approccio più generale può essere dato dall'algoritmo K-means, il quale prevede di avere un numero N di dati per l'addestramento che vogliamo rappresentare tramite un numero K di vettori. L'algoritmo divide i dati x^n in K sottoinsiemi S_j, i quali contengono N_j punti. Il vettore associato a questi sottoinsieme, avrà media

$$\mu_j = \frac{1}{N_j} \sum_{n_j} x^n \tag{2.26}$$

L'algoritmo parte inizializzando i centri μ_j delle funzioni eguagliandole a un sottoinsieme di dati. In seguito ogni vettore appartente al set di dati viene associato alla funzione base con il centro più vicino, il valore del quale viene sostituito con la media tra i vettori associati alla funzione. Il processo continua iterativamente fino a che non converge.
Un approccio di tipo diverso si basa sul riconoscere che le funzioni base possono essere viste come componenti di un *mixture density model*, e quindi è possibile ottimizzarle con l'algoritmo *maximum likelihood*. Prendiamo quindi le funzioni base per rappresentare la densità di probabilità di un vettore di *input*, e poi con i dati per l'addestramento possiamo determinare i parametri delle funzioni. La densità di probabilità è equivalente a una combinazione lineare delle funzioni base:

$$p(x) = \frac{1}{m} \sum_{j=1}^{m} \frac{1}{(2\pi)^{d/2} \sigma_j^d} \phi_j(x) \tag{2.27}$$

dove i fattori che precedono $\phi_j(x)$ assicurano che il risultato dell'integrale sulla densità di probabilità abbia risultato unitario. Se i vettori di *input* per l'addestramento vengono introdotti in maniera indipendente, avremo luogo a una funzione *likelihood* del tipo

$$L = \prod_{q=1}^{n} p(x^q) \tag{2.28}$$

I parametri per le funzioni base sono quindi ottenuti massimizzando questa funzione. Per farlo esistono vari algoritmi che risolvono il problema con una velocità ragionevole.
Per quanto riguarda invece la determinazione dei pesi del secondo livello, consideriamo come già fissati i parametri delle funzioni base e scriviamo la funzione d'errore in modo analogo a quello usato nel MLP

$$E = \frac{1}{2} \sum_{q=1}^{n} \sum_{k=1}^{c} \left\{ y_k \left(x^q, w \right) - t^q \right\}^2 \tag{2.29}$$

sostituendo (2.25) in (2.29) e derivando rispetto ai pesi, possiamo minimizzare la funzione errore

$$0 = \sum_{q=1}^{n} \phi_j^q \left(\sum_{j'=1}^{m} w'_{kj'} \phi_j'^q - t_k^q \right) \tag{2.30}$$

scrivendo il tutto in forma matriciale avremo

$$0 = \Phi^{\mathrm{T}} \left\{ \Phi W^{\mathrm{T}} - T \right\} \tag{2.31}$$

in cui le matrici Φ conterranno ϕ_j^q. La matrice dei pesi W sarà composta da $w'_{kj'}$ e la matrice dei *target* da t_k^q. Noi siamo interessati a trovare i pesi che minimizzano la funzione errore, quindi dall'equazione (2.26) possiamo trovare

$$W^{\mathrm{T}} = \left(\Phi^{\mathrm{T}} \Phi \right)^{-1} \Phi^{\mathrm{T}} T \tag{2.32}$$

Quindi dopo aver trovato i parametri delle funzioni base, siamo in grado di risolvere questo tipo di sistema e trovare i pesi in maniera esatta. Questo metodo corrisponde all'interpolazione esatta, la quale non è desiderata. Infatti, come è stato segnalato in precendenza, nella pratica si prende m molto minore di n.

3 Ottimizzare la rete

3.1 Processo di generalizzazione

L'obiettivo finale di una rete neurale non è tanto trovare un'esatta predizione dei dati di addestramento, ma di determinare un buona predizione riguardante i nuovi dati in *input*. Spesso se mettiamo pochi coefficienti a disposizione daremo luogo a un'interpolazione molto povera, invece se ne interpoliamo troppi rischiamo di prendere in considerazione anche il rumore e quindi di avere scarsi risultati.
Alzando infatti di troppi ordini il grado del polinomio con cui interpoliamo i dati, rischiamo di trovarci davanti a un fenomeno di *overfitting* (fig. 10), il quale ha funzione d'errore uguale a zero, ma da risultati molto scarsi riguardo la predizione di nuovi dati. Sostanzialmente diminuendo man mano i gradi di libertà del suddetto polinomio la funzione d'errore riguardante i dati di addestramento andrà sempre più vicina allo zero, ma quella riguardante i dati di test a un certo punto tenderà a risalire. I gradi di libertà sono in relazione al numero di pesi cioè al numero di unità nascoste della rete, quindi il nostro obiettivo è quello di trovare il numero ottimo di unità nascosta in modo da avere le performance migliori con i dati di test.
Per misurare le prestazioni di una rete dobbiamo quindi dividere i dati in due insiemi indipendenti, i quali sono detti insiemi di test e di addestramento, cosicchè possiamo verificare come varia l'errore sia sull'apprendimento che su nuovi dati introdotti successivamente. Solo in questo modo possiamo trovare il numero m di unità nascoste migliore per la rete, dobbiamo fare in modo che la rete abbia abbastanza flessibilità, ma non dia luogo a fenomeni di *overfitting*. La rete per funzionare in modo ottimale deve avere un errore basso per quanto riguarda i dati di test.

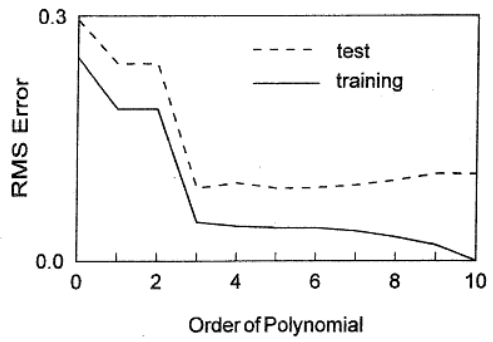

fig 10: Andamento dell'errore quando ha luogo overfitting

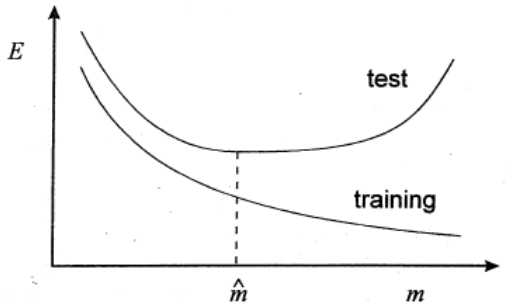

fig 11: Errore in funzione del numero di unità nascoste, m il numero di unità ottimo

Se abbiamo a disposizione un quantità di dati limitata, c'è un metodo detto *cross validazione* che funziona in questo modo: dividiamo in N sottoinsiemi i dati a nostra disposizione, un sottoinsieme viene utilizzato per il *training* e gli altri per il test. Ripeto questo procedimento per tutte le combinazioni di sottoinsieme e scelgo per l'addestramento la suddivisione che mi da errore minore sul test.

3.2 Data processing

Utilizzare nelle reti neurali gli *input* nella forma in cui ci vengono dati è un procedimento che non dà grandi risultati. Spesso siamo costretti a trasformarli in una nuova rappresentazione, migliorando di molto le prestazioni della rete. Il procedimento di modifica della rappresentazione dei dati in *input* è detto *preprocessing*, la modifica dell'*output* per farli tornare allo stato originale è detto *postprocessing* (fig. 12).

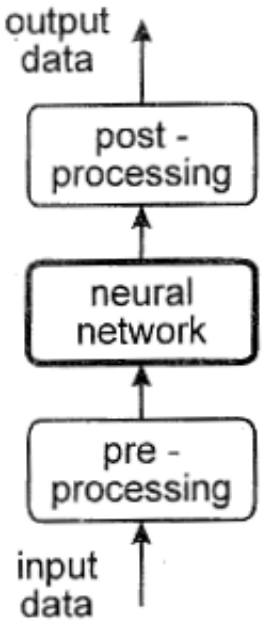

fig 12: Processing sui dati in ingresso e in uscita

Un esempio di *preprocessing* lo troviamo in alcuni casi, quando conviene paradossalmente eliminare alcune variabili di *input* per avere dei risultati migliori. Questo può essere spiegato pensando che se ho un numero maggiore di variabili ho certamente una maggiore precisione, ma l'elaborazione di ogni dato richiede un tempo maggiore rispetto ad un dato meno preciso. Talvolta aggiungere dati che sono fortemente correlati con quelli che già possediamo ci da una bassa quantità di informazione addizionale. Questo tipo di processo, che ci fa prendere in considerazione solo un sottoinsieme dei dati a nostra disposizione, viene detto di estrazione delle *features*, le quali sono delle combinazione (spesso non lineari) tra le variabili di ingresso, che vengono scelte in modo da rendere la rete più semplice.
Un altro data *preprocessing* molto utilizzato è il *rescaling lineare*, cioè il classico procedimento che viene utilizzato per elaborare i dati di ingresso in modo che si abbia media nulla e varianza unitaria. Di conseguenza gli *input* della rete saranno

$$x_i' = \frac{\{x_i - \quad \hat{x}_i\}}{s_i} \tag{3.33}$$

ed in questa formula poniamo

$$\hat{x}_i = \frac{1}{n}\sum_{q=1}^{n} x_i, \qquad s_i^2 = \frac{1}{n-1}\sum_{q=1}^{n}\{x_i - \quad \hat{x}_i\}^2 \tag{3.34}$$

Questo procedimento deve essere ripetuto (anche dopo il processo di addestramento) con i nuovi dati che vengono presentati alla rete.

4 Simulazioni

4.1 Classificazione

In questa prima simulazione verrà trattato un problema di classificazione, cioè insegnare alla rete neurale da noi creata a catalogare nella giusta categoria i dati che le si presentano. Per svolgere questa simulazione è stato utilizzato un insieme di dati presenti nella banca dati di *MatLab*, il quale è denominato *glass_dataset*. Questo dataset è composto da 214 esempi di vetro dei quali è specificata la composizione chimica, cioè:

-indice di rifrazione;
-la percentuale nel peso di:
-sodio;
-magnesio;
-alluminio;
-silicone;
-potassio;
-calcio;
-bario;
-ferro;

L'obiettivo della simulazione è costruire una rete neurale che possa catalogare gli esempi di vetro che vengono dati in input, in vetri adatti per costruire finestre o vetri inadatti a questa finalità.
Si inizia quindi importando i dati che ci servono

load glass_dataset

in seguito inizializziamo gli input e i target prendendoli dal dataset

[x,t]=glass_dataset;

si procede con la creazione della rete, le cui funzioni di attivazione saranno per entrambi i livelli la funzione logistica e, scegliendo come algoritmo di apprendimento la discesa del gradiente con l'utilizzo del momento, e in seguito sarà possibile fissarne i parametri

net=patternnet(10,'traingdm'); % viene creata una rete con 10 unità nascoste e con l'algoritmo di addestramento suddetto

net.trainParam.lr=0.02; % viene fissato il tasso d'addestramento

net.trainParam.mc=0.5; % viene fissato il momento

net.trainParam.epochs=1000; % viene scelto il numero di cicli

net.divideParam.trainRatio=0.6;

net.divideParam.valRatio=0.25;

net.divideParam.testRatio=0.15; % viene scelta la percentuale di dati che è destinata rispettivamente all'addestramento, alla validazione e al test

[net,tr]=train(net,x,t);% procedo con l'addestramento

Al termine del test questa rete però non da i risultati sperati, dato che nella fase di test vengono catalogati in maniera esatta solamente 87.5% degli *input*.

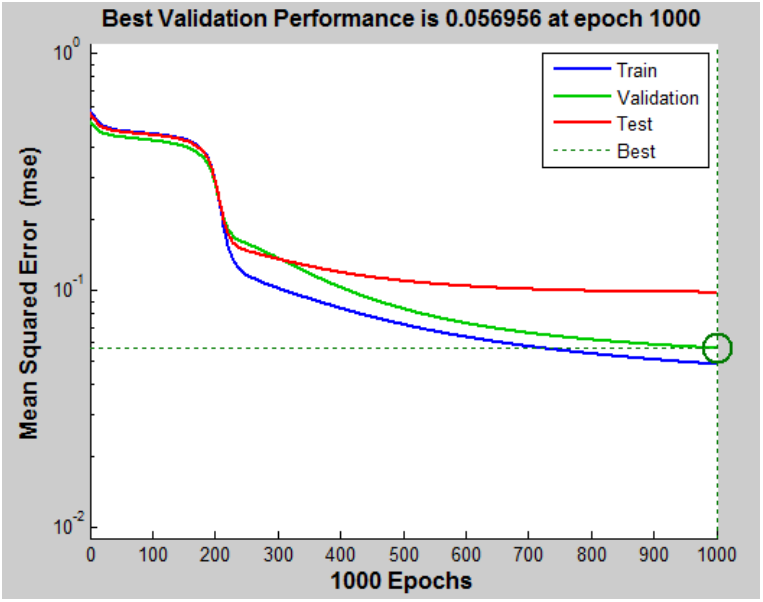

fig 13: Andamento dell'errore nella prima simulazione

Dato che il processo di minimizzazione del gradiente non è stato portato a termine nei 1000 cicli prefissati si è quindi aumentato il tasso di apprendimento per rendere il processo più veloce, di dover aumentare le unità nascoste e i dati usati per l'addestramento per rendere la rete più precisa.
Quindi si crea una nuova rete stavolta con 20 unità nascoste

net1=patternnet(20,'traingdm');

net1.trainParam.lr=0.05 % il tasso di apprendimento viene incrementato

net.trainParam.mc=0.5; % il momento viene lasciato invariato

net.trainParam.epochs=1000; % anche per il numero cicli il valore è invariato

net.divideParam.trainRatio=0.7;

net.divideParam.valRatio=0.15;

net.divideParam.testRatio=0.15; % viene cambiata anche la divisione dei dati

Con questa configurazione di parametri la rete è molto più efficiente, infatti riesce a catalogare in maniera esatta il 96.9% degli *input* di test.

Possiamo infine qui sotto esaminare l'andamento della funzione d'errore che raggiunge un valore più basso rispetto alla precedente (fig. 14).

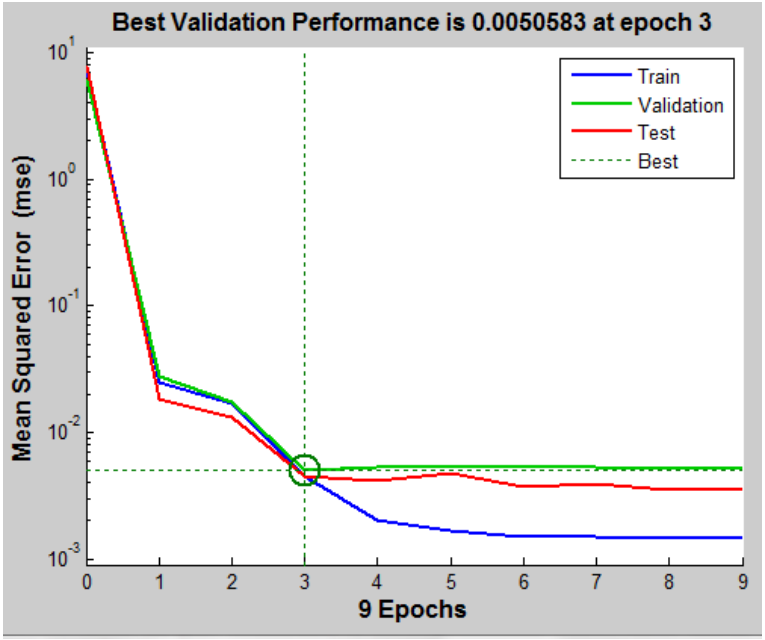

fig 14: Andamento dell'errore nella seconda simulazione

4.2 Regressione e introduzione all'algoritmo di Levenberg-Marquardt

Per risolvere i problemi di fitting di una funzione esiste un algoritmo il quale è particolarmente efficiente in queste applicazioni, ed è detto algoritmo di Levenberg-Marquardt.
Per spiegare questo algoritmo si scrive la funzione d'errore in questo modo

$$E = \frac{1}{2}\sum_n (\epsilon^n)^2 = \frac{1}{2}\sum_n \|\epsilon\|^2 \tag{4.35}$$

dove ϵ^n è l'errore nell'n-esimo *pattern*.
Supponiamo ora di volerci spostare nello spazio dei pesi trovando un punto in cui l'errore ha un valore minore. Chiamiamo il punto appena lasciato w_o, e il nuovo punto raggiunto w_n. Se lo spostamento effettuato è abbastanza piccolo possiamo effettuare uno sviluppo in serie di Taylor

$$\epsilon\,(w_n) = \epsilon\,(w_o) + Z\,(w_n - w_o) \tag{4.36}$$

dove Z è una matrice del tipo

$$(Z)_{ni} = \frac{\partial(\epsilon^n)}{\partial(w_i)} \tag{4.37}$$

sostituendo la (4.36) nella (4.35) si ottiene la funzione d'errore espressa in questo modo

$$E = \frac{1}{2}\|\epsilon(w_o) + Z(w_n - w_o)\|^2 \tag{4.38}$$

minimizzando l'errore derivando rispetto il nuovo peso w_n

$$w_n = w_o - \left(ZZ^T\right)^{-1} Z^T \epsilon(w_o) \tag{4.39}$$

Ottenuta questa formula possiamo applicarla iterativamente per minimizzare l'errore, però questo tipo di approccio può dare dei problemi poiché il passo dato dalla formula (4.39) potrebbe diventare troppo grande e farebbe perdere di significato l'approssimazione effettuata nella formula (4.38). Quindi il procedimento si propone di minimizzare la funzione d'errore cercando di mantere un passo abbastanza piccolo in ogni iterazione, in modo che l'approssimazione di Taylor non perda significato. Possiamo quindi introdurre una funzione d'errore modificata in questo modo

$$E' = \frac{1}{2}\|\epsilon(w_o) + Z(w_n - w_o)\|^2 + \lambda\|w_n - w_o\|^2 \tag{4.40}$$

procedendo con la minimizzazione dell'errore come in precedenza, il vettore del passo successivo sarà

$$w_n = w_o - \left(Z^T Z + I\lambda\right)^{-1} Z^T \epsilon(w_o) \tag{4.41}$$

Durante l'esecuzione dell'algoritmo il valore di λ deve cambiare in maniera appropriata, in modo da non far perdere significato all'approssimazione lineare della funzione d'errore. Spesso si parte con un valore arbitrario di λ, la quale si pone uguale a 0.1. Se dopo un'iterazione l'errore decresce si diminuisce λ di un fattore 10, al contrario se aumenta il parametro viene aumentato sempre di un fattore 10 e tornando al vettore di pesi precedente e si riprova ad applicare la formula. Questo metodo viene applicato iterativamente fino a raggiungere una diminuzione soddisfacente dell'errore.
Possiamo ora applicare questo algoritmo per risolvere un semplice problema di *fitting* di una funzione *coseno*. Per iniziare dobbiamo inserire gli *input*

*x=-2*pi:0.16:4*pi;* % prendiamo i valori da -2π a 4π a intervalli di 0.16

*t=cos(x)+0.05*randn(size(x));* % i target saranno come da consegna i coseni degli input a cui vanno sommati una componente di rumore gaussiano

net=fitnet(15,'trainlm'); % viene creata una rete neurale che risolva il problema di fitting

net.trainParam.mu=0.1;

net.trainParam.mu_dec=0.1;

net.trainParam.mu_inc=10; % vengono fissati i parametri dell'algoritmo di Levenberg-Marquardt, rispettivamente il λ iniziale, e di quanto decresce o cresce in caso di diminuzione o accrescimento dell'errore

[net,tr]=train(net,x,t); % addestro la rete con l'algoritmo di L-M

I risultati dell'addestramento sono buoni dato che l'addestramento ha termine in 9 iterazioni e la funzione coseno è riprodotta fedelmente. Nei grafici sottostanti si può esaminare l'interpolazione effettuata dalla rete e l'andamento dell'errore, il quale diminuisce in maniera molto rapida (fig 15, fig 16).

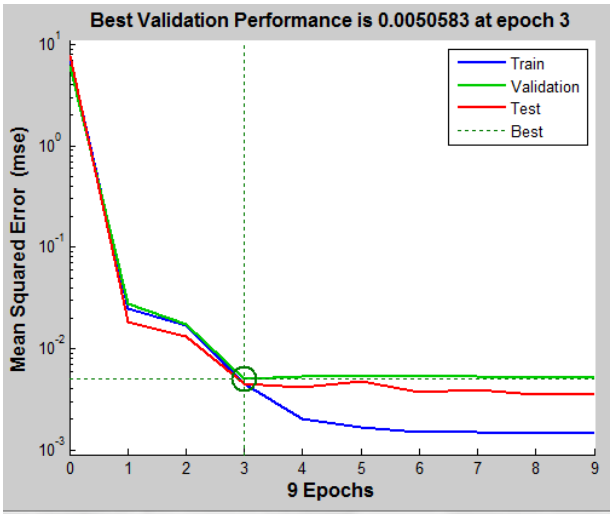

fig 15: Andamento dell'errore

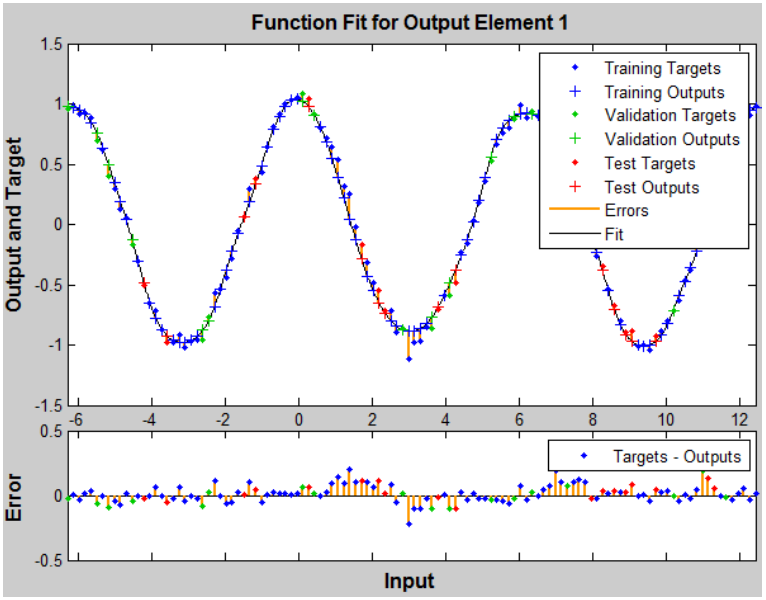

fig 16: Funzione rappresentata dalla rete

5 Conclusione

In conclusione le reti neurali sono un buono e veloce strumento per risolvere alcuni problemi per i quali è difficile trovare una soluzione tramite modelli. Come si è visto nelle simulazioni sopra effettuate le reti neurali possono essere utilizzate per l'interpolazione di dati sperimentali, per esempio con l'algoritmo di Levenberg-Marquardt. Oppure un efficace utilizzo di questo metodo riguarda appunto i problemi di classificazione (come la simulazione precedente riguardante le tipologie di vetro), i quali sono forse di ancora più difficile risoluzione rispetto ai precedenti con metodi di altro genere.
Le reti neurali sono presenti in molte applicazioni ingegneristiche, infatti avendo a disposizione più tempo, sarebbe stato interessante approfondire le applicazioni pratiche delle suddette. Per esempio implementare le reti neurali tramite dei linguaggi di programmazione, e quindi tentare di utilizzarle per applicazioni dal punto di vista software, come viene fatto per problemi di tipo finanziario o metereologico. Sarebbe stato di grande interesse anche approfondire le applicazioni hardware sia dal punto di vista dell'elettronica analogica come l'intel ETANN (electrically trainable analogue neural network), che consiste in un chip "addestrabile" il quale contiene 10000 pesi, che in quella digitale.
Un ultimo campo il cui approfondimento sarebbe stato di grande richiamo è quello delle applicazioni biomediche, sia dal punto di vista della ricerca riguardante il tentativo di comprensione del funzionamento dei neuroni del cervello tramite reti neurali, sia nelle applicazioni in bioinformatica.

Bibliografia

- Chris M. Bishop, *Neural networks and their applications*, Review of Scientific Instruments / Volume 65 / Issue 6, (1994).

- Chris M. Bishop, *Neural networks for pattern recognition*, Clarendon Press-Oxford, (1995).

- David J.C. MacKay, *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press 2003, pp.467-482, (2005).

- *http://archive.ics.uci.edu/ml/datasets.html*, Machine Learning Repository.