

UNIVERSITÀ DEGLI STUDI DI TORINO

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Magistrale in Informatica

Riassunto di

Reti Neurali

Enrico Mensa

Basato sulle lezioni di: *Rossella Cancelliere, Valentina Gliozzi*

Indice

| | | |
|----------|---|-----------|
| 1 | Introduzione a MATLAB | 1 |
| 1.1 | Dichiarazione di variabili e comandi preliminari | 1 |
| 1.1.1 | Dichiarare matrici | 2 |
| 1.1.2 | Salvare variabili su file | 2 |
| 1.1.3 | Riferirsi agli elementi di una matrice | 3 |
| 1.2 | Operazioni su matrici | 3 |
| 1.2.1 | Operazioni elemento per elemento | 4 |
| 1.3 | Grafici in MATLAB | 5 |
| 1.4 | Cicli e condizioni | 5 |
| 1.5 | Funzioni esterne | 6 |
| 1.6 | Utilità | 6 |
| 2 | Introduzione | 7 |
| 2.1 | Cos'è una rete neurale | 7 |
| 2.2 | Cosa studieremo | 8 |
| 2.2.1 | Reti neurali e psicologia | 8 |
| 2.3 | Il primo neurone (1943) | 9 |
| 2.3.1 | Critiche alle reti neurali | 10 |
| 3 | Reti a percettrone | 13 |
| 3.1 | Il neurone | 13 |
| 3.1.1 | La funzione di attivazione | 14 |
| 3.2 | Formulare il problema | 14 |
| 3.3 | L'algoritmo di apprendimento | 16 |
| 3.3.1 | Come modificare i pesi | 16 |
| 3.3.2 | Lo pseudocodice | 17 |
| 3.3.3 | Pesi e rette: cosa succede dal punto di vista geometrico? | 18 |
| 3.3.4 | Un esempio riassuntivo | 19 |
| 3.4 | Una garanzia importante: il teorema di convergenza | 22 |
| 3.5 | Le reti a percetroni: vantaggi e svantaggi | 24 |
| 3.5.1 | Usare la rete a percettrone | 24 |
| 3.5.2 | Un grande limite: problemi lineramente separabili | 25 |
| 3.6 | La delta rule | 26 |

| | | |
|----------|--|-----------|
| 3.6.1 | Differenze rispetto al percettrone | 27 |
| 3.6.2 | Cosa vogliamo fare | 27 |
| 3.6.3 | Come modificare un peso | 28 |
| 3.6.4 | Calcolare l'errore globale | 29 |
| 3.6.5 | Come modificare i pesi: una formula per tutti | 30 |
| 3.6.6 | Lo pseudocodice | 32 |
| 3.6.7 | Un breve esempio | 33 |
| 3.6.8 | Sulla convergenza | 34 |
| 4 | Reti multilivello | 35 |
| 4.1 | Implementare lo XOR: un esempio introduttivo | 35 |
| 4.2 | Reti back propagation: funzione di attivazione ed architettura | 37 |
| 4.2.1 | Definizioni preliminari | 37 |
| 4.3 | Addestrare le reti | 38 |
| 4.3.1 | Calcolare la variazione di peso per i neuroni output | 39 |
| 4.3.2 | Calcolare la variazione di peso per i neuroni hidden | 43 |
| 4.3.3 | Il punto della situazione: dove usare le formule | 46 |
| 4.3.4 | Un esempio riassuntivo | 47 |
| 4.3.5 | Algoritmo di apprendimento | 49 |
| 4.4 | Garanzie teoriche | 50 |
| 4.4.1 | Il learning rate | 50 |
| 4.4.2 | Criteri di stop | 51 |
| 4.4.3 | Riassumendo | 51 |
| 4.5 | I limiti delle reti back propagation | 52 |
| 4.6 | Applicazioni delle reti back propagation | 52 |
| 4.6.1 | Applicazioni ingegneristiche | 53 |
| 4.6.2 | Applicazioni in scienze cognitive | 55 |
| 5 | Self-Organizing Maps | 59 |
| 5.1 | Introduzione | 59 |
| 5.2 | Algoritmo di apprendimento e modifica dei pesi | 60 |
| 5.2.1 | Calcolare l'attivazione e trovare la BMU | 61 |
| 5.2.2 | Modificare i pesi | 61 |
| 5.2.3 | L'algoritmo | 63 |
| 5.3 | Un esempio cumulativo | 64 |
| 5.3.1 | Esecuzione dell'algoritmo | 66 |
| 5.3.2 | Risultati dell'algoritmo dopo quattro iterazioni | 72 |
| 5.4 | Applicazioni ed esempi conclusivi | 73 |
| 6 | Radial-Basis Function Networks | 79 |
| 6.1 | Introduzione | 79 |
| 6.2 | L'architettura di una rete RBF | 79 |
| 6.2.1 | La funzione di attivazione radiale | 81 |
| 6.3 | Funzionamento della rete: un problema di interpolazione | 81 |

| | | |
|----------|--|------------|
| 6.3.1 | Una rappresentazione matriciale | 82 |
| 6.3.2 | Il numero di unità hidden | 83 |
| 6.4 | Diversi algoritmi di apprendimento | 84 |
| 6.4.1 | Il bias | 84 |
| 6.5 | Algoritmo di apprendimento (1) | 84 |
| 6.5.1 | La pseudo-inversione | 85 |
| 6.5.2 | Ottimizzare il calcolo della pseudo-inversa | 86 |
| 6.5.3 | Riflessioni ed uso dell'algoritmo | 89 |
| 6.6 | Algoritmo di apprendimento (2) | 89 |
| 6.6.1 | I centri dei neuroni hidden | 89 |
| 6.6.2 | Definizione degli spreads e dei pesi per il livello output | 90 |
| 6.6.3 | Riflessioni ed uso dell'algoritmo | 90 |
| 6.7 | Algoritmo di apprendimento (3) | 91 |
| 6.8 | Combinare gli algoritmi | 91 |
| 6.9 | Comparazione RBF-FFNN | 91 |
| 7 | Extreme Learning Machine | 93 |
| 7.1 | Introduzione | 93 |
| 7.2 | L'architettura SLFN | 94 |
| 7.3 | I teoremi | 95 |
| 7.3.1 | Il teorema di interpolazione | 95 |
| 7.3.2 | Il teorema di approssimazione | 95 |
| 7.4 | Algoritmo di apprendimento | 96 |
| 7.4.1 | Quando la matrice è prossima alla singolarità | 96 |
| 7.4.2 | Calcolare H mediante Single Value Decomposition . . | 97 |
| 7.5 | Quando usare ELM e che vantaggi porta | 98 |
| 7.5.1 | Qualche risultato | 98 |
| 7.5.2 | Gli effetti della regolarizzazione | 100 |
| 8 | Hopfield Networks | 103 |
| 8.1 | Introduzione | 103 |
| 8.1.1 | Memorie associative | 103 |
| 8.2 | L'architettura | 104 |
| 8.2.1 | Da cosa è costituita la rete | 104 |
| 8.3 | Il funzionamento della rete | 105 |
| 8.3.1 | Fase di storage | 106 |
| 8.3.2 | Uso della rete | 106 |
| 8.3.3 | Un esempio | 108 |
| 8.4 | In merito alla convergenza | 108 |
| 8.4.1 | Il concetto di energia | 108 |
| 8.4.2 | Il teorema di convergenza | 110 |
| 8.4.3 | Risultato del teorema | 112 |
| 8.5 | Dati sperimentali | 112 |
| 8.5.1 | I problemi delle reti di Hopfield | 113 |

| | |
|---|------------|
| 8.6 La capacità delle reti di Hopfield | 115 |
| 9 Boltzmann machines | 117 |
| 9.1 Introduzione | 117 |
| 9.2 Studiare un sistema fisico | 118 |
| 9.2.1 Ad ogni stato una probabilità | 118 |
| 9.2.2 Lo spazio degli stati | 120 |
| 9.3 L'algoritmo di Simulated Annealing | 120 |
| 9.3.1 Algoritmo di Metropolis | 120 |
| 9.3.2 Diminuire la temperatura in ogni iterazione | 122 |
| 9.3.3 Implementazione e criteri di stop | 122 |
| 9.4 Reti neurali stocastiche | 123 |
| 9.5 Boltzmann machines | 124 |
| 9.5.1 Neuroni stocastici | 125 |
| 9.5.2 Il concetto di energia | 126 |
| 9.5.3 L'algoritmo di apprendimento | 126 |
| 9.5.4 Studio dell'algoritmo di apprendimento | 129 |
| 9.5.5 Algoritmo di Metropolis: come funziona la transizione | 132 |
| 9.6 Usare le macchine di Boltzmann | 133 |
| 9.7 Commenti sulle macchine di Boltzmann | 133 |
| 10 Deep Neural Networks | 135 |
| 10.1 Introduzione | 135 |
| 10.1.1 Perché DNN | 135 |
| 10.1.2 L'architettura | 136 |
| 10.1.3 L'apprendimento: idea introduttiva | 137 |
| 10.2 Restricted Boltzmann machines | 139 |
| 10.2.1 Pesi, energia e probabilità | 139 |
| 10.3 L'algoritmo di apprendimento | 141 |
| 10.3.1 Fase clamped su RBM | 141 |
| 10.3.2 Fase free su RBM | 142 |
| 10.3.3 L'update dei pesi | 143 |
| 10.4 Prestazioni delle DNN (MNIST dataset) | 143 |
| 10.4.1 Il MNIST dataset | 143 |
| 10.4.2 L'architettura della DNN | 144 |
| 10.4.3 Diversi modelli a confronto | 144 |
| 10.4.4 Risultati (hidden units arbitrarie) | 145 |
| 10.4.5 Risultati (hidden units limitate) | 146 |
| Bibliografia | 147 |

Capitolo 1

Introduzione a MATLAB

MATLAB è un linguaggio di alto livello e un ambiente interattivo per il calcolo numerico, l'analisi e la visualizzazione dei dati e la programmazione. MATLAB consente di analizzare dati, sviluppare algoritmi e creare modelli e applicazioni. Il linguaggio, gli strumenti e le funzioni matematiche incorporate consentono di esplorare diversi approcci e di raggiungere una soluzione più velocemente rispetto all'uso di fogli di calcolo o di linguaggi di programmazione tradizionali, quali C/C++ o Java.¹

Il nome MATLAB è la contrazione di matrix laboratory; è un sistema interattivo il cui elemento base è la matrice. MATLAB lavora essenzialmente con un unico tipo di oggetto: una matrice numerica rettangolare. Casi particolari sono le matrici 1×1 (ovvero i valori scalari) e le matrici con una sola riga o colonna, ovvero i vettori. Anche le operazioni ed i comandi di MATLAB sono pensati per l'ambito della matematica matriciale.

In questo capitolo andremo ad introdurre alcuni dei comandi più classici per utilizzare MATLAB. Ogni comando utilizzato in MATLAB dispone di una pagina di help esaustiva, che possiamo ottenere mediante il comando `help command`, dove `command` è il comando di cui vogliamo visualizzare l'help.

1.1 Dichiaraione di variabili e comandi preliminari

È possibile dichiarare variabili usando la scrittura:

```
» variabile=espressione
```

La dichiarazione termina con il tasto "Invio". Qualora si aggiunga il carattere `;`, il valore della variabile appena dichiarata non compare a video. Le variabili non hanno tipo. Se non si specifica un nome di variabile, MATLAB ne crea una (`ans`) in automatico e vi memorizza il risultato di `espressione`.

¹Tratto da www.mathworks.it.

In generale, tutte le variabili sono globali e vengono salvate nel workspace. Un'eccezione è rappresentata dalle variabili dichiarate nei files funzione, che vedremo più avanti.

1.1.1 Dichiarare matrici

Le matrici si dichiarano seguendo tre semplici regole:

- Ogni riga è separata da un punto e virgola.
- Ogni elemento di riga è separato da uno spazio o da una virgola.
- Tutti gli elementi devono essere chiusi fra quadre.

Ad esempio abbiamo:

```
» A=[1 2 3; 4 5 6; 7 8 9]
```

Che produrrà come risultato:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Ci sono modi alternativi per creare matrici. È possibile usare le funzioni:

- `eye(N)` che crea una matrice identità di dimensione $N \times N$.
- `zeros(N)` che crea una matrice di zeri di dimensione $N \times N$.
- `rand(N)` che crea una matrice con numeri random di dimensione $N \times N$

È possibile usare ulteriori parametri: per maggiori informazioni consultare l'`help`.

Volendo è anche possibile riempire vettori usando la notazione `x = i : k : n`. In questo modo si riempirà l'array `x` con elementi da `i` a `n` usando un passo `k`. Il passo può anche essere negativo, qualora i valori intorno siano coerenti.

1.1.2 Salvare variabili su file

Volendo è possibile salvare variabili su file `.mat`. Il file può essere creato tramite il comando:

```
save filename.mat
```

In questo modo si salveranno tutte le variabili attualmente nel workspace. Volendo salvare invece le variabili `a` e `b`, si userà il comando:

```
save filename.mat a b
```

Caricare il file è altrettanto intuitivo, basta utilizzare il comando `load`:

```
load filename.mat
```

Se le variabili contenute nel file esistono già nel workspace, verranno sovrascritte. Si rende quindi utile il comando di `clear`, che pulisce l'intero workspace. Per rimuovere invece una determinata variabile `var` è sufficiente usare il comando `clear var`.

1.1.3 Riferirsi agli elementi di una matrice

Ci si può riferire ai singoli elementi di una matrice utilizzando degli indici entro parentesi tonde. Ad esempio:

```
A(3, 3) = 100
```

modificherà il valore 9 di A in 100. Se si tratta con vettori e l'indice del nuovo valore non è ancora presente, vengono aggiunti tutti gli altri elementi (con valore 0) per poi popolare quello richiesto con il valore espresso.

Come elementi di una matrice possono essere usate anche delle matrici più piccole. Così, se vogliamo aggiungere ad una precedente matrice A una nuova riga, possiamo scrivere:

```
A = [A; nuova riga]
```

Se invece vogliamo aggiungere una nuova colonna, scriveremo

```
A = [A, nuova colonna]
```

Quindi possiamo scrivere `A = [A; 1, 2, 3]` e vedremo aggiunta la riga `[1, 2, 3]` alla matrice A. Non possiamo però scrivere scrivere `A = [A, 1;2;3]` perché MATLAB non riuscirebbe a distinguere la nuova fra i vari parametri. Si deve quindi scrivere: `A = [A, [1; 2; 3]]`.

Per indicare le righe (o le colonne) che vanno dalla *i*-esima alla *j*-esima di una matrice A, si può usare la notazione `i:j`. Inoltre, è possibile indicare tutte le righe o tutte le colonne mediante l'uso dei due punti. Supponendo che B sia una matrice 5×5 , per ottenere le prime due righe si deve usare il comando `B(1:2, :)`. Volendo invece ottenere una matrice 3×3 a partire dall'alto a sinistra si deve scrivere `B(1:3, 1:3)`.

1.2 Operazioni su matrici

È chiaramente possibile operare sulle matrici.

- `A'` fornisce la trasposta della matrice A.

- $A + B$ fornisce la somma fra una matrice A ed una matrice B (che devono avere stesse dimensioni).
- $A - B$ fornisce la sottrazione fra una matrice A ed una matrice B (che devono avere stesse dimensioni).
- $A + 2$ somma a tutti gli elementi della matrice A il valore 2.
- $A - 2$ sottrae a tutti gli elementi della matrice A il valore 2.
- $A * 2$ moltiplica a tutti gli elementi della matrice A il valore 2.
- $A / 2$ divide tutti gli elementi della matrice A per il valore 2.
- $A ^ 2$ è equivalente a $A * A$.

Il prodotto fra matrici è leggermente più sofisticato. Come sappiamo due matrici possono essere moltiplicate fra loro solo se il numero di colonne della prima è uguale al numero di righe della seconda. Inoltre, la moltiplicazione fra matrici non è commutativa. Il prodotto si indica come:

$$A * B$$

e verrà eseguito il calcolo:

$$\sum_{r=1}^n a_{ir} b_{rj}$$

per ogni valore di riga i e di colonna j .

Risoluzione di sistemi lineari ed inversa

Due operazioni molto importanti sono chiaramente la soluzione di sistemi lineari e il calcolo dell'inversa della matrice.

Scrivendo:

$$X = A \setminus B$$

risolveremo il sistema lineare $A * X = B$. Analogamente, potremmo scrivere:

$$X = \text{inv}(A) * B$$

Il primo operatore è però più performante.

1.2.1 Operazioni elemento per elemento

Abbiamo introdotto gli operatori $*$, $/$, \setminus , 2 , $.^*$. Di questi operatori esiste anche una versione che lavora "elemento per elemento". È sufficiente aggiungere un punto davanti all'operatore per sortire questo effetto.

Se ad esempio scriviamo $A.^2$ otterremo ogni elemento di A al quadrato anziché $A * A$. Analogamente, se scriviamo $G .* F$ otterremo una moltiplicazione elemento per elemento anziché la moltiplicazione matriciale classica.

1.3 Grafici in MATLAB

Mediante il comando `plot` è possibile disegnare su un asse cartesiano. La funzione richiede almeno un parametro (il valore delle ascisse) ma è anche possibile fornire il valore delle ordinate, che se mancante verrà riempito con un array da 0 ad n dove n è il numero di valori di ascissa che abbiamo.

I comandi `title`, `xlabel` e `ylabel` permettono di inserire label per il grafico e per gli assi. Volendo, è possibile fornire un terzo parametro alla funzione `plot` così da definire lo stile (colore, tratto, ecc) della linea. Si consulti l'`help` per maggiori dettagli.

È possibile aggiungere due linee su uno stesso grafico usando il comando `hold on` seguito da due `plot` (e poi `hold off` una volta terminato), oppure usando direttamente un comando `plot` del tipo:

```
plot(x1, y1, 'stile1', x2, y2, 'stile2')
```

Infine, i comandi `figure(i)` e `close(i)` permettono di mantenere più finestre aperte contemporaneamente (e agiscono sulla i -esima finestra).

1.4 Cicli e condizioni

Chiaramente è possibile esprimere cicli come negli altri linguaggi di programmazione. Vediamo alcuni esempi

```

1 % Ciclo for
2 for j=1:4, % j da 1 a 4
3     istr1;
4     istr2;
5     ...
6 end
7
8 % Ciclo while
9 while (x < 4)
10    istr1;
11    istr2;
12    ...
13 end
14
15 % If semplice
16 if (condition statement)
17     (matlab commands)
18 end
19
20 % If in cascata
21 if (condition statement)
```

```

22     (matlab commands)
23 elseif (condition statement)
24     (matlab commands)
25 elseif (condition statement)
26     (matlab commands)
27     ...
28 else
29     (matlab commands)
30 end

```

Listing 1.1: Sintassi per cicli e condizioni.

1.5 Funzioni esterne

In MATLAB è possibile definire funzioni e salvarle in file .m. Vediamo ad esempio una funzione che calcola la media:

```

1 function [y]=mean(x)
2 % mean calcola la media o il valor medio
3 % nel caso di vettori restituisce il valor medio
4 % nel caso di matrici restituisce un vettore riga
5 % contenente il valor medio calcolato per ogni colonna
6
7 [m,n]=size(x)
8 if m==1
9     m=n;
10 end
11 y=sum(x)/m;

```

Listing 1.2: File mean.m.

Come vediamo, in alto vengono specificati i parametri (x) e i valori di output (y). A riga 11 vediamo che il valore di output y viene popolato. Al termine dell'esecuzione di una funzione così dichiarata, le variabili locali scompaiono.

1.6 Utilità

Modificare la cartella di inizio. Per modificare la cartella di inizio è sufficiente aggiungere un file `startup.m` nel path di MATLAB (ad esempio in `Documents/MATLAB`) e scrivere il comando `cd('nuovo_path')`.

Capitolo 2

Introduzione

Nel corso degli anni i computer hanno sempre più sostituito molte delle operazioni compiute dagli uomini, riuscendo a risolvere in maniera performante molte classi di problemi.

Alcune classi di problemi, però, risultano difficilmente risolvibili mediante l'uso dei calcolatori, pensiamo ad esempio a problemi di pattern recognition o di scienze cognitive. È per risolvere questo tipologie di problemi che nascono le *reti neurali*, uno strumento che ha l'obiettivo di simulare il pensiero umano ottenendo risultati competitivi rispetto al cervello vero e proprio.

2.1 Cos'è una rete neurale

Il nostro cervello contiene miliardi di neuroni i quali producono milioni di milioni di sinapsi. Ogni neurone riceve una serie di segnali elettrici in input, e, possibilmente, ne trasmetterà ai neuroni che lo seguono mediante l'uso di dendriti.

Le nostre reti neurali provano a simulare questo comportamento, sfruttando una sorta di neuroni simulati. Questi neuroni, infatti, riceveranno e peseranno i segnali dai propri input, li sommeranno e li faranno passare all'interno di una funzione di attivazione così da, eventualmente, trasmettere un output.

Diverse configurazioni di neuroni e diverse funzioni di attivazione definiscono diverse architetture, ovvero diverse reti neurali.

Una rete neurale contiene quindi *sia la conoscenza e sia le regole* per fare inferenza: entrambe le parti sono espresse tramite i pesi dei vari input interpretati dai neuroni.

I pesi non vengono definiti a monte ma vengono appresi dalla rete stessa, mediante una fase di *learning*. Tale fase può essere *supervisionata* o meno. Se lo è, sarà l'utente umano a dover dare alla rete una vasta serie di esempi così da educare la rete, altrimenti sarà la rete stessa a fare learning a partire da zero.

2.2 Cosa studieremo

Come appena detto, esistono diverse *architetture* di reti neurali. Porremo la nostra attenzione sulle reti più usate o più significative dal punto di vista didattico:

- Reti a *percetroni*: è l'architettura più semplice, si ha un solo livello di neuroni.
- Reti *delta rule*: una variante delle reti a percetroni che cerca di valutare l'errore globale della rete.
- Reti *back-propagation*: le semplici reti a percetroni non sono in grado di rappresentare problemi relativamente semplici; ad esempio non sono in grado di rappresentare lo XOR. Nascono così le reti dette *back propagation*, che introducono più livelli di neuroni e che aggiungono una fase di "propagazione all'indietro" dell'errore stimato, così da aggiustare i pesi. Si noti che questo fenomeno è assolutamente inesistente nel cervello umano. È di gran lunga la tipologia di rete neurale più diffusa.
- *Self Organizing Maps* (SOM): si tratta di reti che sono alternative psicologicamente valide alle reti back propagation, ma hanno il vantaggio di non necessitare di learning supervisionato.
- *Reti radiali*: reti a due livelli, ognuno dei quali ha un significato diverso.
- *Extreme Learning Machines*
- *Hopfield Networks*
- *Deep Neural Networks*

Tutte quelle reti i cui nodi di output *non* fungono da input per la rete sono dette FFNN (*feedforward neural network*). Le reti non FFNN presentano dei loop.

2.2.1 Reti neurali e psicologia

Come già accennato le reti neurali sono in grado di risolvere problemi interessanti, fra cui:

- Categorizzazione di oggetti.
- Pattern recognition.
- Imparare una lingua.

In molti casi, gli studi dimostrano che l'andamento delle reti neurali è paragonabile allo sviluppo di un bambino. Si riscontrano infatti gli stessi errori e le stesse fasi di progresso fra la rete e l'infante stesso.

Per poter effettuare "test" sui bambini, si frutta la cosiddetta *novelty preference procedure*, si misura cioè lo sguardo del bambino e maggiore è la quantità di secondi per cui il bambino fissa un'immagine, maggiore sarà il suo stupore verso quell'immagine. Capiremo in seguito come interpretare questo stupore.

Esempio sulla categorizzazione. Uno degli esempi che possiamo portare è quello della categorizzazione di immagini. Fornendo infatti i due insiemi indicati in Figura 2.1 sia le reti neurali che i bambini sono stati in grado di delinare nel primo caso un solo insieme di elementi e nel secondo due insiemi diversi (collo lungo o gambe lunghe). Parleremo approfonditamente di questo esempio nel

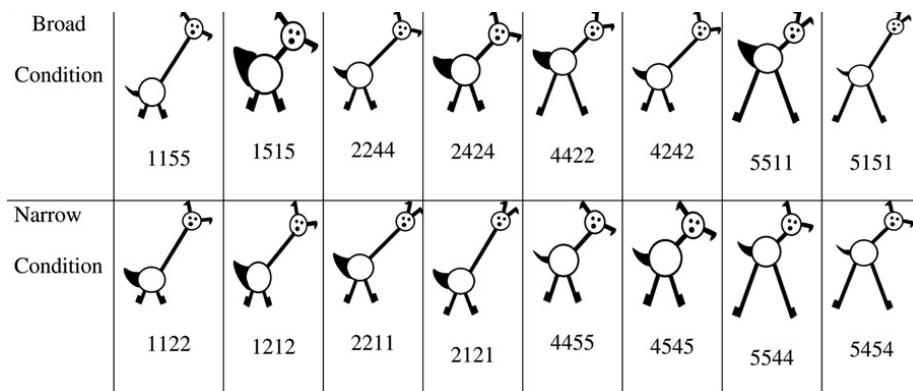


Figura 2.1: Due diversi insiemi tramite i quali allenare un bambino o una rete neurale.

capitolo relativo alle SOM.

2.3 Il primo neurone (1943)

Il primo neurone sviluppato nel 1943 è molto semplice ed è rappresentato in Figura 2.2. Dato in neurone j , abbiamo una serie di input x_1, \dots, x_p ad ognuno dei quali è assegnato un peso w_{j1}, \dots, w_{jp} proprio del neurone. I pesi possono essere considerati come eccitatori (se maggiori di zero) o inibitori (se minori di zero) ma possono anche essere zero.

Per ogni input i si ha quindi che il segnale che ricevuto dal neurone è del tipo:

$$x_i \cdot w_i$$

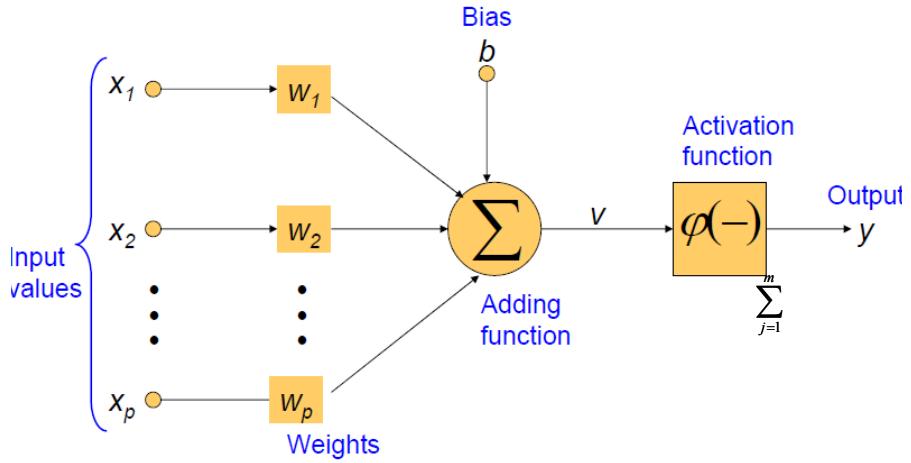


Figura 2.2: Il primo neurone.

Tutti questi valori vengono sommati fra loro tramite la *adding function* che calcola quindi il *netinput*:

$$j = \sum_{i=1}^p w_{ji}x_i$$

Se il netinput è sufficiente, allora l'activation function propagherà il valore di output.

Ogni neurone dispone inoltre di un *bias*, un valore che rappresenta l'attivazione spontanea della rete. Se il bias è maggiore di zero il neurone è sovente attivo, anche se il netinput è pari a 0. Con bias negativo invece dovremo avere un neurone attivo (e il bias rappresenta la soglia) affinché propaghi il suo output. In definitiva, quindi, l'output del neurone dipenderà dal netinput e dal bias secondo la formula:

$$y_j = \varphi(u_j + b)$$

dove $(u_j + b)$ è detto *potenziale di attivazione* o *campo locale* del neurone j .

Il bias come un output

Al fine di semplificare i calcoli è possibile pensare al bias come un input che vale sempre 1 ed ha peso pari a b . Modificheremo quindi la formula del netinput:

$$j = \sum_{i=0}^p w_{ji}x_i$$

2.3.1 Critiche alle reti neurali

Durante gli anni sono state mosse tre grandi critiche alle reti neurali:

- La semantica non è chiara (regole e fatti sono combinati insieme).
- Non si capisce bene come venga computato il risultato, ma funzionano.
- Alcune reti imparano molto lentamente.

Capitolo 3

Reti a percettrone

Le reti a percettrone sono state introdotte da Frank Rosenblatt nel 1958. Sono una forma molto semplice di reti neurali ad un livello, in grado di risolvere problemi lineramente separabili. L'ambizione di Rosenblatt era quella di simulare il funzionamento del cervello: l'idea piacque molto, soprattutto perché le reti a percettrone forniscono interessanti garanzie di convergenza. L'inabilità di queste reti nel risolvere determinate classi di problemi, però, ne ha segnato il declino.

3.1 Il neurone

Abbiamo già studiato la forma di un neurone nella Sezione 2.3. Ricordiamo che assimiliamo il bias come un input sempre ad 1, con peso pari al valore del bias. In Figura 3.1 vediamo la sua rappresentazione.

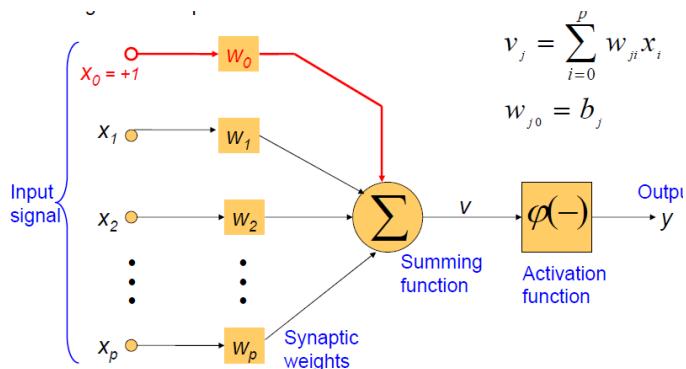


Figura 3.1: Il neurone.

Le reti a percettrone sono molto semplici e coinvolgono un solo livello di neuroni. Si ha dunque un insieme di input (possiamo immaginarlo come segnali derivanti dalla codifica di un'immagine, oppure valori di bit se la nostra rete

implementa una funzione logica) che sono collegati ad ogni percettrone. In Figura 3.2 vediamo un esempio di rete a percettrone.

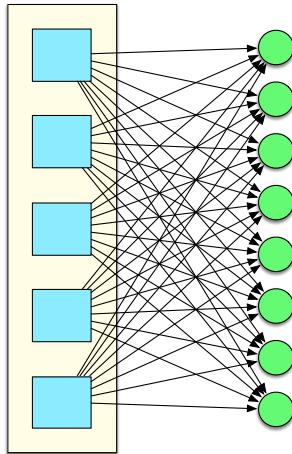


Figura 3.2: Una rete a percettrone: con quadrati rappresentiamo l'input mentre con i tondi rappresentiamo i neuroni.

Noi ci limiteremo a studiare un caso semplice: una rete costituita da un solo neurone. Con una rete di questo tipo, si avrà un valore di output binario (1 oppure -1 , come stiamo per vedere). Saremo quindi in grado di categorizzare l'input in due insiemi distinti. Nel caso in cui volessimo definire più insiemi, è sufficiente aggiungere neuroni e combinare i valori di output di ogni neurone. È quindi fondamentale non confondere il concetto di livello con il concetto di parallelismo fra neuroni.

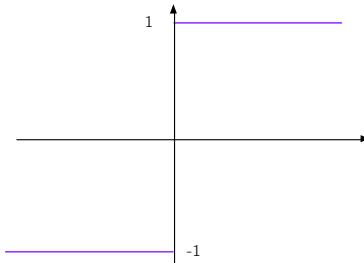
3.1.1 La funzione di attivazione

Abbiamo già accennato che la funzione di attivazione funge da soglia, in grado di stabilire se il neurone debba propagare o no il suo segnale (definisce quindi l'output del neurone stesso). Nel caso delle reti a percettrone, la funzione di attivazione è discreta ed estremamente semplice: se il valore della sommatoria è *strettamente maggiore di 0*, la funzione di attivazione restituisce 1. Se invece il valore della sommatoria è *minore o uguale a zero*, allora la funzione ritornerà -1 . In Figura 3.3 vediamo la rappresentazione sul piano cartesiano della funzione.

3.2 Formulare il problema

Prima di capire come la rete sia in grado di apprendere i pesi da sola, dobbiamo capire come formalizzare un problema per una rete.

Il problema viene definito come una serie di coppie del tipo:

Figura 3.3: Funzione di attivazione φ per la rete a percettrone.

```
([input], output desiderato)
```

che vogliamo la rete apprenda. Al termine della fase di apprendimento, ci aspettiamo che la rete abbia trovato un insieme di pesi (uno per ogni input) tale per cui *per ogni input* facente parte di quelli forniti alla rete venga restituito l'output desiderato.

Chiaramente saremo interessati anche a calcolare quanto la rete sia *generale*, ovvero in grado di dare un output corretto anche per un input che non è stato utilizzato in fase di apprendimento.

Un esempio: una rete che simuli l'OR

Per chiarificare quanto appena detto, prendiamo il problema dell'OR. Ci servirà una rete con un solo neurone e con due valori di input. Il problema è così formalizzato:

```
([1, 1], 1)
([1, 0], 1)
([0, 1], 1)
([0, 0], -1)
```

Listing 3.1: Problema dell'OR per una rete a percettrone.

Ricordiamo che però, in realtà, gli input sono tre a causa del bias (che viene sempre aggiunto con valore 1). La rete avrà quindi tre input ed il problema risulta essere:

```
([1, 1, 1], 1)
([1, 1, 0], 1)
([1, 0, 1], 1)
([1, 0, 0], -1)
```

Listing 3.2: Problema dell'OR per una rete a percettrone (con bias come input).

Esistono infiniti valori assegnabili ai pesi. Una possibile configurazione risulta essere:

$$w_0 = -0.1$$

$$w_1 = 0.5$$

$$w_2 = 0.5$$

Infatti, per tutti i possibili valori di input x_1, x_2 abbiamo che:

$$\begin{cases} \text{output} = 1, & \text{se } w_1x_1 + w_2x_2 + w_0 > 0 \\ \text{output} = -1, & \text{se } w_1x_1 + w_2x_2 + w_0 \leq 0 \end{cases}$$

e dunque vengono rispettati gli output richiesti dal problema.

In questo esempio siamo stati noi a dare i pesi alla rete: scopo dell'algoritmo di apprendimento è quello di riuscire a fornire automaticamente i pesi tali per cui ogni pattern fornito dal problema abbia un output corretto nella rete.

3.3 L'algoritmo di apprendimento

L'algoritmo di apprendimento (come del resto la rete) è estremamente semplice. L'idea di base è quella di prendere ogni pattern costituente il problema e testarlo sulla rete: se l'output fornito dalla rete corrisponde all'output desiderato, non si fa nulla e si passa al pattern successivo. Qualora però il risultato sia diverso da quello atteso, si modificano (in maniera abbastanza grossolana) i pesi affinché l'input venga classificato correttamente.

È evidente che questo approccio possa portare ad una modifica dei pesi troppo pesante, rendendo pattern precedentemente classificati in maniera corretta nuovamente mal classificati. L'algoritmo procede, infatti, finché tutti i pattern non sono correttamente classificati.

3.3.1 Come modificare i pesi

Introduciamo un po' di notazione.

Supponendo di avere due categorie C_1 e C_2 secondo le quali vogliamo suddividere l'insieme dei pattern descritti dal problema, vogliamo che l'output della rete y sia 1 se $\vec{x} \in C_1$ e vogliamo che y sia invece -1 se $\vec{x} \in C_2$. Con \vec{x} indichiamo quindi uno dei pattern specificati dal problema, mentre la sua appartenenza a una delle due categorie indica l'output atteso dalla rete. Definendo \vec{w} come il vettore dei pesi mantenuti nella rete, vogliamo sostanzialmente che:

$$\begin{cases} \vec{w}^T \vec{x} > 0, & \text{se } \vec{x} \in C_1 \\ \vec{w}^T \vec{x} \leq 0, & \text{se } \vec{x} \in C_2 \end{cases}$$

Abbiamo usato una scrittura vettoriale per semplificare il calcolo della somma.

Ad esempio, con il neurone che rappresentava l'OR di poco fa, avremmo per il primo pattern ($[1, 1, 1]$, 1):

$$\begin{aligned} \vec{w}^T \vec{x} = \\ [-0.1 \ 0.5 \ 0.5] \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \\ -0.1 \cdot 1 + 0.5 \cdot 1 + 0.5 \cdot 1 = 0.9 \end{aligned}$$

Essendo 0.9 maggiore di zero, abbiamo come output 1: proprio come desiderato.

Andiamo ora ad esplicitare come modificare i pesi se ci troviamo all'iterazione n e stiamo passando alla $n+1$ -esima. Indichiamo con $w(n)$ il vettore dei pesi all'iterazione n e con $x(n)$ gli input al passo n -esimo (ricordiamo che ad ogni iterazione l'input può cambiare, dovendo coinvolgere tutti i pattern forniti dal problema).

Qualora l'output della rete sia quello atteso (ovvero la classificazione è corretta), ci troviamo in uno dei seguenti casi:

$$\begin{cases} w(n+1) = w(n), & \text{se } w(n)^T x(n) > 0 \text{ e } x \in C_1 \\ w(n+1) = w(n), & \text{se } w(n)^T x(n) \leq 0 \text{ e } x \in C_2 \end{cases}$$

Ovvero i pesi rimangono inalterati.

Se invece l'output non è quello atteso (la classificazione *non* è corretta), ci troviamo in uno dei seguenti casi:

$$\begin{cases} w(n+1) = w(n) - \eta(n)x(n), & \text{se } w(n)^T x(n) > 0 \text{ e } x \in C_2 \\ w(n+1) = w(n) + \eta(n)x(n), & \text{se } w(n)^T x(n) \leq 0 \text{ e } x \in C_1 \end{cases}$$

I pesi devono essere modificati poiché non corretti: nel primo caso vengono ridotti perché l'output è risultato troppo grande, mentre nel secondo vengono aumentati poiché l'output è risultato troppo piccolo. In entrambi i casi il peso viene aggiustato usando l'input dell'iterazione corrente moltiplicato per un fattore $\eta(n)$ detto *learning rate*. Il learning rate può essere fisso o variabile ad ogni iterazione: la sua stima è tutt'altro che semplice, ed è spesso derivante dall'esperienza più che da nozioni teoriche.

3.3.2 Lo pseudocodice

Riportiamo ora lo pseudocodice dell'algoritmo di apprendimento che sfrutta le funzioni di aggiustamento dei pesi di cui abbiamo appena parlato.

```

n = 1;
inizializza w(n) casualmente;

while(ci sono ancora esempi mal classificati) {
    Seleziona un esempio mal classificato a caso (x(n), d(n));
    w(n + 1) = w(n) + η(n)d(n)x(n);
    n = n + 1;
}

```

Listing 3.3: Pseudocodice per l'algoritmo di apprendimento delle reti a percettrone.

Abbiamo indicato con $d(n)$ il segno dell'output fornito dalla rete, così da scrivere più comodamente la riga di aggiornamento dei pesi.

Si noti che l'algoritmo ha quindi una visione locale del problema (pattern by pattern): abbiamo garanzia di terminazione? La risposta è sorprendentemente sì (se il problema è lineramente separabile); ne parleremo profusamente nella sezione dedicata alla garanzia di convergenza.

3.3.3 Pesi e rette: cosa succede dal punto di vista geometrico?

Come sappiamo le reti a percettrone sono in grado di risolvere problemi linearmente separabili. Questo si traduce geometricamente nel tracciare un piano che suddivide lo spazio degli input in due sezioni fra loro totalmente separate. In Figura 3.4 abbiamo una rappresentazione geometrica di questo fatto (in un caso bidimensionale). Più precisamente, ciò che accade è che il vettore \vec{w}

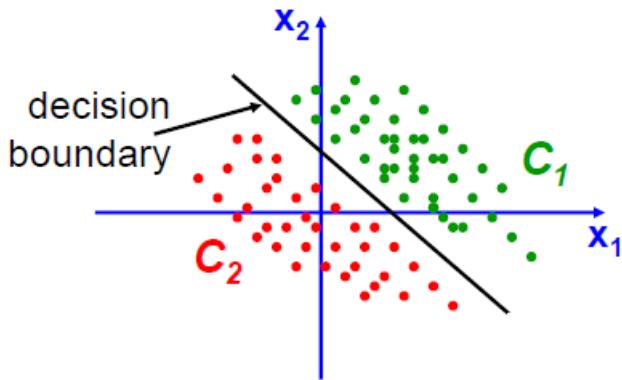


Figura 3.4: Una retta suddivide le due categorie, la cui intersezione degli elementi è vuota.

(d'ora in poi rappresentato senza freccia per comodità) definisce una retta, la cui perpendicolare funge da *decision line*. Infatti, per tutti gli input con vettore x che si trovano a destra della decision line si ha che $w \cdot x > 0$. Questo è

facilmente verificabile. Come sappiamo, il prodotto scalare è così definito:

$$w \cdot x = \|w\| \|x\| \cos \theta$$

dove θ è l'angolo fra w e il vettore relativo all'input considerato x . Per avere tale prodotto maggiore di zero, θ deve essere incluso fra -90 e +90 gradi (e quindi x deve trovarsi a destra della decision line).

Chiarifichiamo quanto appena detto con l'ausilio della Figura 3.5.

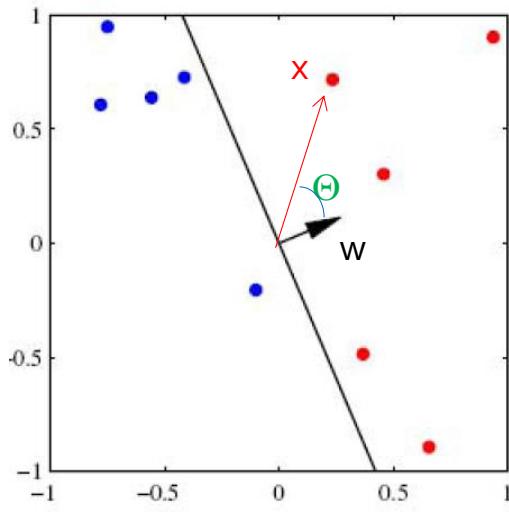


Figura 3.5: La decision line è perpendicolare a w : tutto ciò che si trova alla sua destra appartiene ad una categoria, mentre il resto appartiene ad un'altra.

3.3.4 Un esempio riassuntivo

Facciamo ora un esempio di esecuzione dell'algoritmo, verificando anche la posizione della decision line. Il problema è monodimensionale (il primo input è il bias, quindi di fatto è bidimensionale) e corrisponde a questa descrizione:

- A: ([1, 0.2], 1)
- B: ([1, -0.7], -1)
- C: ([1, 0.4], 1)
- D: ([1, -0.5], -1)

Listing 3.4: Problema per una rete a percettrone.

Abbiamo assegnato dei nomi ai vari pattern così da poterli identificare più comodamente. Useremo un $\eta = 0.2$.

Iniziamo l'esecuzione dell'algoritmo assegnando i pesi a caso (diciamo a 0) ed inizializzando il numero di iterazioni n ad 1. In Figura 3.6 vediamo la situazione iniziale dell'algoritmo dal punto di vista grafico.

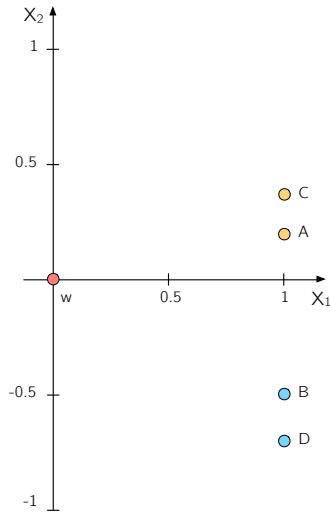


Figura 3.6: Situazione ad inizio algoritmo.

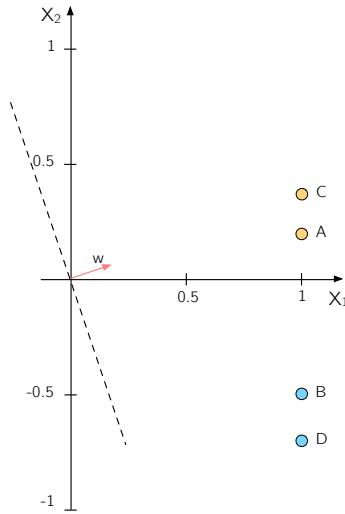
Iterazione 1. Testiamo gli attuali pesi con il pattern *A*:

$$w^T \cdot x = 0 \rightarrow -1$$

La funzione di attivazione restituisce -1 mentre noi volevamo 1 . Modifichiamo quindi i pesi:

$$w(2) = w(1) + \eta x(1) = [0 \ 0] + 0.2 [1 \ 0.2] = [0.2 \ 0.04]$$

In Figura 3.7 vediamo come si è spostata la retta dei pesi: È evidente che la

Figura 3.7: La retta dei pesi $w(2)$ e decision line (tratteggiata).

decision line non separa ancora i nostri input in maniera corretta.

Iterazione 2. Testiamo gli attuali pesi con il pattern B :

$$w^T \cdot x = 0.172 \rightarrow 1$$

La funzione di attivazione restituisce 1 mentre noi volevamo -1 . Modifichiamo quindi i pesi:

$$w(3) = w(2) - \eta x(2) = [0.2 \ 0.04] + 0.2 [1 \ -0.7] = [0 \ 0.18]$$

In Figura 3.8 vediamo come si è spostata la retta dei pesi. A quanto pare la

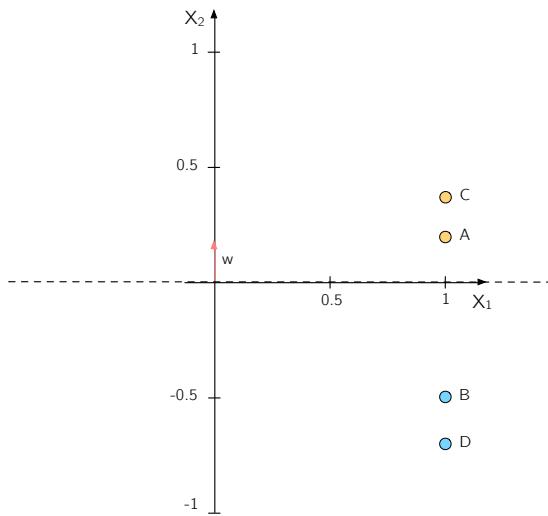


Figura 3.8: La retta dei pesi $w(3)$ e decision line (tratteggiata).

decision line sembra essere in grado di dividere tutti i pattern; lo verifichiamo numericamente:

- Per il pattern A : $w^T \cdot x = 0.036 \rightarrow 1$. Corrisponde.
- Per il pattern B : $w^T \cdot x = -0.126 \rightarrow -1$. Corrisponde.
- Per il pattern C : $w^T \cdot x = 0.072 \rightarrow 1$. Corrisponde.
- Per il pattern D : $w^T \cdot x = -0.09 \rightarrow -1$. Corrisponde.

L'algoritmo può terminare.

Il nostro esempio è stato parecchio fortunato... non sempre si termina in così pochi passi. Il vettore dei pesi, infatti, può essere “sballottato” e modificato molte volte. La garanzia di terminazione ci è data però dal teorema di convergenza che stiamo per studiare.

3.4 Una garanzia importante: il teorema di convergenza

Il teorema di convergenza che stiamo per studiare è stato il fiore all'occhiello delle reti a percettrone. È infatti un risultato del tutto inaspettato, giacché l'algoritmo di apprendimento ragiona con un principio di località (esamina infatti il problema fornитогli pattern by pattern senza considerarli tutti insieme).

Il teorema pone due limiti (uno superiore ed uno inferiore) al valore della norma dei pesi: combinando i due limiti otterremo che la norma deve dunque essere finita, e così le iterazioni che ne richiedono il calcolo. Otteniamo così garanzia di terminazione. Facciamo due assunzioni iniziali:

- I pesi sono inizializzati tutti a 0.
- Il learning rate è a 1 (per semplificare i calcoli).

Inoltre, come passo preliminare, viene trasformato il problema negando tutti i valori di input presenti in pattern che hanno output pari a -1 .

Più formalmente, diciamo che C è il set ottenuto da $C_1 \cup C_2$ dove x viene trasformato in $-x$ per tutte le $x \in C_2$ (cioè con output desiderato pari a -1). Il problema è così diventato la ricerca di un determinato w_* tale per cui $w_*^T \cdot x > 0$ per ogni x , ovvero una distribuzione di pesi che classifichi ogni input fornendo output 1.

Definita $x(1), \dots, x(k) \in C$ la serie di input che sono state considerate per correggere il vettore dei pesi w durante le prime k iterazioni, allora avremo che:

$$w(k+1) = x(1) + \dots + x(k)$$

Questo poiché se abbiamo dovuto correggere l'output, questo doveva necessariamente essere pari a -1 (per come abbiamo trasformato il problema), dunque in ogni correzione abbiamo dovuto aggiungere l'input del passo precedente.

Determinare il lower bound. Supponiamo per ipotesi che esista il w_* che cerchiamo, ovvero tale per cui $w_*^T \cdot x > 0$ per ogni $x \in C$. Avremo allora che:

$$w_*^T \cdot w(k+1) = w_*^T \cdot (x(1) + \dots + x(k))$$

Abbiamo semplicemente moltiplicato il vettore dei pesi all'iterazione $k+1$ per il nostro vettore dei pesi soluzione. Svolgendo la moltiplicazione otteniamo:

$$w_*^T \cdot w(k+1) = w_*^T \cdot x(1) + \dots + w_*^T \cdot x(k)$$

Tale valore deve essere necessariamente maggiore o uguale di $k\alpha$, dove α è il più basso valore di input moltiplicato per il vettore w_* . Questo è banalmente vero poiché abbiamo una somma di queste moltiplicazioni e abbiamo

considerato l'addendo più piccolo:

$$w_\star^T \cdot x(1) + \dots + w_\star^T \cdot x(k) \geq k\alpha$$

Sfruttando ora la disegualanza di Cauchy-Schwarz possiamo dire che:

$$\|w_\star\|^2 \|w(k+1)\|^2 \geq [w_\star^T w(k+1)]^2$$

Ciò che si trova a sinistra della disegualanza (elevato al quadrato) sappiamo anche essere maggiore di $k\alpha$ per quanto detto sopra, e dunque possiamo effettuare una maggiorazione e dire:

$$\|w_\star\|^2 \|w(k+1)\|^2 \geq (k\alpha)^2$$

ovvero:

$$\|w(k+1)\|^2 \geq \frac{(k\alpha)^2}{\|w_\star\|^2}$$

Ecco quindi che abbiamo trovato un lower bound per la norma del nostro vettore dei pesi all'iterazione $k+1$.

Determinare l'upper bound. Andiamo ora a trovare un upper bound per la norma del vettore dei pesi ($\|w(k+1)\|^2$). Sappiamo che:

$$w(k+1) = w(k) + x(k)$$

e dunque è anche vero:

$$\|w(k+1)\|^2 = \|w(k)\|^2 + \|x(k)\|^2 + 2w^T(k)x(k)$$

L'ultimo termine, però, sappiamo essere minore o uguale a zero, poiché l'input x al passo k è mal classificato (cioè risulta essere minore di zero), altrimenti non avremmo fatto l'aggiustamento. Possiamo perciò sostenere che:

$$\|w(k+1)\|^2 \leq \|w(k)\|^2 + \|x(k)\|^2$$

Abbiamo semplicemente rimosso l'ultimo termine e, sapendo che questo è minore o uguale a zero, abbiamo corretto il simbolo di uguaglianza ottenendo così il simbolo di disegualanza.

Questa disegualanza è vera per ogni iterazione, dunque, possiamo scrivere:

$$\|w(k+1)\|^2 \leq \sum_{i=0}^k \|x(i)\|^2$$

Abbiamo semplicemente esploso ogni $w(i)$ nella somma dei suoi termini (ricordiamo che il valore dei pesi al passo i è pari alla somma di tutti gli input fino al passo $i-1$ esimo).

Se definiamo ora β come la massima norma dei vettori input possibili:

$$\beta = \max \|x(n)\|^2 \forall x(n) \in C$$

possiamo facilmente scrivere:

$$\|w(k+1)\|^2 \leq k\beta$$

poiché, se ad ogni iterazione (e ne abbiamo k) abbiamo aggiunto il valore di un determinato input, prendendo il massimo input e moltiplicandolo per il numero di iterazioni non possiamo che ottenere un valore più grande dell'effettiva norma dei pesi al passo $k+1$.

Unendo i limiti. Abbiamo così determinato che valgono le due condizioni:

- $\|w(k+1)\|^2 \geq \frac{(k\alpha)^2}{\|w_*\|^2}$
- $\|w(k+1)\|^2 \leq k\beta$

e quindi possiamo dire:

$$\frac{(k\alpha)^2}{\|w_*\|^2} \leq \|w(k+1)\|^2 \leq k\beta$$

Essendo quindi la norma del vettore dei pesi inclusa fra due valori, non può essere infinita: *l'algoritmo termina.* \square

3.5 Le reti a percetroni: vantaggi e svantaggi

3.5.1 Usare la rete a percettrone

Le reti a percettrone nascono come un device ottimo per il pattern recognition. La percezione mima una parte di ciò che sappiamo in merito al funzionamento della visione nei mammiferi: ad esempio, gruppi di celle rispondono all'esistenza di una linea in una specifica direzione. Consideriamo ad esempio il problema in Figura 3.9: ogni input fornisce delle informazioni in merito alle 9 celle prese in considerazione.

Più precisamente abbiamo che l'input $Ah1$ varrà 1 se almeno due delle tre celle sulla prima linea orizzontale sono nere. L'input $Ah2$ seguirà lo stesso schema ma per la seconda linea orizzontale, mentre $Ah3$ lo farà per la terza. Abbiamo anche gli input $Av1$, $Av2$ ed $Av3$ che si comportano allo stesso modo ma relativamente alle colonne.

Dando la T (a cui vogliamo associato il valore 1) e la H (a cui vogliamo associato il valore -1) maiuscole come training set, la rete calcolerà i seguenti

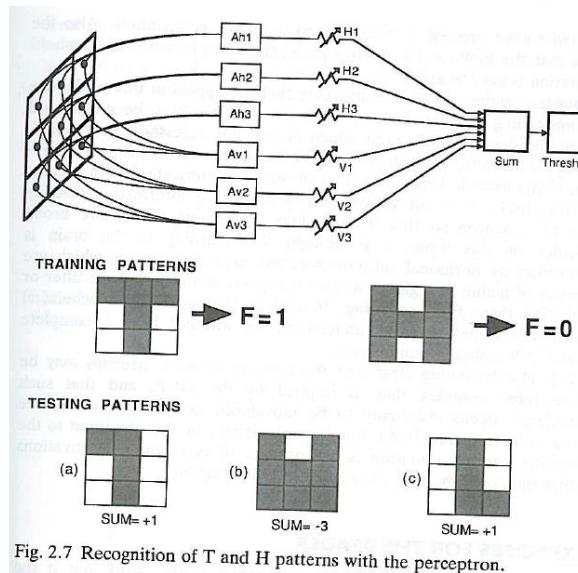


Fig. 2.7 Recognition of T and H patterns with the perceptron.

Figura 3.9: Riconoscimento della T e della H usando una rete a percettrone.

pesi: ad Av_2 verrà fornito un grande peso positivo poiché se quella colonna è "attiva" allora certamente avremo una T. Il peso associato a Ah_1 sarà invece prossimo allo zero, poiché è un input positivo per entrambe le lettere. Gli altri input avranno invece piccoli pesi negativi.

3.5.2 Un grande limite: problemi lineramente separabili

Lo abbiamo detto più volte: le reti a percettrone sono in grado di risolvere solamente problemi linearmente separabili. Dovrebbe ormai essere evidente la ragione: non è possibile trovare un iperpiano che separi in due insiemi non distinti i vari input. In Figura 3.10 vediamo il problema dello XOR.

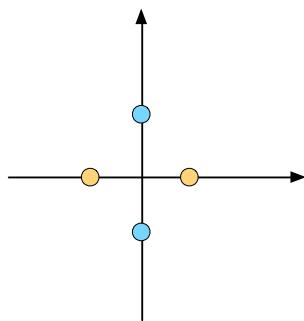


Figura 3.10: Rappresentazione grafica del problema dello XOR.

Nel 1969 Minksy e Papert pubblicano il paper *Perceptrons*, aprendo una grande

crisi del percettrone e in generale della ricerca nelle reti neurali. In particolare, a causa della loro località, le reti a percettrone non sono in grado di distinguere proprietà fondamentali fra cui connettività e parità (lo vediamo in Figura 3.11).

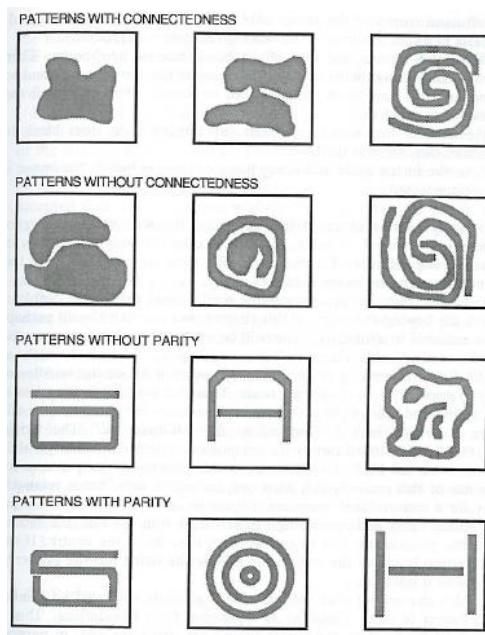


Figura 3.11: Pattern problematici per le reti a percettrone.

3.6 La delta rule

La Delta rule è una regola di discesa del gradiente per aggiornare i pesi dei segnali di input che giungono ad un neurone. È una tecnica alternativa all'algoritmo di apprendimento che abbiamo studiato poco fa, ed è stato proposto da Widrow ed Hoff nel 1960. L'idea di fondo è quella di calcolare l'errore di categorizzazione compiuto dal network e di aggiustare i pesi in modo da minimizzarlo.

Fra le reti a percettrone e la delta rule c'è quindi una grande differenza di filosofie: nella prima cerchiamo di minimizzare (fino a 0) il numero di pattern mal classificati, mentre nella seconda cerchiamo di minimizzare l'errore globale portandolo sotto una determinata soglia. Questo significa che una rete neurale che fa uso della delta rule potrebbe anche *non classificare correttamente* tutti i pattern che gli sono forniti in input nella fase di learning.

3.6.1 Differenze rispetto al percettrone

Dal punto di vista tecnico, le differenze rispetto al percettrone sono due: la funzione di attivazione e l'algoritmo di apprendimento, mentre la struttura della rete rimane immutata.

L'algoritmo di apprendimento avrà una sezione dedicata (sarà lì che calcoleremo l'errore), mentre possiamo subito parlare della funzione di attivazione.

La funzione di attivazione delle reti con delta rule è una *funzione lineare* F della somma, cioè:

$$y = F\left(\sum_{j=0}^p x_j^k w_j\right)$$

In Figura 3.12 vediamo la rappresentazione geometrica della funzione.

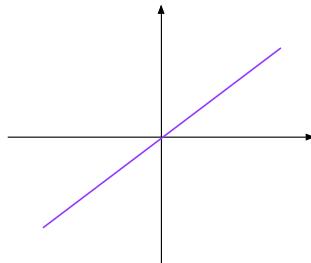


Figura 3.12: Funzione di attivazione φ per la rete a percettrone con delta rule.

Per semplicità, noi considereremo $F(x) = x$, ovvero una funzione che non fa nulla. Dunque, avremo che l'output della rete sarà:

$$y = \sum_{j=0}^p x_j^k w_j \quad (3.1)$$

cioè il valore della somma pesata degli input. Ai fini della categorizzazione due output saranno considerati "uguali" se sufficientemente vicini fra loro (ad esempio 0.9 e 1.1 possono essere considerati entrambi 1).

3.6.2 Cosa vogliamo fare

Riassumiamo quanto detto fin'ora:

- Vogliamo minimizzare l'errore globale della rete.
- Tale errore è funzione dei pesi della rete stessa.
- Dobbiamo quindi studiare la variazione di errore correlata alla variazione dei pesi (cioè una derivata).

- I pesi devono quindi essere modificati così da minimizzare l'errore.
- L'apprendimento termina quando l'errore scende sotto una determinata soglia.

Addentriamoci ora nel vivo della discussione, cercando di formalizzare il concetto di errore e di capirne l'uso nell'algoritmo di apprendimento.

3.6.3 Come modificare un peso

Supponiamo di voler modificare un determinato peso w_i della rete. Supponiamo di poter calcolare l'errore E in funzione della variazione del valore del peso w_i (dopo vedremo come e a che cosa si riferisce questo errore). Siamo quindi in grado di tracciare il grafico in Figura 3.13.

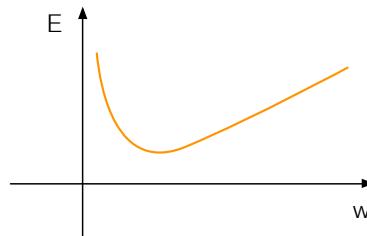


Figura 3.13: Andamento dell'errore al variare del peso w_i .

Se quindi calcoliamo la derivata di tale funzione:

$$f(w_i) = \frac{\partial E}{\partial w_i} + c$$

siamo in grado di capire come la variazione del peso apporti contributo positivo o negativo all'errore. Più precisamente:

- Se $\frac{\partial E}{\partial w_i} < 0$ allora E diminuisce quando w_i aumenta (cioè i due sono inversamente proporzionali), dunque, per diminuire l'errore si deve *aumentare* il peso.
- Se $\frac{\partial E}{\partial w_i} > 0$ allora E aumenta quando w_i aumenta (cioè i due sono direttamente proporzionali), dunque, per diminuire l'errore si deve *diminuire* il peso.

In pratica, qualunque sia il valore derivata, possiamo usare questa funzione di correzione dei pesi:

$$w_i(t+1) = w_i(t) - \eta \cdot \left(\frac{\partial E}{\partial w_i} \right)$$

Stando a quanto detto prima, infatti, è sufficiente aggiungere un aggiustamento al peso dell'iterazione precedente ($w_i(t)$) che sia di segno opposto rispetto alla derivata (ecco perché c'è un meno davanti a η). Abbiamo scelto che quel contributo sia proprio il valore della derivata, modulato dal solito learning rate.

La tecnica di ottimizzazione appena illustrata è detta *discesa del gradiente*.

Il vettore di gradienti

Quanto appena illustrato vale per un singolo peso w_i . Come sappiamo, però, una rete a perceptron può avere più neuroni ed ognuno di questi ha i pesi associati agli input che riceve. La modifica appena accennata dovrà dunque essere effettuata peso per peso, generando così il vettore dei gradienti:

$$\nabla E = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_n} \end{bmatrix} \quad (3.2)$$

Vedremo in seguito come usare questo vettore per correggere i pesi tutti insieme.

3.6.4 Calcolare l'errore globale

Andiamo ora a capire come si calcola E , l'errore globale della rete. Iniziamo col definire $E^k(w)$, ovvero l'errore relativo ad un dato pattern k (definito come (x^k, d^k)) e per una determinata configurazione di pesi data dal vettore w . Attenzione: per "configurazione" intendiamo l'insieme dei vettori dei pesi di ogni neurone della rete. Definiamo tale errore come segue:

$$E^k(w) = \frac{1}{2} \sum_{j \in \text{Output}} (d_j^k - y_j^k)^2$$

dove con j scorriamo tutti i neuroni della rete, e per ciascuno di essi calcoliamo la differenza fra d^k e y^k (calcolato come in Equazione 3.1), cioè la differenza fra il valore desiderato per il pattern k preso in considerazione e il valore effettivamente ottenuto dalla rete per quel pattern. Tale differenza è poi elevata al quadrato. Ognuna di queste differenze al quadrato viene infine sommata e divisa per due, così da darci E^k .

Ora che siamo in grado di stabilire l'errore su un singolo pattern, risulta intuitiva la definizione di errore globale:

$$E_{\text{TOT}}(w) = \sum_{k=1}^N E^k(w)$$

Andiamo semplicemente a sommare, per ognuno degli N pattern, l'errore sul singolo pattern. Possiamo anche definire l'errore medio:

$$E_{\text{AVG}}(w) = \frac{E_{\text{TOT}}(w)}{N}$$

Riassumendo, il nostro intento è quello di minimizzare E_{TOT} e lo facciamo *pattern by pattern*, calcolando e minimizzando l'errore E^k per ogni pattern k .

Quindi, quando abbiamo parlato di errore globale E in precedenza, abbiamo inteso in realtà E^k , l'errore globale per un determinato pattern (non avrebbe però avuto senso introdurre la notazione fin da subito, sarebbe stata confusoria).

Questo approccio pattern by pattern risulta comune a quello già visto nella rete a percettrone.

3.6.5 Come modificare i pesi: una formula per tutti

Sappiamo che per modificare un singolo peso, dobbiamo calcolare:

$$w_{ji}(k+1) = w_{ji}(k) - \eta \left(\frac{\partial E^k}{\partial w_{ji}} \right)$$

Notiamo due cose importanti dal punto di vista della notazione: con i indichiamo un determinato peso (non vettore!) di un certo neurone j , mentre con k indichiamo il pattern su cui stiamo lavorando. Dato però che ogni iterazione coinvolge un pattern soltanto, con k identifichiamo anche l'iterazione a cui ci troviamo. Ovviamente anche il peso w_{ji} che dobbiamo derivare si riferisce ad una determinata iterazione k , ma per semplicità di lettura evitiamo di trascrivere la parentesi.

Per comodità, definiamo anche:

$$\Delta w_{ji}(k) = -\eta \left(\frac{\partial E^k}{\partial w_{ji}} \right)$$

così da poter scrivere più agilmente:

$$w_{ji}(k+1) = w_{ji}(k) + \Delta w_{ji}(k)$$

Se poi ragioniamo su un neurone per intero (quindi usando il vettore di pesi del neurone j , cioè w_j) possiamo anche scrivere:

$$w_j(k+1) = w_j(k) - \eta \nabla E^k(w) = \begin{bmatrix} w_{j0} - \eta \frac{\partial E^k}{\partial w_{j0}} \\ \vdots \\ w_{jn} - \eta \frac{\partial E^k}{\partial w_{jn}} \end{bmatrix}$$

ovvero, in sostanza, la sottrazione fra tutti gli elementi del vettore dei pesi w_j e tutti gli elementi del vettore dei gradienti (Equazione 3.2) moltiplicati per η .

Il nostro intento è quindi quello di calcolare $\Delta w_{ji}(k)$. Andiamo a lavorare su questa formula.

$$\Delta w_{ji}(k) = -\eta \left(\frac{\partial E^k}{\partial w_{ji}} \right) = -\eta \frac{\partial}{\partial w_{ji}} \frac{1}{2} \sum_{t \in \text{Output}} \left(d_t^k - \sum_{g=0}^p x_g^k w_{tg} \right)^2$$

Non abbiamo fatto nulla di particolare: abbiamo sostituito E^k con la sua definizione. Per fare ciò abbiamo dovuto introdurre i due nuovi indici. Abbiamo usato t per scorrere tutti i neuroni della rete¹ e g per scorrere tutti gli input (e calcolare così la distanza il valore in output dalla rete).

Se esplodiamo la sommatoria relativa ai neuroni, otteniamo:

$$\begin{aligned} \Delta w_{ji}(k) = & -\eta \left[\right. \\ & \frac{\partial}{\partial w_{ji}} \frac{1}{2} \left(d_1^k - \sum_{g=0}^p x_g^k w_{1g} \right)^2 + \\ & \dots + \\ & \frac{\partial}{\partial w_{ji}} \frac{1}{2} \left(d_j^k - \sum_{g=0}^p x_g^k w_{jg} \right)^2 + \\ & \dots + \\ & \left. \frac{\partial}{\partial w_{ji}} \frac{1}{2} \left(d_n^k - \sum_{g=0}^p x_g^k w_{ng} \right)^2 \right] \end{aligned}$$

Abbiamo esplicitato il j -esimo elemento della sommatoria, che è proprio il calcolo relativo al neurone j di cui vogliamo aggiustare i pesi. Andando ad eseguire le derivate addendo per addendo, notiamo però che sono tutte zero tranne quella dell'elemento j !

Questo è naturale: modificare i pesi del nodo w_j (denominatore della derivata) non può modificare l'output degli altri $n-1$ neuroni (che sfruttano per il calcolo i vari pesi w_t con $t \neq j$) e dunque non può neanche modificare l'errore (il cui calcolo è semplicemente fra l'output di quei neuroni e l'output atteso). Si annullano dunque tutti gli addendi tranne il j -esimo:

$$\Delta w_{ji}(k) = -\eta \frac{\partial}{\partial w_{ji}} \frac{1}{2} \left(d_j^k - \sum_{g=0}^p x_g^k w_{jg} \right)^2$$

¹L'errore E^k è sì riferito ad un determinato pattern, ma è comunque globale, cioè riferito a tutta la rete. Dunque, se anche se vogliamo modificare i pesi per un determinato neurone j , abbiamo comunque bisogno dell'errore globale.

Calcolando la derivata otteniamo:

$$\Delta w_{ji}(k) = -\eta \frac{1}{2} \cdot 2 \left(d_j^k - \sum_{g=0}^p x_g^k w_{jg} \right) \cdot (-1) \cdot x_i^k$$

che diventa dunque:

$$\Delta w_{ji}(k) = \eta(d_j^k - y_j^k)x_i^k$$

Si ponga particolare attenzione al cambio di segno.

Siamo finalmente riusciti a calcolare $\Delta w_{ji}(k)$, ovvero la quantità che dobbiamo sommare al valore del peso i del neurone j in fase di calcolo di $w_{ji}(k+1)$. Questo valore è dunque relativo ad *un peso soltanto di un determinato neurone*.

Volendo, è possibile pensare alla formula in una notazione vettoriale, più comoda per i calcoli (poiché modifica tutti i pesi relativi ad un neurone j in una volta soltanto):

$$w_j(k+1) = w_j(k) + \eta(d_j^k - y_j^k)x^k$$

dove con x^k intendiamo i valori del pattern preso in considerazione all'iterazione k e con $w_j(k)$ intendiamo l'intero vettore dei pesi del neurone j . In questo modo, $\eta(d_j^k - y_j^k)$ può essere visto come una sorta di coefficiente di modifica, uguale per tutti gli input e tutti i pesi di un determinato neurone j .

Questa formula dà il nome all'intera rete ed è chiamata *delta rule*.

3.6.6 Lo pseudocodice

Riportiamo ora lo pseudocodice dell'algoritmo di apprendimento che sfrutta la delta rule.

```
Dato un training set ( $x^k, d^k$ )
Inizializza tutti i pesi casualmente con piccoli valori.

while ( $E_{TOT}$  non ha raggiunto un valore limite pre-impostato
      oppure le iterazioni superano un certo limite) {

    Prendi un esempio del training set ( $x^k, d^k$ ).
    Calcolare la risposta  $y^k$  della rete fornendo  $x^k$  come input.
    Applicare la Delta Rule su ogni neurone.

}
```

Listing 3.5: Pseudocodice per l'algoritmo di apprendimento delle reti a percettrone con delta rule.

L'inizializzazione dei pesi con valori piccoli permette di raggiungere la convergenza in minor tempo. Parleremo della convergenza di questa architettura a breve.

3.6.7 Un breve esempio

Consideriamo il solito problema dell'OR:

A: ([1, 1, 1], 1)
B: ([1, 1, 0], 1)
C: ([1, 0, 1], 1)
D: ([1, 0, 0], -1)

Listing 3.6: Problema dell'OR per una rete a percettrone (con bias incluso).

E consideriamo una rete a tre input ed un solo neurone con pesi:

$$\begin{aligned} w_0 &= 0 \\ w_1 &= 0.1 \\ w_2 &= -0.2 \end{aligned}$$

Consideriamo inoltre un $\eta = 0.1$, mentre non diamo una tolleranza sull'errore globale (eseguiremo soltanto un paio di passi).

Iterazione 1. Testiamo gli attuali pesi con il pattern *A*:

$$w^T \cdot x = -0.1$$

La funzione di attivazione restituisce -0.1 mentre noi volevamo 1. Modifichiamo quindi i pesi dell'unico neurone che abbiamo:

$$\begin{aligned} w(2) &= w(1) + \eta(d_k - y_j)x \\ &= [0 \ 0.1 \ -0.2] + 0.1(1 - (-0.1)) [1 \ 1 \ 1] \\ &= [0 \ 0.1 \ -0.2] + 0.11 [1 \ 1 \ 1] \\ &= [0.11 \ 0.21 \ -0.09] \end{aligned}$$

Abbiamo adottato la notazione vettoriale per comodità. In presenza di più neuroni, questo calcolo dorebbe essere iterato su ciascuno di essi (modificando ovviamente i valori dei pesi e l'output).

A questo punto, si dovrebbe calcolare l'errore globale per ogni pattern e ottenere così l'errore totale. Dunque, si dovrebbe verificare se quel valore è entro la tolleranza. Avendo un solo neurone e quattro pattern, il calcolo è molto rapido:

$$\begin{aligned} E_{\text{TOT}}(w) &= \sum_{k=1}^4 E^k(w) = \\ &\quad \frac{1}{2}(d^A - y^A)^2 + \frac{1}{2}(d^B - y^B)^2 + \\ &\quad \frac{1}{2}(d^C - y^C)^2 + \frac{1}{2}(d^D - y^D)^2 \end{aligned}$$

Come si vede, in ogni addendo è stata rimossa la sommatoria sui neuroni (poiché ne abbiamo uno soltanto), e al posto di k è stato inserito il "nome" del pattern. Si ha quindi che:

$$E_{TOT}(w) = \frac{1}{2}[(1 - y^A)^2 + (1 - y^B)^2 + (1 - y^C)^2 + (-1 - y^D)^2]$$

Si lascia al lettore il calcolo di y^A, y^B, y^C, y^D e lo sviluppo di qualche iterazione successiva.

3.6.8 Sulla convergenza

Le garanzie di convergenza in merito alle reti che sfruttano la delta rule sono più deludenti di quelle viste con le reti a percettrone classiche.

Infatti, abbiamo garanzia di convergenza soltanto se il problema è *linearmente indipendente*, che è una condizione più restrittiva rispetto alla separabilità. Più precisamente, se un problema è linearmente indipendente allora è anche linearmente separabile, ma non vale il contrario.

Ricordiamo che due vettori \vec{v}_k e \vec{v}_q sono linearmente indipendenti se combinandoli linearmente l'unico modo di ottenere il vettore vuoto è assegnarvi coefficienti a_k e a_q pari a 0 (cioè tutti i coefficienti assegnati ai vettori devono essere nulli). Ovvero:

$$\vec{0} = a_k \vec{v}_k + a_q \vec{v}_q$$

è vero solo se $a_k = a_q = 0$ (se infatti non fosse vero allora potremmo scrivere $a_k \vec{v}_k = -a_q \vec{v}_q$ e quindi i due vettori sarebbero combinazione lineare l'uno dell'altro e più non linearmente indipendenti).

Se ne deduce che un problema che abbia un numero di pattern maggiore della quantità dei valori di input di ogni singolo pattern non sia linearmente indipendente. Questo è banalmente vero, poiché se vi sono più pattern che valori è certo che almeno uno dei pattern sia combinazione lineare degli altri.

Capitolo 4

Reti multilivello

Nel 1969 Minsky e Papert criticano fortemente le reti neurali poiché non in grado di risolvere problemi classici di fondamentale importanza (XOR, parità e connettività su tutti). Questa critica porta una ventata di pessimismo che blocca la ricerca in questo campo, fino all'introduzione delle reti *multilivello*, pensate da David Rumelhart, Geoffrey Hinton e Ronald Williams. È nel 1986 che pubblicano l'articolo "Learning Internal Representations by Error Propagation", rivitalizzando l'intero settore e introducendo le reti multilivello così come le conosciamo oggi.

4.1 Implementare lo XOR: un esempio introduttivo

Prima di andare a definire precisamente cosa siano le reti multilivello e di studiarne l'algoritmo di apprendimento, facciamo un piccolo esempio introduttivo così da poter toccare con mano la potenza di queste reti. Riprendiamo il problema dello XOR (Listing 4.1) che le nostre reti a percettrone non erano in grado di risolvere.

```
([1, 1, 1], -1)
([1, 1, 0], 1)
([1, 0, 1], 1)
([1, 0, 0], -1)
```

Listing 4.1: Problema dello XOR (con bias incluso).

L'idea principale è che una rete multilivello sfrutta dei neuroni intermedi per poter risolvere parti del problema principale. Il livello finale, poi, combina tutta l'elaborazione e fornisce in output il risultato. Nel caso dello XOR, possiamo formalizzare il problema come segue:

$$x_1 \oplus x_2 = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$$

Ora, saremmo in grado di costruire due reti per i due AND nelle parentesi e anche una rete che risolve un OR. Vediamo in Figura 4.1 la rete che possiamo

sfruttare per risolvere il problema dello XOR: Abbiamo usato la funzione di

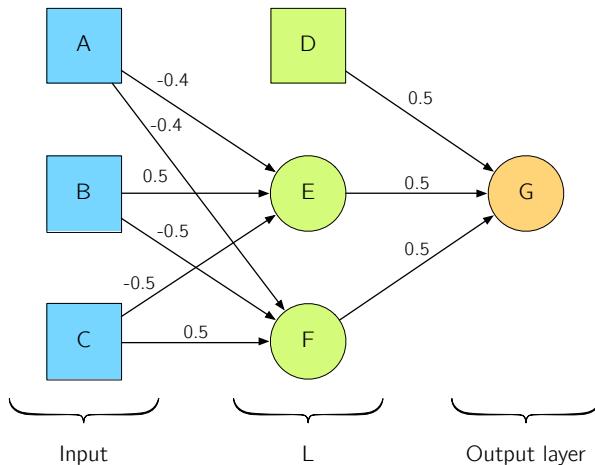


Figura 4.1: Rete neurale multilivello che risolve il problema dello XOR. Rappresentiamo con quadrati gli input/bias e con tondi i neuroni veri e propri.

attivazione discreta riportata in Figura 3.3. In sostanza il neurone E risolve la prima parantesi, mentre il neurone F risolve la seconda. Il neurone G, infine, implementa l'OR fra le due parti. Si noti che il neurone G riceve in input +1 o -1 (non più 0 ed 1), che sono l'output di E ed F. I livelli intermedi (fra l'input e l'ultimo livello) vengono dette *hidden units*.

In Figura 4.2 vediamo la rappresentazione geometrica di quanto appena descritto.

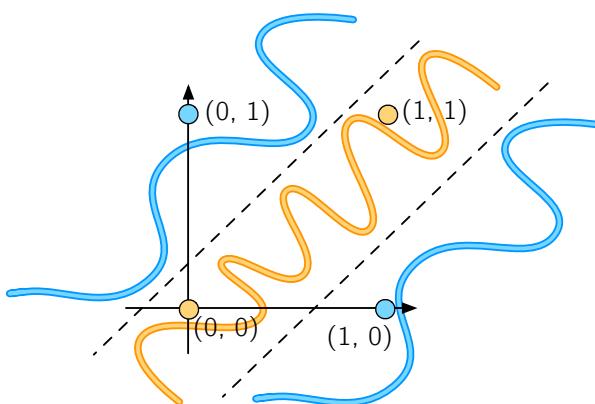


Figura 4.2: Le hidden unit permettono di fornire due rette diverse, la cui combinazione permette di trovare tre porzioni di piano (categorizzate come due insiemi diversi).

4.2 Reti back propagation: funzione di attivazione ed architettura

Andiamo ora ad introdurre più formalmente i concetti che abbiamo visto poc' anzi.

Come sappiamo, la composizione di funzioni lineari è a sua volta una funzione lineare. Se ne deduce che la funzione di attivazione non possa essere lineare come lo era per la regola delta, altrimenti ci ricondurremmo al caso di reti a un layer soltanto. La funzione di attivazione scelta è quindi la funzione sigmoide (Figura 4.3).

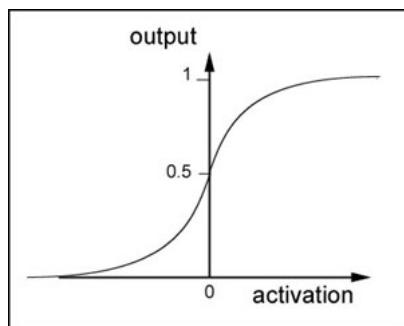


Figura 4.3: La funzione sigmoide

Perciò, l'output di un certo neurone j sarà:

$$y_j = \frac{1}{(1 + e^{-\text{netinput}_j})}$$

Dal punto di vista dell'architettura c'è un'interessante teorema: dato un numero sufficiente di hidden units, tutto ciò che può essere risolto da una rete multilayer a n livelli può essere risolto anche da una rete a due livelli. Pertanto nei nostri esempi ci limiteremo ad usare soltanto due livelli.

4.2.1 Definizioni preliminari

Diamo ora alcune definizioni preliminari in merito alla rete, che sono in gran parte comuni alle reti a percettrone che abbiamo già visto.

Netinput per neuroni a livello 1. Dato il neurone j a livello 1 il suo netinput sarà pari a:

$$v_j = \sum_{i=0}^m w_{ji}x_i$$

dove $x_1 \cdot x_m$ sono i valori di input del pattern considerato e x_0 è il bias.

Netinput per neuroni a livello > 1. Dato il neurone j ad un livello maggiore di 1 il suo netinput sarà pari a:

$$v_j = \sum_{i=0}^m w_{ji} y_i$$

dove $y_1 \cdot y_m$ sono i valori di output dei neuroni che precedono (al livello precedente) j e y_0 è il bias.

Output dei neuroni. Dato il neurone j (quale che sia il suo livello) il suo output sarà pari a:

$$y_j = \varphi(v_j) = \frac{1}{(1 + e^{-v_j})}$$

Chiaramente l'output dipende da v_j (l'input) che a sua volta dipenderà dal livello a cui il neurone si trova.

4.3 Addestrare le reti

L'algoritmo di addestramento è molto simile a quello già visto per la regola delta. Infatti, si cerca di minimizzare l'errore sfruttando la derivata, così come abbiamo già studiato. Per la delta rule avevamo che, per un neurone j :

- Se $\frac{\partial E}{\partial w_j} < 0$ allora E diminuisce quando w_j aumenta (cioè i due sono inversamente proporzionali), dunque, per diminuire l'errore si deve *aumentare* il peso.
- Se $\frac{\partial E}{\partial w_j} > 0$ allora E aumenta quando w_j aumenta (cioè i due sono direttamente proporzionali), dunque, per diminuire l'errore si deve *diminuire* il peso.

Il nostro intento è quindi quello di, pattern by pattern, calcolare l'errore. L'errore di un determinato neurone j rispetto ad un pattern n sarà quindi calcolato come:

$$e_j(n) = d_j(n) - y_j(n) \quad (4.1)$$

La notazione è leggermente differente (usiamo n al posto di k per definire l'iterazione/pattern). La funzione che utilizziamo per l'errore sul singolo neurone è quindi:

$$E_j(n) = \frac{1}{2} e_j^2(n)$$

Che, iterata su ogni neurone definisce:

$$E(n) = \frac{1}{2} \sum_{j \in \text{Output}} e_j^2(n)$$

Fino a qui, a meno di differenze notazionali, è rimasto tutto invariato rispetto a quanto visto per la delta rule. L'approccio che usiamo è infatti identico: andremo ad ogni iterazione a modificare i pesi sfruttando $E(n)$ (errore globale relativo al pattern preso in considerazione), sperando così di diminuire l'errore medio sulla rete (il cui calcolo sarebbe però costoso).

Inizia ora una lunga avventura al termine della quale saremo in grado di calcolare la variazione di peso sia per i neuroni hidden che quelli output. Il calcolo, infatti, è diverso e affronteremo separatamente i due casi.

4.3.1 Calcolare la variazione di peso per i neuroni output

Consideriamo ora un neurone j che si trova all'ultimo livello della rete. Consideriamo poi il peso w_{ji} che pesa l'input y_i , arrivatogli dal neurone i . L'obiettivo è sempre lo stesso; data la formula di modifica dei pesi:

$$w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}$$

vogliamo calcolare Δw_{ji} , cioè come modificare i pesi. Come detto sopra, vogliamo modificare i pesi andando in senso opposto rispetto alla derivata, ovvero:

$$\Delta w_{ji} = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)}$$

dove, come sempre η rappresenta il learning rate. Siamo quindi interessati a calcolare la variazione dell'errore rispetto al variare del peso. Abbiamo già effettuato questo calcolo per la delta rule, ma questa volta la definizione di $E(n)$ è leggermente diversa:

$$E(n) = \frac{1}{2} \sum_{t \in \text{Output}} \left(d_t(n) - y_t(n) \right)^2 \quad (4.2)$$

Ogni y_t (output del neurone t) è però definito come $y_t = \varphi(v_t)$ cioè come il risultato della funzione di attivazione sul suo input v_t . Abbiamo perciò, espandendo la somma:

$$E(n) = \frac{1}{2} [(d_1 - \varphi(v_1(n)))^2 + \dots + (d_j - \varphi(v_j(n)))^2 + \dots]$$

fino ad arrivare all'ultimo neurone del livello. Sappiamo anche che ognuno di egli input v_t è definito come: $v_t = \sum_{i=0}^m w_{ti} y_i$, dove l'indice i scorre fra tutti i

neuroni al livello precedente di t . Abbiamo perciò:

$$\begin{aligned} E(n) = & \frac{1}{2} \left[\right. \\ & \left(d_1 - \varphi \left(\sum_{i=0}^m w_{1i}(n) y_i(n) \right) \right)^2 + \\ & \dots + \\ & \left(d_j - \varphi \left(\sum_{i=0}^m w_{ji}(n) y_i(n) \right) \right)^2 + \\ & \dots \\ & \left. \right] \end{aligned}$$

Abbiamo quindi che $E(n)$ è in realtà una funzione di funzione: più precisamente, è una funzione $E(v(w))$. Possiamo quindi usare la formula di derivazione delle funzioni composte (di cui faremo vastissimo uso da qui in avanti):

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x} \quad (4.3)$$

ed ottenere:

$$\Delta w_{ji} = -\eta \frac{\partial E(n)}{\partial v_j(n)} \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)} \quad (4.4)$$

Si ponga molta attenzione alle lettere e ai pedici. Abbiamo sostanzialmente scomposto la nostra derivata iniziale, ottenendo due differenti derivate.

Esaminiamo i due moltiplicandi (tralasciamo η semplicità):

- $-\eta \frac{\partial E(n)}{\partial v_j(n)}$: rappresenta il variare dell'errore globale del pattern rispetto al netinput del neurone j . È anche detto *gradiente locale* e lo chiameremo δ_j .

Intuitivamente, δ_j indica come modificare l'input del neurone j (v_j):

- se $\delta_j > 0$ allora v_j deve essere aumentato.
- se $\delta_j < 0$ allora v_j deve essere diminuito.

Chiaramente il gradiente locale è solo parte del calcolo, starà anche alla seconda componente contribuire (positivamente o negativamente) al valore di w_{ji} per sapere se aumentare o meno il peso. Intanto, δ_j ci dà informazioni in merito a come debba comportarsi l'input di j .

- $\frac{\partial v_j(n)}{\partial w_{ji}(n)}$: rappresenta il variare dell'input a j rispetto alla variazione del peso w_{ji} .

Abbiamo quindi ottenuto *due diverse componenti* che dobbiamo calcolare separatamente per poter ottenere w_{ji} . Si tenga presente che nonostante possano sembrare simili, le due componenti rappresentano concetti estremamente diversi. Questa fase ci ha permesso sostanzialmente di dire che la variazione dell'errore globale rispetto ai pesi è dipendente dalla variazione dell'errore globale rispetto all'input del neurone moltiplicata per la variazione dell'input del neurone rispetto al peso.

Calcolo di δ_j . Andiamo ora a calcolare la prima componente, δ_j . Ricordiamo che:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial v_j(n)}$$

Andando ad esplodere $E(n)$, otteniamo:

$$\delta_j(n) = -\frac{\partial}{\partial v_j(n)} \cdot \frac{1}{2} [(d_1 - \varphi(v_1(n)))^2 + \dots + (d_j - \varphi(v_j(n)))^2 + \dots]$$

Come sempre, andando a derivare di tutti questi elementi rimane solo quello dove figura v_j . Andandone a calcolare la derivata otteniamo:

$$\delta_j(n) = -(d_j - \varphi(v_j(n)))(-1)\varphi'(v_j(n)) = (d_j - \varphi(v_j(n)))\varphi'(v_j(n))$$

Riconosciamo nel primo moltiplicando la definizione di errore per il neurone j (Equazione 4.1) e quindi possiamo scrivere:

$$\delta_j(n) = e_j \varphi'(v_j(n)) \quad (4.5)$$

Abbiamo risolto la prima delle due derivate necessarie per calcolare w_{ji} (ricordiamo l'Equazione 4.4). Abbiamo trovato che la variazione dell'errore globale rispetto all'input del neurone è dato dall'errore su quel neurone, moltiplicato per la derivata della funzione di attivazione per il valore di input.

Calcolo della seconda componente. Non resta che calcolare:

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

Come sappiamo,

$$v_j(n) = \sum_{i=0}^m w_{ji} y_i$$

E dunque, possiamo scrivere:

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = \frac{\partial}{\partial w_{ji}(n)} \cdot [w_{j1}y_1 + \dots + w_{ji}y_i + \dots + w_{jm}y_m]$$

Ancora una volta, l'unico elemento che rimarrà andando a derivare sarà quello che coinvolge w_{ji} . Perciò, possiamo scrivere che:

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

Abbiamo risolto anche la seconda parte! Abbiamo in questo caso trovato che la variazione dell'input da i al neurone j rispetto al peso a lui assegnato è dipendente dall'output del neurone entrante (il che è decisamente sensato).

Unire le equazioni: il calcolo dell'incremento

Andando ora ad unire l'Equazione 4.3.1 e l'Equazione 4.5 siamo finalmente in grado di scrivere:

$$\Delta w_{ji} = \eta \cdot \delta_j \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)} = \eta \cdot e_j \cdot \varphi'(v_j(n)) \cdot y_i(n)$$

Sostituendo la derivata della funzione sigmoide $\varphi'(x) = \varphi(x) \cdot (1 - \varphi(x))$ si ha:

$$\Delta w_{ji} = \eta \cdot e_j \cdot \varphi(v_j(n))(1 - \varphi(v_j(n))) \cdot y_i(n)$$

Dopodiché, per semplificare la notazione possiamo anche sostituire a $\varphi(v_j)$ il suo corrispettivo y_j (ricordiamo che l'applicazione della funzione di attivazione sul netinput è ovviamente l'output):

$$\Delta w_{ji} = \eta \cdot e_j \cdot y_j(n)(1 - y_j(n)) \cdot y_i(n)$$

La formula così scritta risulta abbastanza leggibile. In fase di calcolo sostituiamo anche e_j , ottenendo così:

$$\Delta w_{ji} = \eta \cdot (d_j(n) - y_j(n)) \cdot y_j(n)(1 - y_j(n)) \cdot y_i(n) \quad (4.6)$$

Ricordiamo che chiaramente Δw_{ji} è solo la quantità necessaria per aggiustare il peso w_{ji} , dunque la formula di aggiornamento totale risulta essere:

$$w_{ji}(n+1) = w_{ji}(n) + \eta \cdot (d_j(n) - y_j(n)) \cdot y_j(n)(1 - y_j(n)) \cdot y_i(n)$$

Incremento o decremento? Studiamo i segni

Andiamo brevemente ad osservare la formula per cercare di capire cosa succede effettivamente a seconda che i valori restituiti dalle componenti siano positivi o negativi. Abbiamo che (usando la notazione più breve possibile):

$$\Delta w_{ji} = \eta \cdot \delta_j \cdot y_i(n)$$

Dunque, se l'input verso j deve essere aumentato ($\delta_j > 0$) e anche $y_i(n) > 0$, si dovrà potenziare il peso. Altrimenti se $y_i(n) < 0$ il peso si dovrà indebolire. Si ragiona in maniera inversa nel caso in cui l'input debba essere diminuito.

4.3.2 Calcolare la variazione di peso per i neuroni hidden

La derivazione che abbiamo appena effettuato ci ha fornito la formula per calcolare Δw_{ji} , l'incremento di peso per un neurone j di output. Purtroppo la stessa formula non può essere applicata anche per un neurone hidden. Rivediamola:

$$\Delta w_{ji} = \eta \cdot e_j \cdot y_j(n)(1 - y_j(n)) \cdot y_i(n)$$

Se j fosse un neurone hidden, infatti, non saremo in grado di calcolare e_j poiché non disponiamo di d_j , ovvero l'output desiderato per quel neurone hidden. Il problema infatti ci dice solo quale vogliamo che sia l'output finale e non quello dei neuroni nascosti!

L'idea è quindi quella che l'errore delle hidden units si possa calcolare a partire dall'errore dei neuroni di output a cui quelle unità sono connesse. D'altronde, se l'output di un determinato neurone di output è troppo alto o troppo basso, questo dipende anche dall'input che riceve (e quindi dall'output del neurone hidden che lo precede). In realtà sappiamo già come dovrebbe variare l'input di un determinato neurone di output per poter diminuire l'errore: si tratta del suo gradiente locale δ_j .

Fino a qui sembra tutto semplice, ma un neurone hidden propaga il suo valore a tutti i neuroni a livello successivo, i quali possono avere gradiente locale discorde: in pratica qualche neurone vorrebbe più input e qualche altro neurone ne vorrebbe di meno (sempre al fine di minimizzare l'errore). Dovrebbe quindi essere chiaro (almeno intuitivamente) che le hidden unit dovranno tenere conto del gradiente locale dei neuroni al livello successivo per poter regolare i pesi delle sinapsi dei livelli precedenti.

Vediamo se dal punto di vista matematico la nostra intuizione ha senso oppure no (SPOILER: sì, ha senso).

Ripartiamo dalla formula iniziale già usata in precedenza, per cui:

$$\Delta w_{ji} = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} = -\eta \frac{\partial E(n)}{\partial v_j(n)} \cdot \frac{\partial v_j(n)}{\partial w_{ji}(n)} = \eta \delta_j(n) y_i(n)$$

Per ora nulla di nuovo, se non fosse che sappiamo che il secondo moltiplicando (δ_j) non possiamo calcolarlo poiché non possiamo valutare e_j (che compare nella sua definizione, Equazione 4.5). La formula che usiamo, come si vede, è "apparentemente" uguale, sia che si tratti di neuroni hidden che di neuroni di output. La differenza consta nella definizione della δ_j .

Cerchiamo dunque di definire il gradiente locale in modo che non appaia e_j . Per fare ciò applichiamo la regola di derivazione per funzioni composte (Equazione 4.3) e otteniamo:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial v_j(n)} = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \frac{\partial y_j(n)}{\partial v_j(n)}$$

Ottimo! Ora non resta che calcolare queste due derivate singolarmente e poi usare il tutto come δ_j nella formula di cui sopra. Il secondo moltiplicando è semplice da calcolare, infatti, giacché $y_j(n) = \varphi(v_j(n))$ possiamo scrivere:

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \frac{\partial \varphi(v_j(n))}{\partial v_j(n)} = \varphi'(v_j(n))$$

Un altro pezzetto è stato calcolato, e siamo in questa situazione:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \varphi'(v_j(n))$$

Andiamo ora eseguire la derivata mancante.

Calcolo della variazione dell'errore in funzione dell'output del neurone

Iniziamo col sostituire $E(n)$ con la sua definizione (una sommatoria degli errori, Equazione 4.2), dopodiché, sapendo che la derivata della somma è la somma delle derivate, spostiamo all'interno la derivazione:

$$\frac{\partial E(n)}{\partial y_j(n)} = \frac{\partial \frac{1}{2} \sum_{k \in \text{Output}} e_k^2(n)}{\partial y_j(n)} = \frac{1}{2} \sum_{k \in \text{Output}} \frac{\partial e_k^2(n)}{\partial y_j(n)}$$

Usiamo nuovamente la formula di derivazione per funzioni composte (Equazione 4.3) sulla derivata nella sommatoria ed otteniamo:

$$\frac{\partial E(n)}{\partial y_j(n)} = \frac{1}{2} \sum_{k \in \text{Output}} \frac{\partial e_k^2(n)}{\partial y_j(n)} = \frac{1}{2} \sum_{k \in \text{Output}} \frac{\partial e_k^2(n)}{\partial y_j(n)} \cdot \frac{\partial e_k(n)}{\partial y_j(n)}$$

Andiamo ad effettuare la derivata del primo moltiplicando e otteniamo:

$$\frac{\partial E(n)}{\partial y_j(n)} = \frac{1}{2} \sum_{k \in \text{Output}} \frac{\partial e_k^2(n)}{\partial y_j(n)} \cdot \frac{\partial e_k(n)}{\partial y_j(n)} = \sum_{k \in \text{Output}} e_k(n) \cdot \frac{\partial e_k(n)}{\partial y_j(n)}$$

Usiamo ancora una volta la formula di derivazione per funzioni composte ed otteniamo:

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_{k \in \text{Output}} e_k(n) \cdot \frac{\partial e_k(n)}{\partial y_j(n)} = \sum_{k \in \text{Output}} e_k(n) \cdot \frac{\partial e_k(n)}{\partial v_k(n)} \cdot \frac{\partial v_k(n)}{\partial y_j(n)}$$

Siamo dunque giunti a questo punto:

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_{k \in \text{Output}} e_k(n) \cdot \frac{\partial e_k(n)}{\partial v_k(n)} \cdot \frac{\partial v_k(n)}{\partial y_j(n)}$$

Cerchiamo ora di risolvere separatamente le ultime due derivate. Noi sappiamo che:

$$e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi(v_k(n))$$

e dunque possiamo risolvere la prima delle due derivate scrivendo:

$$\frac{\partial e_k(n)}{\partial v_k(n)} = \frac{\partial d_k(n) - \varphi(v_k(n))}{\partial v_k(n)} = \varphi'(v_k(n))$$

Ok, prima derivata risolta. Per quanto concerne la seconda derivata, sappiamo che:

$$v_k(n) = \sum_{t=0}^m w_{kt}(n) y_t(n)$$

e dunque possiamo scrivere:

$$\frac{\partial v_k(n)}{\partial y_j(n)} = \frac{\partial \sum_{t=0}^m w_{kt}(n) y_t(n)}{\partial y_j(n)} = \sum_{t=0}^m \frac{\partial w_{kt}(n) y_t(n)}{\partial y_j(n)}$$

Ancora una volta, di tutti gli elementi della somma rimarrà diverso da zero solo quello per cui $t = j$ e dunque avremo:

$$\frac{\partial v_k(n)}{\partial y_j(n)} = \frac{\partial w_{kj}(n) y_j(n)}{\partial y_j(n)} = w_{kj}(n)$$

Abbiamo risolto le due derivate separatamente e possiamo sostituirle nella formula sopra:

$$\frac{\partial E(n)}{\partial y_j(n)} = - \sum_{k \in \text{Output}} e_k(n) \cdot \varphi'(v_k(n)) \cdot w_{kj}(n)$$

Ma i primi due addendi della sommatoria sono la definizione di δ_k ! Perfetto! Possiamo dire quindi:

$$\frac{\partial E(n)}{\partial y_j(n)} = - \sum_{k \in \text{Output}} \delta_k \cdot w_{kj}(n)$$

Si ponga *estrema* attenzione a quello che è appena successo: il gradiente locale di cui si parla qui è quello di k , neurone ad un livello *successivo* rispetto a j , il neurone hidden sul quale stiamo lavorando. Siamo quindi in grado di calcolare questo gradiente poiché j sarà o un neurone output oppure un altro neurone hidden per cui, però, abbiamo già calcolato il gradiente (proprio con la formula che stiamo calcolando).

Non ci deve stupire questo risultato, poiché del tutto in linea rispetto a quanto avevamo intuito a inizio paragrafo: la variazione di errore rispetto all'output del neurone j -esimo (hidden) dipende dal valore del gradiente locale dei i neuroni (si noti la sommatoria, ci si basa su tutti i neuroni al livello successivo) che lo succedono. Quei gradienti sono poi modulati dal peso della sinapsi che intercorre fra il neurone hidden e quello a livello successivo (sinapsi da k a j). È evidente quindi la ragione per cui l'algoritmo si chiama *back propagation*:

l'errore è propagato all'indietro.

Torniamo a noi e concludiamo la derivazione matematica mettendo i pezzi insieme. Eravamo rimasti a:

$$\delta_j(n) = -\frac{\partial E(n)}{\partial y_j(n)} \cdot \varphi'(v_j(n))$$

ed ora siamo in grado di aggiungere l'ultimo tassello:

$$\delta_j(n) = \varphi'(v_j(n)) \cdot \sum_{k \in \text{Output}} \delta_k \cdot w_{kj}(n) \quad (4.7)$$

Non resta che scrivere esplicitamente la derivata della funzione sigmoide:

$$\delta_j(n) = y_j(n) \cdot (1 - y_j(n)) \cdot \sum_{k \in \text{Output}} \delta_k \cdot w_{kj}(n)$$

Ecco quindi che abbiamo il δ_j su misura per i neuroni hidden, da inserire nella solita formula di variazione del peso:

$$\Delta w_{ji} = \eta \cdot \delta_j \cdot y_i(n)$$

4.3.3 Il punto della situazione: dove usare le formule

È stato bello attraversare tutti questi calcoli, ma facciamo il punto della situazione.

Le reti back propagation si basano sul concetto di "propagare all'indietro l'errore".

L'algoritmo di apprendimento prende in esame un determinato pattern ad ogni iterazione e su questo valuta un errore globale, cercando così di modificare tutti i pesi al fine di minimizzarlo. Per fare ciò, per ogni peso $w_{ji}(n)$ (peso dal neurone i al neurone j per il pattern n) si calcola un certo $\Delta w_{ji}(n)$ che vi verrà sommato al fine di ottenere $w_{ji}(n+1)$, cioè il peso da usarsi all'iterazione successiva. La formula di aggiornamento dei pesi è dunque la seguente:

$$w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}(n) \quad (4.8)$$

e, qualsiasi sia la posizione del neurone j , abbiamo che:

$$\Delta w_{ji} = \eta \delta_j(n) y_i(n) \quad (4.9)$$

Il valore $\delta_j(n)$ è detto gradiente locale e a seconda che il neurone j a cui si riferisce sia di hidden o di output, il suo valore è calcolato diversamente.

- Per i neuroni di output si usa:

$$\delta_j = (d_j(n) - y_j(n)) \cdot y_j(n)(1 - y_j(n)) \quad (4.10)$$

- Per i neuroni hidden si usa:

$$\delta_j = y_j(n) \cdot (1 - y_j(n)) \cdot \sum_{k \in \text{Output}} \delta_k w_{kj}(n) \quad (4.11)$$

Si ponga particolare attenzione al fatto che i neuroni k su cui itera la somma sono i neuroni al livello successivo di j e quindi siamo in grado di calcolare (in realtà abbiamo già calcolato, seguendo la ricorsione) il loro δ_k .

Tutti i ragionamenti appena esplicati riguardano l'aggiornamento di *un peso soltanto alla volta*.

4.3.4 Un esempio riassuntivo

Mettiamo in pratica le nostre formule in un esempio riassuntivo. Consideriamo il problema e la rete neurale in Figura 4.4.

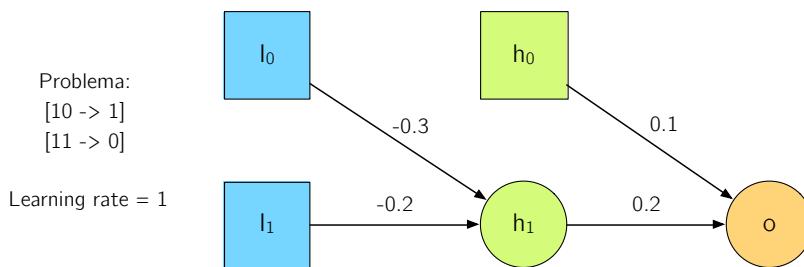


Figura 4.4: Una problema ed una rete neurale per risolverlo (i pesi sono assegnati a caso). i_0 ed h_0 sono bias. Con h indichiamo le hidden unit, con i gli input e con o l'output. Il learning rate η è pari a 1.

Il problema è molto semplice, si tratta in sostanza di trovare una rete che implementi il NOT. Consideriamo il primo pattern (iterazione 1).

Per comodità abbiamo già calcolato i valori di output per i vari neuroni alla prima iterazione:

$$\begin{aligned} y_{h_1}(1) &= 0.426 \\ y_o(1) &= 0.546 \end{aligned}$$

L'output della rete è quindi pari a 0.546, mentre noi desidereremmo un valore pari ad 1. Andiamo quindi a modificare i pesi dei vari neuroni al fine di minimizzare l'errore.

Modifica dei pesi per il neurone di output o

Abbiamo un solo neurone di output il quale ha due input (h_0 ed h_1). Dobbiamo quindi calcolare $\Delta w_{o,h_0}$ e $\Delta w_{o,h_1}$ per modificare i pesi. Usiamo dunque

I' Equazione 4.9 e iniziamo col calcolarci il gradiente locale per i neuroni di output (Equazione 4.10):

$$\delta_o(1) = e_o(1) \cdot y_o(1)(1 - y_o(1)) = (1 - 0.546) \cdot 0.546(1 - 0.546) = 0.112$$

Siamo ora in grado di calcolare i due Δw :

$$\Delta w_{o,h_0}(1) = \eta \cdot \delta_o(1) \cdot y_{h_0}(1) = 1 \cdot 0.112 \cdot 1 = 0.110$$

$$\Delta w_{o,h_1}(1) = \eta \cdot \delta_o(1) \cdot y_{h_1}(1) = 1 \cdot 0.112 \cdot 0.426 = 0.047$$

Si ponga particolare attenzione al fatto che l'output di un bias è semplicemente pari a quanto questo trasmette (cioè sempre 1). Lo stesso discorso vale per gli input: il valore che emettono è ciò che propagano. Questa è una osservazione ovvia: gli input ed i bias non sono certo neuroni (non dispongono di alcuna funzione di attivazione né di pesi).

Nell'esempio sopra, infatti, $y_{h_0}(1)$ era pari ad 0.1 poiché h_0 è un bias ed il peso fra o e h_0 è pari a 0.1.

Calcoliamo dunque i nuovi valori dei pesi (Equazione 4.8):

$$w_{o,h_0}(2) = w_{o,h_0}(1) + \Delta w_{o,h_0}(1) = 0.1 + 0.110 = 0.210$$

$$w_{o,h_1}(2) = w_{o,h_1}(1) + \Delta w_{o,h_1}(1) = 0.2 + 0.047 = 0.247$$

Modifica dei pesi per il neurone hidden h_1

Ripetiamo ora quanto appena fatto con o , ma questa volta usiamo chiaramente la formula adeguata per calcolare δ_{h_1} (Equazione 4.11):

$$\delta_{h_1}(1) = y_{h_1}(1) \cdot (1 - y_{h_1}(1)) \cdot \sum_{k \in \text{Output}} \delta_k w_{k,h_1}(1)$$

In questo caso siamo fortunati, poiché la sommatoria sui k neuroni di output si limita ad o , l'unico che abbiamo. La formula diventa quindi:

$$\delta_{h_1}(1) = y_{h_1}(1) \cdot (1 - y_{h_1}(1)) \cdot \delta_o w_{o,h_1}(1) = 0.426 \cdot (1 - 0.426) \cdot 0.495 \cdot 0.2 = 0.254$$

Si noti che abbiamo usato il δ_o calcolato poco fa: *l'errore si propaga all'interno!*

Usiamo ora il gradiente locale appena calcolato per trovare gli incrementi:

$$\Delta w_{h_1,i_0}(1) = \eta \cdot \delta_{h_1}(1) \cdot y_{i_0}(1) = 1 \cdot 0.254 \cdot 1 = 0.254$$

$$\Delta w_{h_1,i_1}(1) = \eta \cdot \delta_{h_1}(1) \cdot y_{i_1}(1) = 1 \cdot 0.254 \cdot 0 = 0$$

I valori di y_{i_0} e y_{i_1} sono ottenuti, come detto in precedenza, considerando semplicemente il valore emesso dall'unità.

Aggiorniamo dunque i pesi:

$$\begin{aligned} w_{h_1, i_0}(2) &= w_{h_1, i_0}(1) + \Delta w_{h_1, i_0}(1) = -0.3 + 0.254 = -0.046 \\ w_{h_1, i_1}(2) &= w_{h_1, i_1}(1) + \Delta w_{h_1, i_1}(1) = -0.2 + 0 = -0.2 \end{aligned}$$

In Figura 4.5 vediamo la rete neurale aggiornata con i nuovi pesi.

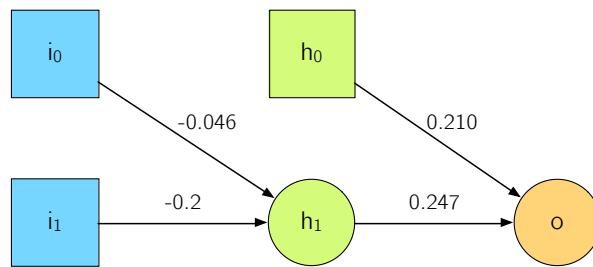


Figura 4.5: Situazione della rete neurale dopo una prima iterazione usando il primo pattern.

Si noti che non è stato effettuato alcun calcolo per i gradienti locali di i_0 , i_1 ed h_0 : ancora una volta ribadiamo che si tratta di nodi di input/bias e *non di neuroni*, dunque non ha alcun senso che abbiano un gradiente locale. Infatti, il gradiente locale indica come debba comportarsi il loro input al fine di diminuire l'errore... Ma questi elementi non hanno alcun input, piuttosto ne emettono uno (per altro sempre costante)!

4.3.5 Algoritmo di apprendimento

Andiamo dunque a vedere dove e come vengono applicate le formule che abbiamo ottenuto.

```

Dato un training set
Inizializza tutti i pesi casualmente.

while(il criterio di stop non è soddisfatto || il numero
      massimo di epoch non è raggiunto) {
    for each (esempio nel training set) {
        Esegui la rete e ottieni i valori in output
        Modifica i pesi per i neuroni di output
        Modifica i pesi per i neuroni hidden
    }
}
  
```

Listing 4.2: Pseudocodice per l'algoritmo di apprendimento delle reti a percettrone con delta rule.

Ogni esecuzione del ciclo while indica un'epoca (capiremo a breve cosa si intende per *criterio di stop*). I pesi vengono modificati esempio per esempio, dunque ogni pattern usa i pesi più recenti calcolati.

Esiste anche una versione *batch* dell'algoritmo in cui tutti i Δw calcolati (uno per ogni pattern) vengono sommati e l'aggiornamento dei pesi accade soltanto a fine epoca. Questa implementazione risulta particolarmente interessante per implementazioni parallele dell'algoritmo in cui la comunicazione dei vari Δw può essere complicata o costosa.

4.4 Garanzie teoriche

Nonostante l'algoritmo di back propagation sia estremamente utilizzato, non vi sono sostanzialmente garanzie teoriche sulle quali fare affidamento. L'algoritmo, infatti, trova soluzioni funzionanti (in un numero di epoche sufficientemente grande) ma non è detto che trovi il minimo globale. Osserviamo la Figura 4.6 per comprendere questo fenomeno.

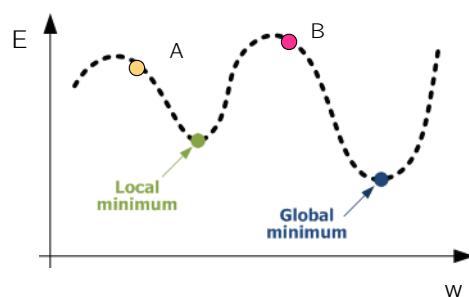


Figura 4.6: Grafico che rappresenta l'andamento dell'errore al variare dei pesi, con rappresentazione di un minimo locale e di un minimo locale.

Supponiamo di partire da una configurazione di pesi A : potremmo mano a mano avvicinarci al minimo locale senza così riuscire a superare la "collina" che ci separa dal minimo globale. Partendo da una configurazione di pesi B , invece, questa situazione potrebbe non verificarsi (dipende se scendiamo a destra o a sinistra).

4.4.1 Il learning rate

Anche la scelta del learning rate può agevolare o meno la stabilizzazione su un minimo locale a rischio di perdere quello globale. Non abbiamo purtroppo alcuna garanzia precisa: un po' di euristiche e tentativi possono portare alla definizione di un adeguato learning rate (che di solito varia fra 0.1 e 0.9). Si tenga però in considerazione che:

- Un learning rate piccolo porta a poche oscillazioni nel valore dei pesi, dunque sarà più facile avvicinarsi ad un minimo globale. Procedendo però in maniera più "sicura", l'algoritmo farà meno epoches per soddisfare il criterio imposto.
- Un learning rate grande permette di fare grandi oscillazioni e, sperabilmente, di superare eventuali "colline" e condurci al minimo globale. Grandi oscillazioni però potrebbero non farci raggiungere mai il minimo se la tolleranza impostata è molto fine.

Volendo è possibile utilizzare un *momentum* che modula ogni Δw verificando se è discorde rispetto all'iterazione precedente. Se lo è, il Δw sarà tappato al fine di evitare oscillazioni troppo ampie. Impiegando una tecnica come questa è possibile usare learning rate più alti con maggiore tranquillità.

4.4.2 Criteri di stop

Giacché l'algoritmo non converge, è necessario definire un criterio di stop per garantirne la terminazione. Vi sono diverse possibilità:

- Introdurre una tolleranza di errore: appena l' E calcolato è entro quella soglia, l'algoritmo si ferma.
- Se l'errore non decresce più al passare delle epoches l'algoritmo si ferma.
- La soluzione trovata è sufficientemente generale, ovvero si comporta bene anche su pattern non forniti in fase di apprendimento. Questo criterio è il più interessante poiché cerca anche di evitare l'overfitting della rete su determinati esempi.

4.4.3 Riassumendo

Facendo il punto della situazione, possiamo dire che i parametri fondamentali sui quali porre attenzione sono:

- La scelta del training set (per quanto riguarda le capacità di generalizzazione).
- La scelta iniziale dei pesi.
- Il numero di hidden units.
- Il learning rate.

4.5 I limiti delle reti back propagation

Le reti back propagation sono sicuramente le più utilizzate in questo momento, nonostante soffrano di tre limiti:

- L'algoritmo di apprendimento è lento (ci vogliono molte epochhe) ed è supervisionato.
- La rete può rimanere intrappolata nei minimi locali.
- La soluzione non è considerata psicologicamente valida (in natura non esiste alcun meccanismo di propagazione all'indietro dell'errore sulle sinapsi).

Il grande successo delle reti è dettato dal fatto che questi problemi siano in realtà futili o poco interessanti: per quanto concerne il primo punto l'algoritmo è stato migliorato e risulta ora più efficiente, il secondo punto non è ritenuto un problema grave poiché ci accontentiamo di una soluzione funzionante e non di quella ottima (questo inoltre ci aiuta ad evitare l'overfitting su un determinato training set) e il terzo punto è sostanzialmente un cruccio e viene ignorato da chiunque utilizzi le reti in senso applicativo.

4.6 Applicazioni delle reti back propagation

In questa sezione parleremo brevemente di alcune delle applicazioni delle reti neurali back propagation. Distingueremo applicazioni di tipo ingegneristico ed altre di tipo cognitivo, ponendo particolare attenzione sulle seconde.

In generale le reti neurali vengono impiegate laddove si necessiti di capacità di riconoscimento (facce, oggetti, testo, ecc.) o di categorizzazione (ad esempio frutti D.O.P.).

Due delle categorie di problemi degne di nota sono i problemi di *classificazione* ed i problemi di *regressione*:

- Classificazione: come sappiamo, questo tipo di problemi cercano di assegnare ad ogni input una determinata categoria. Si tratta pertanto di un approccio discreto, ovvero, fra una categoria e l'altra non c'è alcun passaggio intermedio.
- Regressione: i problemi di regressione forniscono un output diverso: a seconda di un determinato input, l'output fornisce un valore che non è però nel dominio discreto (una categoria, appunto) bensì nel dominio del continuo.

Tendenzialmente, i problemi di classificazione hanno tante unità di output quante sono le categorie in cui si vogliono situare gli input (e si attiva sempre una sola unità di output), mentre i problemi di regressione sono implementati in reti che mostrano una sola unità di output, la quale fornirà l'attivazione della rete per il determinato input mostrato alla rete.

Per i problemi di classificazione l'errore è misurato in termini di categorizzazione (se l'input è inserito nella categoria giusta o meno), mentre per i problemi di regressione l'errore si può stimare mediante errore quadratico medio, cioè come somma di tutti gli scarti calcolati come differenza fra valore ottenuto e valore atteso, al quadrato e sotto radice.

4.6.1 Applicazioni ingegneristiche

NETtalk

NETtalk è una applicazione nata negli anni '80 ed è una rete che impara a pronunciare l'inglese scritto. L'input preso è una stringa (un insieme di caratteri nel contesto) e l'output è il suono (codice fonetico) di quella determinata stringa. In Figura 4.7 vediamo la struttura della rete.

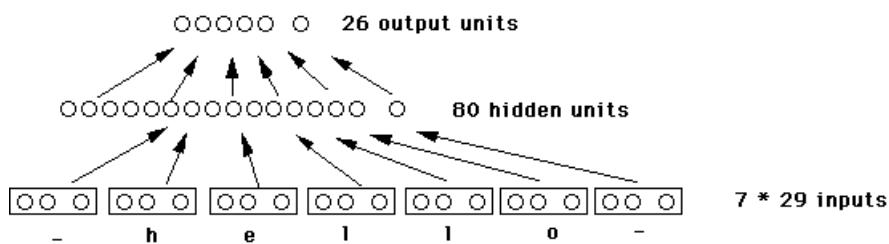


Figura 4.7: Schema che riassume la rete neurale adottata per NETtalk.

OCR

I sistemi di riconoscimento ottico dei caratteri, detti anche OCR (dall'inglese optical character recognition) sono programmi dedicati alla conversione di un'immagine contenente testo, solitamente acquisite tramite scanner, in testo digitale modificabile con un normale editor.

Durante la fase di addestramento vengono forniti alla rete degli esempi di immagini col corrispondente testo in formato ASCII o simile in modo che gli algoritmi si possano calibrare sul testo che usualmente andranno ad analizzare. Gli ultimi software di OCR utilizzano algoritmi in grado di riconoscere i contorni e in grado di ricostruire oltre al testo anche la formattazione della pagina.

Gli OCR sono utilizzati, ad esempio, nei sistemi postali (Figura 4.8).

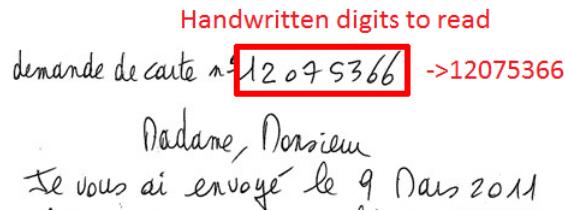


Figura 4.8: Esempio di numeri scritti a mano da essere interpretati da un OCR.

ALVINN

ALVINN è un acronimo che sta per Autonomous Land Vehicle In a Neural Network: si tratta di un sistema percettivo che sostanzialmente impara a guidare un veicolo osservando la guida di un utente umano.

È uno dei primi esempi che hanno dimostrato che le reti neurali possono essere utili per fare qualcosa di utile, infatti, sfruttando ALVINN si è riuscito a portare un veicolo senza autista ai 110 km/h in autostrada.

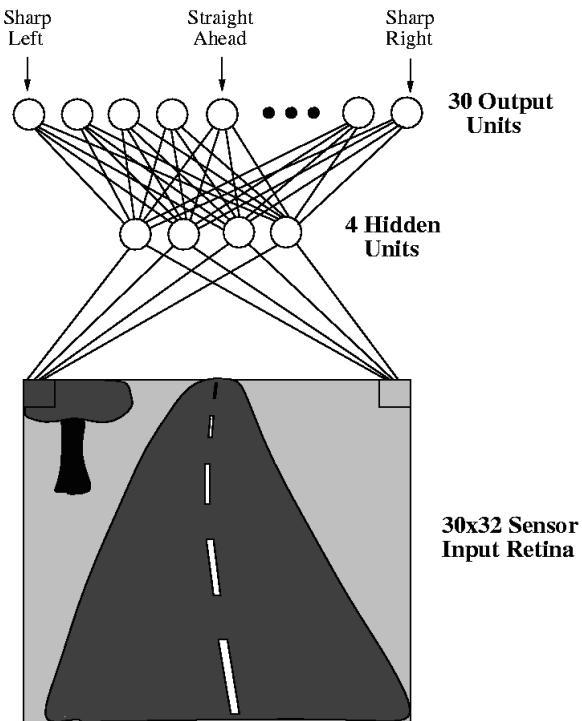


Figura 4.9: Il livello di input è costituito da una rete bidimensionale 30x32 che riceve dati da una videocamera installata sul veicolo. Ogni input è connesso alle quattro unità del livello hidden che a loro volta sono connesse a 30 unità di output, le quali forniscono una rappresentazione lineare della direzione in cui il veicolo dovrebbe transitare in modo da mantenerlo in corsia.

La rete è stata quindi addestrata fornendo una serie di fotografie e azioni da

compiere correlate.

4.6.2 Applicazioni in scienze cognitive

Le reti neurali vengono adoperate nelle scienze cognitive fondamentalmente per due motivi:

1. Le scienze cognitive sono gremite di modelli che studiano, teorizzano e dimostrano come funziona l'apprendimento di determinate nozioni/informazioni. Spesso, però, queste informazioni risultano "dogmatiche" e difficili da confutare. Mediante lo strumento delle reti neurali si cerca quindi di trovare spiegazioni differenti o alternative a determinati fenomeni cognitivi.
2. Le reti neurali possono essere utili per concretizzare fenomeni altrimenti difficili da comprendere.

Linguistic learning

L'ambito del *linguistic learning* è stato trattato mediante l'uso di reti neurali al fine di smentire la teoria classica secondo quale l'apprendimento di una lingua si costituisce in due separati processi cognitivi (Figura 4.10).

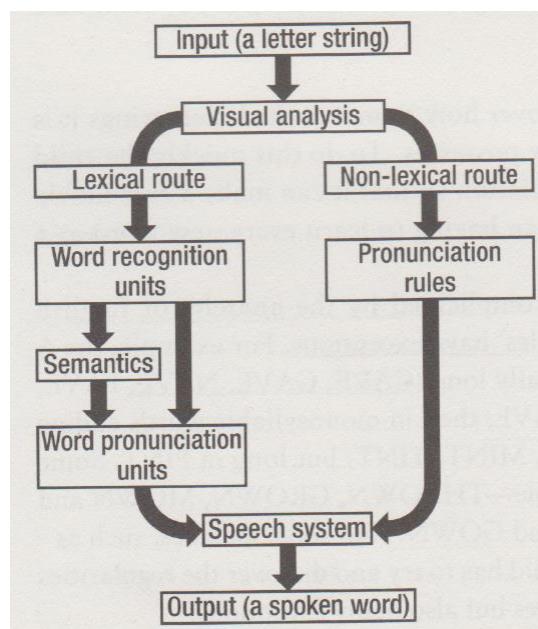


Figura 4.10: Vi sono due percorsi cognitivi diversi: uno sfrutta le regole lessicali (per le irregolarità) e l'altro invece applica semplicemente la regola di pronuncia.

In sostanza si hanno due diversi processi cognitivi e quando si legge una parola

se ne adopera uno soltanto. Se la parola è regolare (ad esempio *integer*), ovvero la pronuncia segue le regole della lingua (precedentemente apprese) allora si seguirà il percorso relativo all'applicazione della determinata regola (non lessicale). Qualora la parola letta sia irregolare (*pint*, che non si legge "pint" ma "paint"), invece, si ricorrerà ad un processo lessicale.

Tale teoria è supportata anche da studi neuropsicologici per cui si sa che alcune malattine possono inibire uno dei due processi: la dislessia fonologica inibisce quello relativo alle regole, mentre la dislessia di superficie inibisce il processo lessicale (e quindi la compressione di parole irregolari).

Mediante le reti neurali si propone un approccio unico di tipo *statistico* che accorda i due processi cognitivi in uno soltanto. Gli schemi in Figura 4.11 mostrano i risultati della ricerca. Il tipo di apprendimento ottenuto è molto

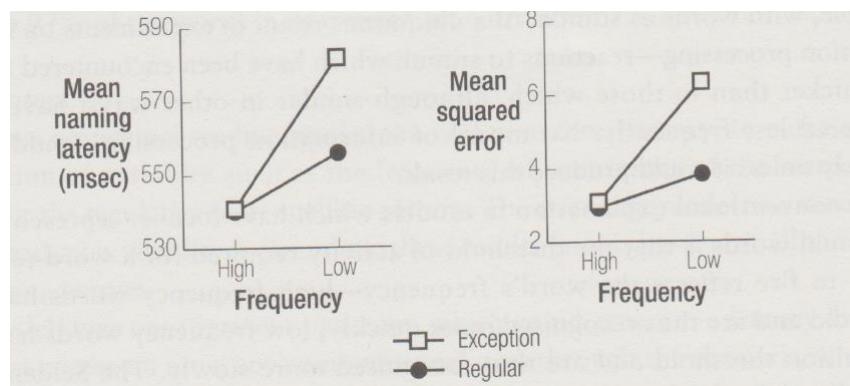


Figura 4.11: Il modello pronuncia bene sia le parole regolari che quelle irregolari. Più una parola è frequente e minore sarà il tempo necessario al modello per restituire un risultato.

simile a quello che hanno i bambini: all'interno di unici pesi vengono definiti entrambi i comportamenti (lessicale e non).

Vocabulary sprout

I bambini, intorno ai 20 mesi, iniziano a padroneggiare l'uso di svariate parole (nel senso di collegare ad un oggetto il suo corretto nome) in maniera rapida ed improvvisa. Questo fenomeno, chiamato *vocabulary sprout* è spiegato dalla teoria tradizionale mediante lo sviluppo di un nuovo meccanismo cognitivo nell'infante, che è quindi in grado di padroneggiare più parole in maniera immediata.

Tramite le reti neurali si vuole fornire una spiegazione alternativa, e cioè che il bambino accumuli, intorno ai 20 mesi, una massa critica di nomi ed informazioni tali per cui inizi a padroneggiarle con più facilità.

L'addestramento. La rete (Figura 4.12) viene addestrata in maniera particolare.

Prima di tutto sono stati definiti 32 prototipi, ovvero 32 immagini col concetto generale di ciò che si vuole apprendere. Ad esempio, possiamo immaginare la foto di un cane di taglia media e pelo medio. Ad ogni prototipo è associata una parola (nel nostro esempio, cane).

Di ogni prototipo sono poi stati fornite 6 "distorsioni", ovvero versioni leggermente diverse dal prototipo (nel nostro esempio: un alano, un labrador, un pitbull, ecc.). La rete viene addestrata usando *soltanente* le distorsioni (che in totale sono 192) e mai i prototipi.

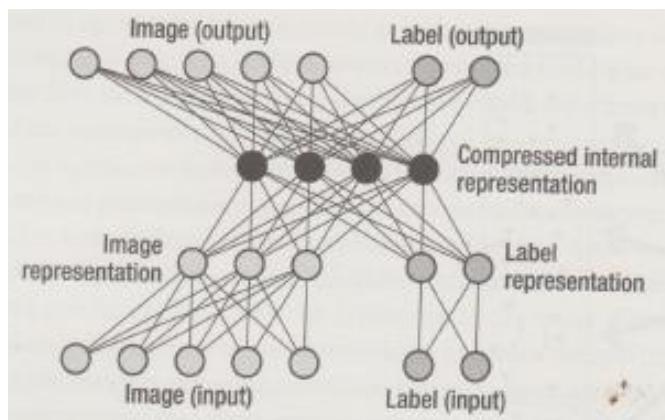


Figura 4.12: La rete è suddivisa in due parti collegate fra loro. Una riconosce le immagini e l'altra riconosce le parole.

L'apprendimento si sviluppa in tre fasi:

1. Si fornisce un'immagine alla rete e ci si aspetta che l'output sia quell'immagine.
2. Si fornisce una parola alla rete e ci si aspetta che l'output sia quella parola.
3. Si fornisce un'immagine con la sua parola e ci si aspetta che l'output sia quell'immagine con quella parola.

Appena ogni coppia viene correttamente riconosciuta (l'output è quindi uguale all'input), la rete è pronta ad essere interrogata in due sensi:

- *Comprensione*: data un'immagine si vuole ottenere la parola associata.
- *Produzione*: data una parola si vuole ottenere la corretta immagine.

La rete è anche testata fornendo i prototipi per vedere se è sufficientemente generale.

I risultati. Ciò che ha prodotto la rete è stato valutato in termini di:

- Comprensione.
- Produzione.
- Overextension: una parola è usata per definire un concetto che non vi riguarda.
- Underextension: una parola non viene usata per definire il suo concetto associato.
- Overfitting: una parola è usata correttamente per definire i prototipi.

I risultati sono impressionanti: le reti si comportano come i bambini in tutto e per tutto! Fanno gli stessi errori, hanno le stesse curve di apprendimento e hanno gli stessi risultati (Figura 4.13).

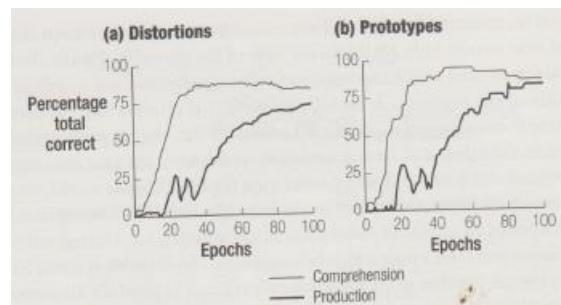


Figura 4.13: Comportamento della rete: è del tutto simile a quanto studiato sui bambini.

Questa ricerca ha anche permesso di scoprire che i bambini non soltanto hanno una esplosione in termini di produzione, bensì anche di comprensione (fatto che era precedentemente sconosciuto).

Capitolo 5

Self-Organizing Maps

5.1 Introduzione

Le *Self-Organizing Maps* (d'ora in poi *SOM*) sfruttano in maniera del tutto diversa i neuroni che abbiamo studiato. Si tratta infatti di reti *non supervisionate*, a cui non si deve fornire alcun valore di output desiderato.

I neuroni all'interno della SOM sono organizzati in una struttura reticolare bidimensionale e sono collegati fra loro da relazioni di adiacenza. A fronte di un input, la SOM ne cercherà le regolarità e lo mapperà in una determinata zona della mappa. L'obiettivo finale è quello di mappare input simili su neuroni vicini all'interno della struttura. C'è quindi una sorta di *competizione* fra neuroni.

In Figura 5.1 vediamo una semplice SOM.

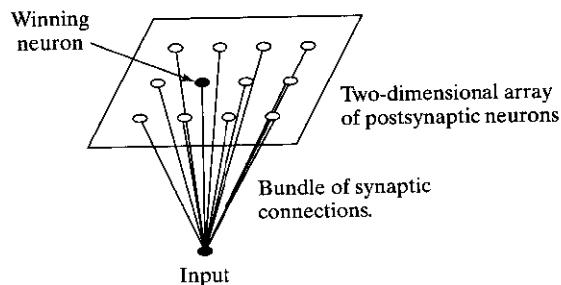


Figura 5.1: L'input viene diffuso su ogni neurone: quello che avrà output maggiore viene definito *best matching unit* (BMU).

L'output di ogni neurone viene calcolato come abbiamo già studiato per le reti a percettrone ma *non c'è bias* e *non c'è funzione di attivazione*.

Le SOM sono viste come alternative psicologicamente plausibili al cervello: l'organizzazione a zone che implementano e il fatto che siano non supervisio-

nate avvicinano di molto il suo funzionamento a quello di un cervello vero e proprio.

Un esempio introduttivo

Consideriamo la SOM in Figura 5.2.

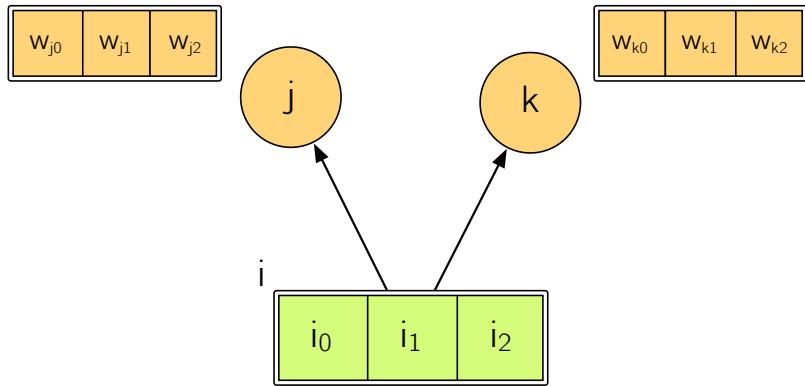


Figura 5.2: Esempio di SOM. j e k sono due neuroni mentre i è l'input, costituito in questo caso da tre valori.

Supponendo di avere come input $i = [1, 0, 2]$ e come pesi:

- Per j : $w_{j0} = 0.1$, $w_{j1} = 0.3$, $w_{j2} = 0.4$.
- Per k : $w_{k0} = 0.3$, $w_{k1} = 0.45$, $w_{k2} = 0.1$.

Allora avremmo come output:

- Per j : $i_0 \cdot w_{j0} + i_1 \cdot w_{j1} + i_2 \cdot w_{j2} = 0.9$.
- Per k : $i_0 \cdot w_{k0} + i_1 \cdot w_{k1} + i_2 \cdot w_{k2} = 0.5$.

La BMU sarebbe dunque j .

Si noti che al termine dell'algoritmo ogni BMU sarà non solo utile per categorizzare input comuni in zone comuni della rete, ma fungerà anche da *prototipo*. Infatti, il vettore di pesi associato ad una BMU sarà una sorta di “media” fra gli input che quella BMU è in grado di riconoscere.

5.2 Algoritmo di apprendimento e modifica dei pesi

Entriamo ora nel dettaglio di questi sistemi andando a capire come funziona l'apprendimento, come è usata la notazione di distanza e soprattutto come si modificano i pesi.

Possiamo riassumere l'algoritmo di apprendimento in quattro fasi:

1. Inizializzazione dei pesi in maniera casuale.
2. Competizione fra neuroni: la BMU viene trovata per un determinato input $x(i)$.
3. Modifica dei pesi della BMU in modo che nel futuro risponda nuovamente
4. Modifica dei pesi di tutti i nodi della rete, modulata dalla distanza fra il neurone e la BMU.

Chiaramente i passi 2-4 vengono ripetuti per diversi input (parleremo in seguito delle condizioni di terminazione).

5.2.1 Calcolare l'attivazione e trovare la BMU

Abbiamo già imparato come si calcola l'attivazione di un neurone j : dato infatti il vettore di input $x = [x_1, \dots, x_m]$ e il vettore dei pesi associato al neurone j $w_j = [w_{j1} \dots w_{jm}]$, possiamo calcolare l'attivazione con il semplice calcolo:

$$w_j^T x = w_{j1}x(1) + \dots + w_{jn}x(n)$$

Tale calcolo viene però implementato in maniera diversa ma (quasi) uguale. Più precisamente si considera BMU il neurone che ha distanza euclidea¹ fra l'input ed il suo vettore dei pesi minore.

In formule, i è la BMU se:

$$\forall j : \|x - w_i\| \leq \|x - w_j\| \quad (5.1)$$

Si noti che la definizione che avevamo dato noi (massima attivazione) è equivalente a quella appena descritta qualora tutti i vettori abbiano la stessa norma.

5.2.2 Modificare i pesi

Giacché calcoliamo la BMU basandoci sulla distanza euclidea riportata in Equazione 5.1, andremo a modificare i pesi cercando di ridurre quella distanza. Iniziamo studiando il caso della modifica dei pesi della BMU, dopodiché troveremo una formula generale per tutti i neuroni (che coinvolgerà quindi le distanze).

Modificare i pesi della BMU

Supponendo che j sia la BMU, modificheremo i pesi seguendo questa formula:

$$w_i(n+1) = w_i(n) + \eta(n)(x(n) - w_i(n))$$

¹Formula per la distanza euclidea: $\Delta(A, B) = \sqrt{(a_1 - b_1)^2 + \dots + (a_n - b_n)^2}$

Ha fatto la comparsa il nostro amico η , il learning rate. La formula è incredibilmente semplice: i pesi per l'iterazione $n + 1$ vengono calcolati aggiungendo la distanza fra l'input ed i pesi stessi all'iterazione n , il tutto modulato dal learning rate.

Il learning rate svolge una funzione particolarmente importante: se fosse 1, infatti, andremo fondamentalmente a sovrascrivere il vettore dei pesi con l'input stesso, assicurandoci quindi che all'iterazione successiva lo stesso input venga assegnato al neurone designato come BMU, ma chiaramente peccheremmo di overfitting rispetto all'input. Per queste ragioni, il learning rate è tipicamente assegnato fra 0.1 e 0.01 (ne parleremo meglio in seguito). Sempre per la stessa ragione, si tende a far decrementare di iterazione in iterazione, seguendo la formula:

$$\eta(n) = \eta_0 \cdot \exp\left(-\frac{n}{\tau}\right)$$

dove τ è una certa costante ed η_0 è il learning rate iniziale.

In seguito alla correzione, comunque, avremo che w_i sarà certamente più simile all'input x considerato e qualora $x(n+1)$ dovesse essere simile a $x(n)$, i sarà più probabilmente in grado di riconoscerlo.

Modificare i pesi: regola generale

Introduciamo ora un fattore nella nostra formula.

Consideriamo il concetto di *neighborhood*: l'insieme dei neuroni "vicini" al neurone BMU. Avendo due neuroni i e j possiamo calcolare il grado di neighborhood mediante la funzione:

$$h_{ji}(n) = \exp\left(-\frac{d_{ji}^2}{2\sigma(n)^2}\right)$$

dove d_{ji} è la distanza calcolata mediante coordinate fra i e j e σ determina l'ampiezza di una funzione gaussiana, come riportato Figura 5.3.

Il valore di σ , come vediamo, è funzione dell'iterazione: mano a mano che le epoche trascorrono, la curva gaussiana è sempre più fine e quindi sempre meno neuroni vengono considerati vicini della BMU.

In pratica stiamo dicendo che più il neurone j è vicino a i , più il valore di h_{ji} sarà maggiore (fino ad arrivare ad 1 qualora j sia proprio i), il tutto è però reso più "smooth" dalla funzione gaussiana. Inoltre, col passare delle epoche, la funzione gaussiana riduce la sua ampiezza rendendo mano sempre più piccolo h_{ji} per i neuroni più distanti da i (su i rimane sempre 1). Più precisamente, la funzione gaussiana diminuisce seguendo la formula:

$$\sigma(n) = \sigma \cdot \exp\left(-\frac{n}{\tau}\right)$$

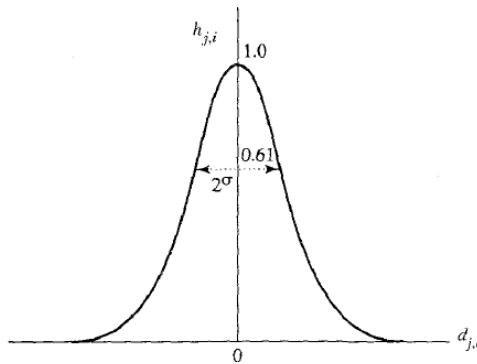


Figura 5.3: Funzione gaussiana che modula l'area di neighborhood.

dove τ è una certa costante (diversa dalla τ del learning rate) ed σ_0 è il valore iniziale di sigma.

Siamo ora in grado di modificare la nostra formula introducendo h (si supponga che i sia la BMU):

$$w_j(n+1) = w_j(n) + \eta(n) \cdot h_{ji}(n) \cdot (x(n) - w_j(n)) \quad (5.2)$$

Quando $j = i$, il termine h_{ji} varrà 1 e quindi avremo ottenuto la formula di prima. Se invece j non è i , la modifica dei pesi verrà modulata dalla vicinanza di j rispetto a i .

5.2.3 L'algoritmo

L'apprendimento si suddivide in due fasi (costituite a loro volta dalle quattro fasi di cui abbiamo accennato a inizio sezione):

1. Auto-organizzazione: si effettuano le prime mille epoche (circa), durante le quali il learning rate diminuisce (partendo da 0.1 arriva fino a 0.01) e la funzione di neighborhood si stringe fino ad includere praticamente solo più la BMU.
2. Convergenza: vengono effettuate ancora cinquecento epoche al fine di rendere i pesi più precisi. Il learning rate scende ulteriormente e solo più la BMU risente della modifica dei pesi.

Si noti che né il learning rate né la funzione di neighborhood raggiungono mai lo zero ma vi tendono. Questo significa che la modifica dei pesi riguarderà sempre tutti i nodi della rete, ma sui più distanti la variazione sarà irrilevante.

Come condizione di terminazione possiamo prendere la differenza fra i vettori peso (considerati neurone per neurone) di due iterazioni: quando non c'è più una variazione significativa, l'algoritmo termina.

5.3 Un esempio cumulativo

Andiamo ora ad applicare quanto appreso in un esempio. Onde evitare calcoli troppo astrusi, applicheremo alcune semplificazioni poco realistiche:

1. Molti dei neuroni partiranno con i pesi impostati a zero: questo non accade mai, i pesi sono inizializzati a piccoli valori randomici maggiori di zero.
2. Il learning rate è 1 e non cambia col procedere con le iterazioni: solitamente varia fra 0.1 e 0.01.
3. La funzione di neighborhood è discretizzata e non cambia col procedere delle iterazioni.

Più precisamente abbiamo che:

$$h_{ij} = \begin{cases} 1, & \text{se } d_{ij} = 0 \\ 0.5, & \text{se } d_{ij} = 1 \\ 0, & \text{altrimenti} \end{cases}$$

In pratica avremo che la funzione di neighborhood varrà 1 se il neurone considerato è la BMU, 0.5 se si trova a distanza 1 (la distanza è calcolata mediante coordinate, quindi solo i vicini diretti sono a distanza 1) e 0 altrimenti.

Utilizzeremo la rete in Figura 5.4.

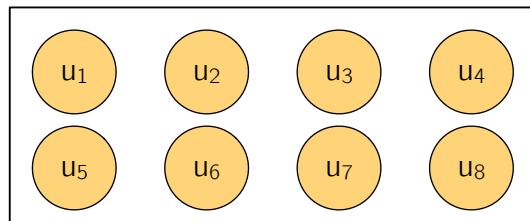


Figura 5.4

Gli input (di due elementi) sono i seguenti:

$$\begin{aligned} x(1) &= [0.1 \quad 0.9] \\ x(2) &= [0.2 \quad 0.8] \\ x(3) &= [0.9 \quad 0.1] \\ x(4) &= [0.8 \quad 0.2] \end{aligned}$$

Possiamo vedere a occhio come $x(1)$ ed $x(2)$ siano simili fra loro e molto dissimili da $x(3)$ ed $x(4)$ (che a loro volta si assomigliano). Ci aspettiamo

quindi una categorizzazione che suddivide i primi due input dai secondi due.
Ecco invece il vettore dei pesi di ogni neurone:

$$\begin{aligned} w_{u1} &= [0 \quad 0.52] \\ w_{u2} &= [0.5 \quad 0.1] \\ w_{u3} &= [0.1 \quad 0.5] \\ w_{u4} &= [0 \quad 0] \\ w_{u5} &= [0 \quad 0] \\ w_{u6} &= [0 \quad 0] \\ w_{u7} &= [0 \quad 0] \\ w_{u8} &= [0.52 \quad 0] \end{aligned}$$

Stato iniziale

Al fine di verificare l'effettivo funzionamento dell'algoritmo (che ovviamente funzionerà) testiamo il risultato attuale sulla rete per i nostri quattro input. Riportiamo i valori delle distanze euclidi fra input e vettori dei pesi per ogni input e per ogni neurone.

Per $x(1)$:

$$\begin{aligned} \Delta(x(1), w_{u1}) &= \mathbf{0.392} \\ \Delta(x(1), w_{u2}) &= 0.894 \\ \Delta(x(1), w_{u3}) &= 0.400 \\ \Delta(x(1), w_{u4}) &= 0.905 \\ \Delta(x(1), w_{u5}) &= 0.905 \\ \Delta(x(1), w_{u6}) &= 0.905 \\ \Delta(x(1), w_{u7}) &= 0.905 \\ \Delta(x(1), w_{u8}) &= 0.993 \end{aligned}$$

Per l'input $x(1)$ la BMU risulta quindi essere u_1 (ha minima distanza euclidea).

Per $x(2)$:

$$\begin{aligned} \Delta(x(2), w_{u1}) &= 0.344 \\ \Delta(x(2), w_{u2}) &= 0.761 \\ \Delta(x(2), w_{u3}) &= \mathbf{0.316} \\ \Delta(x(2), w_{u4}) &= 0.824 \\ \Delta(x(2), w_{u5}) &= 0.824 \\ \Delta(x(2), w_{u6}) &= 0.824 \\ \Delta(x(2), w_{u7}) &= 0.824 \\ \Delta(x(2), w_{u8}) &= 0.861 \end{aligned}$$

Per l'input $x(2)$ la BMU risulta quindi essere u_3 .

Per $x(3)$:

$$\begin{aligned}\Delta(x(3), w_{u1}) &= 0.993 \\ \Delta(x(3), w_{u2}) &= 0.400 \\ \Delta(x(3), w_{u3}) &= 0.894 \\ \Delta(x(3), w_{u4}) &= 0.905 \\ \Delta(x(3), w_{u5}) &= 0.905 \\ \Delta(x(3), w_{u6}) &= 0.905 \\ \Delta(x(3), w_{u7}) &= 0.905 \\ \Delta(x(3), w_{u8}) &= \mathbf{0.392}\end{aligned}$$

Per l'input $x(3)$ la BMU risulta quindi essere u_8 .

Per $x(4)$:

$$\begin{aligned}\Delta(x(4), w_{u1}) &= 0.861 \\ \Delta(x(4), w_{u2}) &= \mathbf{0.316} \\ \Delta(x(4), w_{u3}) &= 0.761 \\ \Delta(x(4), w_{u4}) &= 0.824 \\ \Delta(x(4), w_{u5}) &= 0.824 \\ \Delta(x(4), w_{u6}) &= 0.824 \\ \Delta(x(4), w_{u7}) &= 0.824 \\ \Delta(x(4), w_{u8}) &= 0.344\end{aligned}$$

Per l'input $x(4)$ la BMU risulta quindi essere u_2 .

In Figura 5.5 riassumiamo lo stato iniziale della rete, indicando chi sono le varie BMU per i diversi input. Abbiamo anche indicato con due colori diverse le categorie (presunte) degli input.

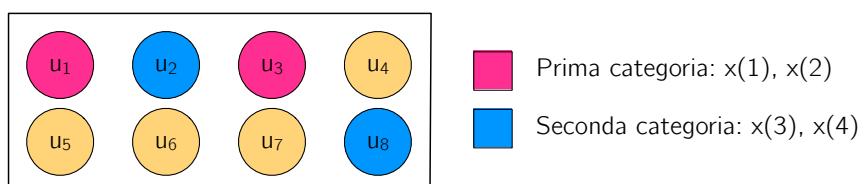


Figura 5.5: Sono evidenziate le BMU per i vari input allo stato iniziale della rete. Per ora la categorizzazione è del tutto errata: input diversi fra loro portano a BMU molto vicini.

5.3.1 Esecuzione dell'algoritmo

Andiamo ora ad eseguire l'algoritmo. Per comodità saranno omessi tutti i calcoli esplicativi delle distanze euclideanee.

Prima iterazione.

Consideriamo il primo input, $x(1)$. Troviamo la BMU per questo input (in questo caso abbiamo appena effettuato i calcoli):

$$\begin{aligned}\Delta(x(1), w_{u1}(1)) &= \mathbf{0.392} \\ \Delta(x(1), w_{u2}(1)) &= 0.894 \\ \Delta(x(1), w_{u3}(1)) &= 0.400 \\ \Delta(x(1), w_{u4}(1)) &= 0.905 \\ \Delta(x(1), w_{u5}(1)) &= 0.905 \\ \Delta(x(1), w_{u6}(1)) &= 0.905 \\ \Delta(x(1), w_{u7}(1)) &= 0.905 \\ \Delta(x(1), w_{u8}(1)) &= 0.993\end{aligned}$$

Come detto prima, la BMU è u_1 .

Modificare i pesi della BMU. Per prima cosa modifichiamo i pesi di u_1 , usando la formula generica in Equazione 5.2:

$$\begin{aligned}w_{u1}(2) &= w_{u1}(1) + \eta(1) \cdot h_{u1,u1}(1) \cdot (x(1) - w_{u1}(1)) \\ &= [0 \ 0.52] + 1 \cdot 1 \cdot \left([0.1 \ 0.9] - [0 \ 0.52] \right) \\ &= [0 \ 0.52] + [0.1 \ 0.38] \\ &= [0.1 \ 0.9]\end{aligned}$$

I pesi sono chiaramente diventati pari all'input a causa del learning rate pari ad 1.

Modificare i pesi delle altre unità. Dobbiamo ora ricalcolare tutti i pesi, sempre usando la formula generica in Equazione 5.2 ma il valore della funzione di neighborhood cambierà a seconda della distanza del nodo considerato rispetto alla BMU (u_1 , in questo caso).

Notiamo però gli unici nodi per cui h è diverso da zero sono u_2 e u_5 , ovvero l'unico nodo che dista al massimo 1 da u_1 . Trascuriamo perciò tutte le modifiche dei pesi per gli altri nodi, poiché l'incremento sarebbe pari a 0 (dato che h sarebbe 0).

$$\begin{aligned}w_{u2}(2) &= w_{u2}(1) + \eta(1) \cdot h_{u2,u1}(1) \cdot (x(1) - w_{u2}(1)) \\ &= [0.5 \ 0.1] + 1 \cdot 0.5 \cdot \left([0.1 \ 0.9] - [0.5 \ 0.1] \right) \\ &= [0.5 \ 0.1] + [-0.2 \ 0.4] \\ &= [0.3 \ 0.5]\end{aligned}$$

$$\begin{aligned}
 w_{u5}(2) &= w_{u5}(1) + \eta(1) \cdot h_{u5,u1}(1) \cdot (x(1) - w_{u5}(1)) \\
 &= [0 \ 0] + 1 \cdot 0.5 \cdot \left([0.1 \ 0.9] - [0 \ 0] \right) \\
 &= [0 \ 0] + [0.05 \ 0.45] \\
 &= [0.05 \ 0.45]
 \end{aligned}$$

Resoconto dei pesi a termine iterazione. Ecco quindi i nostri nuovi pesi (con \star sono indicati quelli che sono stati effettivamente modificati):

$$\begin{aligned}
 w_{u1}(2) &= [0.1 \ 0.9] (\star) \\
 w_{u2}(2) &= [0.3 \ 0.5] (\star) \\
 w_{u3}(2) &= [0.1 \ 0.5] \\
 w_{u4}(2) &= [0 \ 0] \\
 w_{u5}(2) &= [0.05 \ 0.45] (\star) \\
 w_{u6}(2) &= [0 \ 0] \\
 w_{u7}(2) &= [0 \ 0] \\
 w_{u8}(2) &= [0.52 \ 0]
 \end{aligned}$$

Seconda iterazione.

Consideriamo il secondo input, $x(2)$. Troviamo la BMU (minima distanza euclidea) per questo input (dobbiamo rifare i calcoli di prima, alcuni pesi sono cambiati):

$$\begin{aligned}
 \Delta(x(2), w_{u1}(2)) &= 0.141 \\
 \Delta(x(2), w_{u2}(2)) &= 0.316 \\
 \Delta(x(2), w_{u3}(2)) &= \mathbf{0.316} \\
 \Delta(x(2), w_{u4}(2)) &= 0.824 \\
 \Delta(x(2), w_{u5}(2)) &= 0.380 \\
 \Delta(x(2), w_{u6}(2)) &= 0.824 \\
 \Delta(x(2), w_{u7}(2)) &= 0.824 \\
 \Delta(x(2), w_{u8}(2)) &= 0.861
 \end{aligned}$$

In questo caso avremmo due BMU possibili: scegliamo a caso u_2 .

Modificare i pesi della BMU. Per prima cosa modifichiamo i pesi di u_2 , usando la solita formula in Equazione 5.2:

$$\begin{aligned} w_{u2}(3) &= w_{u2}(2) + \eta(2) \cdot h_{u2,u2}(2) \cdot (x(2) - w_{u2}(2)) \\ &= [0.3 \ 0.5] + 1 \cdot 1 \cdot ([0.2 \ 0.8] - [0.3 \ 0.5]) \\ &= [0.3 \ 0.5] + [-0.1 \ 0.3] \\ &= [0.2 \ 0.8] \end{aligned}$$

Come prima, i pesi sono diventati pari all'input a causa del learning rate pari ad 1.

Modificare i pesi delle altre unità. Come prima, per semplificare i calcoli andiamo a considerare gli unici nodi vicini a u_2 e che quindi hanno h diversa da 0. Trattasi di u_1 , u_3 ed u_6 .

$$\begin{aligned} w_{u1}(3) &= w_{u1}(2) + \eta(2) \cdot h_{u1,u2}(2) \cdot (x(2) - w_{u1}(2)) \\ &= [0.1 \ 0.9] + 1 \cdot 0.5 \cdot ([0.2 \ 0.8] - [0.1 \ 0.9]) \\ &= [0.1 \ 0.9] + [0.05 \ -0.05] \\ &= [0.15 \ 0.85] \end{aligned}$$

$$\begin{aligned} w_{u3}(3) &= w_{u3}(2) + \eta(2) \cdot h_{u3,u2}(2) \cdot (x(2) - w_{u3}(2)) \\ &= [0.1 \ 0.5] + 1 \cdot 0.5 \cdot ([0.2 \ 0.8] - [0.1 \ 0.5]) \\ &= [0.1 \ 0.5] + [0.05 \ 0.15] \\ &= [0.15 \ 0.65] \end{aligned}$$

$$\begin{aligned} w_{u6}(3) &= w_{u6}(2) + \eta(2) \cdot h_{u6,u2}(2) \cdot (x(2) - w_{u6}(2)) \\ &= [0 \ 0] + 1 \cdot 0.5 \cdot ([0.2 \ 0.8] - [0 \ 0]) \\ &= [0 \ 0] + [0.1 \ 0.4] \\ &= [0.1 \ 0.4] \end{aligned}$$

Resoconto dei pesi a termine iterazione. Ecco quindi i nostri nuovi pesi (con \star sono indicati quelli che sono stati effettivamente modificati):

$$\begin{aligned} w_{u1}(3) &= [0.15 \quad 0.85] (\star) \\ w_{u2}(3) &= [0.2 \quad 0.8] (\star) \\ w_{u3}(3) &= [0.15 \quad 0.65] (\star) \\ w_{u4}(3) &= [0 \quad 0] \\ w_{u5}(3) &= [0.05 \quad 0.45] \\ w_{u6}(3) &= [0.1 \quad 0.4] (\star) \\ w_{u7}(3) &= [0 \quad 0] \\ w_{u8}(3) &= [0.52 \quad 0] \end{aligned}$$

Terza iterazione.

Consideriamo il terzo input, $x(3)$. Troviamo la BMU (minima distanza euclidea) per questo input:

$$\begin{aligned} \Delta(x(3), w_{u1}(3)) &= 1.060 \\ \Delta(x(3), w_{u2}(3)) &= 0.989 \\ \Delta(x(3), w_{u3}(3)) &= 0.930 \\ \Delta(x(3), w_{u4}(3)) &= 0.905 \\ \Delta(x(3), w_{u5}(3)) &= 0.919 \\ \Delta(x(3), w_{u6}(3)) &= 0.854 \\ \Delta(x(3), w_{u7}(3)) &= 0.905 \\ \Delta(x(3), w_{u8}(3)) &= \mathbf{0.392} \end{aligned}$$

La BMU è u_8 .

NB: D'ora in poi si riporteranno solo alcuni passaggi dei calcoli per semplicità.

Modificare i pesi della BMU. Per prima cosa modifichiamo i pesi di u_8 , usando la solita formula in Equazione 5.2:

$$\begin{aligned} w_{u8}(4) &= w_{u8}(3) + \eta(3) \cdot h_{u8,u8}(3) \cdot (x(3) - w_{u8}(3)) \\ &= [0.52 \quad 0] + 1 \cdot 1 \cdot \left([0.9 \quad 0.1] - [0.52 \quad 0] \right) \\ &= [0.9 \quad 0.1] \end{aligned}$$

Come sempre, i pesi sono diventati pari all'input a causa del learning rate pari ad 1.

Modificare i pesi delle altre unità. Per comodità consideriamo solo i nodi vicini ad u_8 e che quindi hanno h diversa da 0. Trattasi di u_4 ed u_7 .

$$\begin{aligned} w_{u4}(4) &= w_{u4}(3) + \eta(3) \cdot h_{u4,u8}(3) \cdot (x(3) - w_{u4}(3)) \\ &= [0 \ 0] + 1 \cdot 0.5 \cdot ([0.9 \ 0.1] - [0 \ 0]) \\ &= [0.45 \ 0.05] \end{aligned}$$

$$\begin{aligned} w_{u7}(4) &= w_{u7}(3) + \eta(3) \cdot h_{u7,u8}(3) \cdot (x(3) - w_{u7}(3)) \\ &= [0 \ 0] + 1 \cdot 0.5 \cdot ([0.9 \ 0.1] - [0 \ 0]) \\ &= [0.45 \ 0.05] \end{aligned}$$

Resoconto dei pesi a termine iterazione. Ecco quindi i nostri nuovi pesi (con $*$ sono indicati quelli che sono stati effettivamente modificati):

$$\begin{aligned} w_{u1}(4) &= [0.15 \ 0.85] \\ w_{u2}(4) &= [0.2 \ 0.8] \\ w_{u3}(4) &= [0.15 \ 0.65] \\ w_{u4}(4) &= [0.45 \ 0.05] (*) \\ w_{u5}(4) &= [0.05 \ 0.45] \\ w_{u6}(4) &= [0.1 \ 0.4] \\ w_{u7}(4) &= [0.45 \ 0.05] (*) \\ w_{u8}(4) &= [0.9 \ 0.1] (*) \end{aligned}$$

Quarta iterazione.

Consideriamo il quarto input, $x(4)$. Troviamo la BMU (minima distanza euclidea) per questo input:

$$\begin{aligned} \Delta(x(4), w_{u1}(4)) &= 0.919 \\ \Delta(x(4), w_{u2}(4)) &= 0.848 \\ \Delta(x(4), w_{u3}(4)) &= 0.790 \\ \Delta(x(4), w_{u4}(4)) &= 0.380 \\ \Delta(x(4), w_{u5}(4)) &= 0.790 \\ \Delta(x(4), w_{u6}(4)) &= 0.728 \\ \Delta(x(4), w_{u7}(4)) &= 0.380 \\ \Delta(x(4), w_{u8}(4)) &= \mathbf{0.141} \end{aligned}$$

La BMU è u_8 .

Modificare i pesi della BMU. Per prima cosa modifichiamo i pesi di u_8 , usando la solita formula in Equazione 5.2:

$$\begin{aligned} w_{u8}(5) &= w_{u8}(4) + \eta(4) \cdot h_{u8,u8}(4) \cdot (x(3) - w_{u8}(4)) \\ &= [0.9 \ 0.1] + 1 \cdot 1 \cdot \left([0.8 \ 0.2] - [0.9 \ 0.1] \right) \\ &= [0.8 \ 0.2] \end{aligned}$$

Come sempre, i pesi sono diventati pari all'input a causa del learning rate pari ad 1.

Modificare i pesi delle altre unità. Per comodità consideriamo solo i nodi vicini ad u_8 e che quindi hanno h diversa da 0. Trattasi di u_4 ed u_7 .

$$\begin{aligned} w_{u4}(5) &= w_{u4}(4) + \eta(4) \cdot h_{u4,u8}(4) \cdot (x(4) - w_{u4}(4)) \\ &= [0.45 \ 0.05] + 1 \cdot 0.5 \cdot \left([0.8 \ 0.2] - [0.45 \ 0.05] \right) \\ &= [0.625 \ 0.125] \end{aligned}$$

$$\begin{aligned} w_{u7}(5) &= w_{u7}(4) + \eta(4) \cdot h_{u7,u8}(4) \cdot (x(4) - w_{u7}(4)) \\ &= [0.45 \ 0.05] + 1 \cdot 0.5 \cdot \left([0.8 \ 0.2] - [0.45 \ 0.05] \right) \\ &= [0.625 \ 0.125] \end{aligned}$$

Riporto dei pesi a termine iterazione. Ecco quindi i nostri nuovi pesi (con $*$ sono indicati quelli che sono stati effettivamente modificati):

$$\begin{aligned} w_{u1}(5) &= [0.15 \ 0.85] \\ w_{u2}(5) &= [0.2 \ 0.8] \\ w_{u3}(5) &= [0.15 \ 0.65] \\ w_{u4}(5) &= [0.625 \ 0.125] (*) \\ w_{u5}(5) &= [0.05 \ 0.45] \\ w_{u6}(5) &= [0.1 \ 0.4] \\ w_{u7}(5) &= [0.625 \ 0.125] (*) \\ w_{u8}(5) &= [0.8 \ 0.2] (*) \end{aligned}$$

5.3.2 Risultati dell'algoritmo dopo quattro iterazioni

L'algoritmo ha passato in rassegna i quattro output una sola volta: vediamo come sono stati modificati i BMU rispetto a prima

A inizio algoritmo i BMU erano:

- Per $x(1) \Rightarrow u_1$.
- Per $x(2) \Rightarrow u_3$.
- Per $x(3) \Rightarrow u_8$.
- Per $x(4) \Rightarrow u_2$.

Andando ora a calcolare per ogni input il BMU sfruttando i pesi ottenuti al termine della quarta iterazione otteniamo (per semplicità non riportiamo tutte le distanze euclidi, si invita il lettore a verificare i calcoli):

- Per $x(1) \Rightarrow u_1$.
- Per $x(2) \Rightarrow u_2$.
- Per $x(3) \Rightarrow u_8$.
- Per $x(4) \Rightarrow u_8$.

Per interpretare questi risultati, confrontiamo le due SOM (prima dell'algoritmo e dopo l'algoritmo) in Figura 5.6.

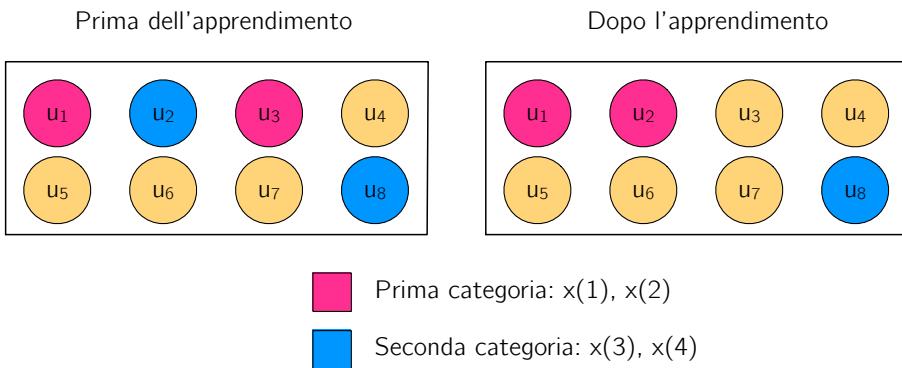


Figura 5.6: In soltanto quattro iterazioni la rete si è polarizzata molto. L'unità u_8 è BMU per entrambi gli input della seconda categoria, mentre u_1 ed u_2 sono i BMU per la prima (il che è legittimo, poiché sono unità fra loro molto vicine).

5.4 Applicazioni ed esempi conclusivi

Come già fatto per le reti back-propagation vediamo qualche applicazione delle SOM. Due esempi dell'efficacia dell'algoritmo ci sono dati dalla Figura 5.7 e dalla Figura 5.8. Al termine dell'algoritmo, ogni neurone ha un vettore peso che è in grado di riconoscere determinati input. Per fare ciò, il vettore dei pesi sarà simile all'input: è per questo che la disposizione finale dei pesi ricalca decisamente la distribuzione iniziale dell'input.

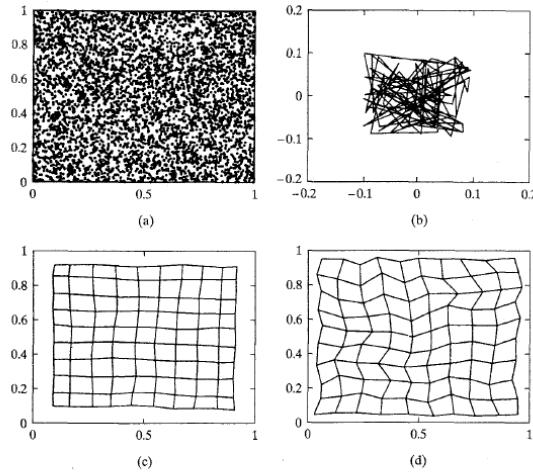


Figura 5.7: In (a) vediamo i dati in input, cioè che vorremmo che la rete riconoscesse. In (b) la distribuzione iniziale dei pesi (ad ogni neurone è associato il punto che è disegnato dal suo vettore dei pesi). In (c) la rappresentazione dei pesi in seguito alla fase di auto-organizzazione. In (d) la rappresentazione dei pesi in seguito alla fase di convergenza.

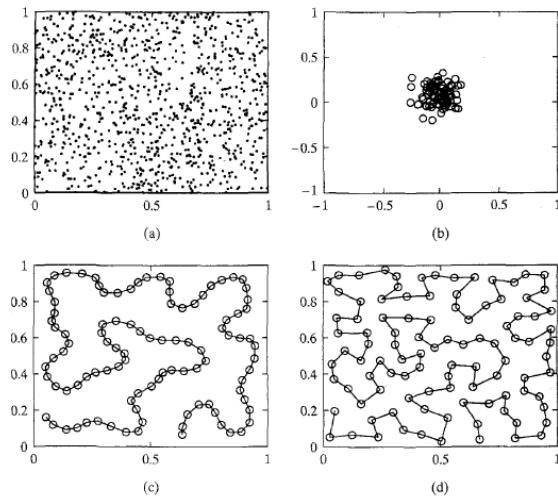


Figura 5.8: In (a) vediamo i dati in input, cioè che vorremmo che la rete riconoscesse. In (b) la distribuzione iniziale dei pesi (ad ogni neurone è associato il punto che è disegnato dal suo vettore dei pesi). In (c) la rappresentazione dei pesi in seguito alla fase di auto-organizzazione. In (d) la rappresentazione dei pesi in seguito alla fase di convergenza.

The Neural Phonetic Typewriter

La prima applicazione che vediamo delle SOM è stata inventata dallo stesso Kohonen e consta di una macchina che, preso in input un segnale acustico sono in grado di mapparlo in una determinata zona della mappa, alla quale corrisponde un preciso suono fonetico.

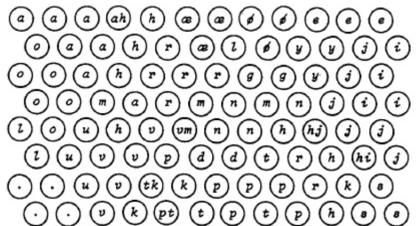


Figura 5.9: Ogni neurone è rappresentato da un cerchio, e al suo interno v'è il simbolo fonetico per cui è la BMU.

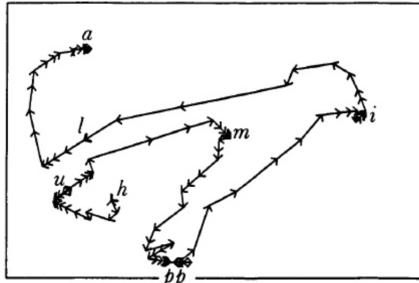


Figura 5.10: Sequenza delle risposte ottenute sulla mappa fonetica al pronunciare della parola *humppila*. Ogni freccia indica un ritardo di 10 millisecondi.

Categorizzazione negli infanti

In Figura 5.11 riprendiamo un disegno già introdotto a inizio di questo volume.

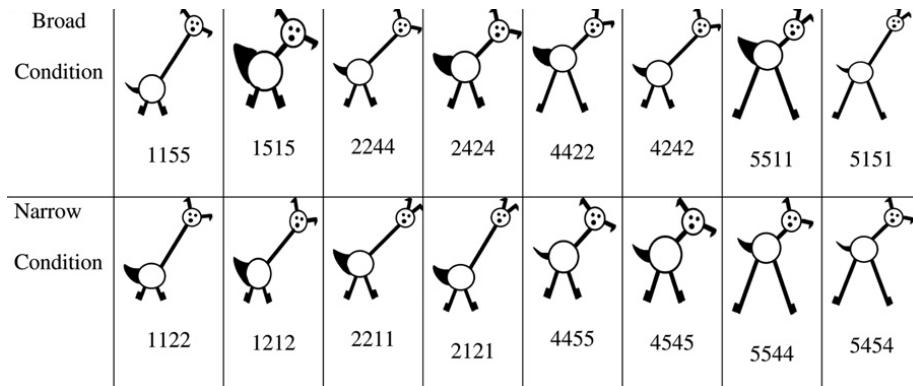


Figura 5.11: Due insiemi di immagini con cui allenare una rete o familiarizzare un infante.

Abbiamo ora tutti gli strumenti per comprendere appieno l'immagine. Broad e narrow sono due insiemi diversi di immagini: la prima contiene un insieme di pupazzi in cui le caratteristiche (collo, piedi, coda) sono distribuite in maniera randomica su ogni immagine, mentre il secondo insieme contiene due gruppi distinti di pupazzi (quelli con il collo lungo e quelli con le gambe lunghe). L'intento è, dati questi due diversi insiemi di input, verificare se le SOM categorizzassero le varie immagini secondo lo stesso principio di categorizzazione di un infante (si parla di bambini di 10 mesi).

La novelty preference procedure. Per capire se il comportamento della SOM e quello di un infante sono paragonabile, è ovviamente necessario capire come si misura la capacità di categorizzare di un infante. Per fare ciò, si utilizza

la *novelty preference procedure*, uno standard molto affermato che consta di due semplici fasi:

1. L'infante viene familiarizzato mediante l'esposizione a un determinato insieme di input.
2. Alcuni stimoli (a coppie) vengono proposti all'infante. Lo stimolo che viene fissato più a lungo è quello per cui c'è maggiore stupore, dunque, è quello che probabilmente non è stato interiorizzato dal bambino.

Nel nostro esempio l'infante è stato familiarizzato con l'insieme narrow oppure con l'insieme broad. Si sono poi adoperati due stimoli diverse, ovvero due coppie diverse di immagini: un prototipo equilibrato del pupazzo con un pupazzo dal collo molto lungo, e un prototipo equilibrato del pupazzo con un pupazzo dalle gambe lunghe. Tali stimoli sono stati mostrati all'infante indipendentemente dall'insieme col quale è stato effettuato l'apprendimento.

Risultati dell'apprendimento. I risultati dell'apprendimento sono interessanti. Per comodità, chiamiamo α il primo stimolo e β il secondo stimolo (ricordiamo che ognuno è costituito da una coppia di immagini).

Per i bambini che sono stati familiarizzati con l'insieme narrow, abbiamo che sia per lo stimolo α che per lo stimolo β il bambino è sempre colpito dal non-prototipo. Ciò che significa che il bambino ha categorizzato tutti gli elementi dell'insieme sotto una unica categoria (quella in cui vede il prototipo, che quindi non lo colpisce).

Per i bambini familiarizzati con l'insieme broad è sempre il prototipo (sia nello stimolo α che nello stimolo β) a colpire l'infante. Questo fatto indica che il bambino ha sviluppato due diverse categorie e riesce a piazzare i due non-prototipi ognuno nella sua categoria, ma la visione di un qualcosa che unisce le due categorie lo stupisce.

In Figura 5.12 vediamo i risultati ottenuti dalla SOM. È evidente che i risultati siano del tutto simili a quanto riscontrato negli infanti: per narrow abbiamo due distinte categorie, mentre per broad tutti i neuroni riconoscono una unica categoria.

Introdurre la componente acustica. È stato effettuato un ulteriore studio provando ad introdurre una componente acustica mentre si familiarizzavano gli infanti con l'insieme narrow (non è stato considerato l'insieme broad per questo esperimento). L'intento era quello di capire se tale componente potesse in qualche modo modificare l'apprendimento (e quindi il suo risultato) da parte degli infanti.

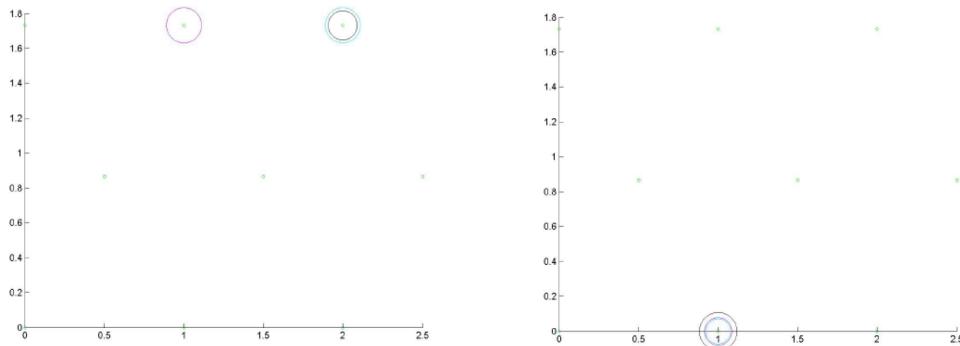


Figura 5.12: A sinistra, abbiamo i risultati per le reti addestrate con l'insieme narrow e a destra quelli per le reti addestrate con l'insieme broad.

Sono state delineate tre metodologie tramite le quali inserire la componente acustica:

- Prima metodologia (*A*): ai pupazzi con collo lungo è stata assegnata una parola inventata e ai pupazzi con gambe lunghe ne è stata assegnata un'altra. C'è quindi coerenza fra la componente acustica e quella visiva.
- Seconda metodologia (*B*): ad ogni immagine è stata assegnata una delle due parole usate in *A*, ma in maniera casuale.
- Terza metodologia (*C*): a tutte le immagini è stata associata una stessa parola inventata.

I risultati sono decisamente interessanti. Infatti, usando la prima metodologia non v'è stata alcuna differenza (si sono ottenute due categorie distinte), ma usando la seconda l'infante non è stato in grado di effettuare alcuna categorizzazione. Infine, usando la terza si è ottenuta una unica categoria.

Risulta dunque evidente che la componente acustica giochi un elemento di fondamentale importanza nel ruolo dell'apprendimento e possa cambiare radicalmente i risultati della categorizzazione.

Capitolo 6

Radial-Basis Function Networks

6.1 Introduzione

Voltiamo pagina ed andiamo ad introdurre una nuova tipologia di reti, le *radial-basis function network*. Tali reti uniscono in un certo senso quanto fanno le SOM e le classiche reti back propagation; infatti, pur mantenendo una struttura a livelli con hidden units, sfruttano anche un principio di località dei neuroni.

Una grande differenza rispetto alle reti back propagation consta nel fatto che i pesi che vanno dagli input verso i neuroni hidden abbiano significato differente rispetto ai pesi che vanno dalle unità hidden alla/e unità di output. Studieremo in seguito quale ruolo ricoprono questi pesi.

Come dice il nome stesso, questo tipo di reti si basa sul concetto di *raggio*. Più precisamente, la funzione di attivazione delle hidden units è una funzione radiale, ovvero una funzione il cui output dipende dalla distanza fra l'input e il neurone considerato. Ad ogni neurone è quindi associata una funzione radiale (che d'ora in poi chiameremo *RBF*).

In sostanza si ha che in una rete RBF i neuroni hidden utilizzano una funzione di attivazione RBF che ne descrive la recettività. Dopodiché, uno o più nodi finali sono utilizzati per combinare linearmente le attivazioni di ogni neurone. Possiamo immaginare questa combinazione come una interpolazione fra le attivazioni dei singoli neuroni hidden. In Figura 6.1 vediamo una rappresentazione grafica di quanto appena detto.

6.2 L'architettura di una rete RBF

In Figura 6.2 vediamo l'architettura di questa tipologie di rete.

Nell'esempio appena illustrato c'è una sola unità di output. Per semplicità ci riferiremo sempre a questo caso, ma tutto ciò che diremo varrà anche per reti

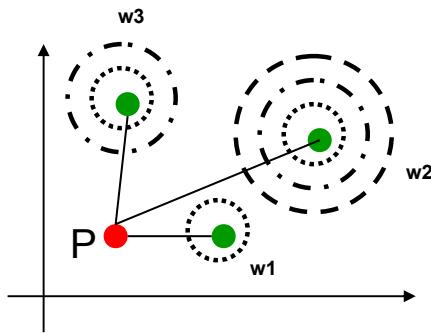


Figura 6.1: Esempio di RBF. w_1 , w_2 e w_3 sono i pesi fra il livello hidden ed il livello di output (e abbiamo $w_2 > w_3 > w_1$). Le sfere verdi rappresentano i neuroni che rispondono mediante RBF all'attivazione di un determinato input. Il vettore P , viene calcolato come interpolazione dei vettori derivanti dai neuroni, debitamente modulati tramite i pesi w .

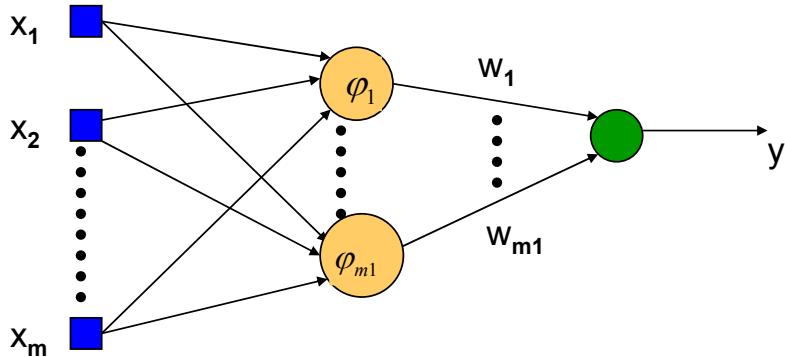


Figura 6.2: Ogni neurone hidden dispone di una funzione φ , che è la vera e propria funzione radiale. Il neurone di output riceve l'attivazione di ogni neurone hidden debitamente pesato tramite w_1 , ..., w_{m_1} .

con due o più unità di output.

L'output della rete risulta essere:

$$y = w_1\varphi_1(\|x - t_1\|) + \dots + w_{m_1}\varphi_{m_1}(\|x - t_{m_1}\|) \quad (6.1)$$

Dove x è il vettore in input e t_1, \dots, t_{m_1} sono i centri dei neuroni (che sono quindi definiti sui pesi fra gli input e i neuroni hidden). Sebbene la questione possa sembrare ostica, in realtà stiamo dicendo qualcosa di molto semplice.

Supponiamo di avere cinque input (quindi x conterrà cinque elementi): lo spazio delle soluzioni è dunque in uno spazio 5-dimensionale. Ogni unità di input sarà collegata a tutti i neuroni a livello hidden, e per ognuno di essi fornirà il valore della coordinata per quella unità di input (cioè per quella dimensione). Ecco quindi che un neurone riceve m pesi in entrata, tanti quanti sono gli input (cioè le dimensioni) ed è quindi possibile piazzare il neurone nello spazio utilizzando quei pesi.

Tornando alla formula, vediamo che la funzione radiale φ prende come parametro la differenza fra l'input x e il centro del neurone t_i , cioè la distanza fra il neurone e l'input. Avremo dunque che il valore della RBF sarà tanto più alto quanto più sarà piccola la distanza fra il centro del neurone e l'input. Ogni valore così calcolato è poi moltiplicato per w_i e sommato, così da combinare linearmente i vari output dei neuroni hidden.

6.2.1 La funzione di attivazione radiale

La funzione di attivazione radiale è in realtà leggermente più sofisticata di quanto abbiamo appena visto. Infatti si ha anche un ulteriore parametro σ , lo *spread* (o sensibilità). Lo spread è in grado di definire l'efficacia della funzione di attivazione diminuendone o ampliandone gli effetti.

La funzione di attivazione che useremo è la stessa già incontrata nelle SOM, la funzione gaussiana:

$$\varphi_\sigma(r) = \exp\left(-\frac{r^2}{2\sigma^2}\right)$$

con $\sigma > 0$. Nel caso di funzione gaussiana, uno spread più alto implica una sensibilità minore, cioè all'aumentare dello spread il neurone reagirà a input più distanti.

Si noti che usare una funzione radiale anziché una funzione sigmoidale (come quella delle reti back propagation) ha un importante impatto: le funzioni sigmoidali si attivano "da un certo punto in poi sempre", cioè sono asintotiche ad 1. Le funzioni radiali, al contrario, hanno due code asintotiche a zero e si attivano solo in una certa area, ovvero hanno una attivazione "locale". Questo significa che è possibile, per un determinato input, che nessuna unità si attivi (si parla di *punti di outline*): è un caso particolare che le funzioni sigmoidali difficilmente riescono a riprodurre.

6.3 Funzionamento della rete: un problema di interpolazione

Di fatto, il problema che deve risolvere la rete neurale è un problema di interpolazione. Dati infatti N punti in uno spazio m dimensionale $\{x_i \in \mathbb{R}^m, i = 1 \dots N\}$ (sono quindi gli input costituiti da m elementi) ed N numeri reali $\{d_i \in \mathbb{R}, i = 1 \dots N\}$ (che sono i valori di output che vogliamo ottenere, uno per ogni input), il nostro intento è trovare una funzione $F : \mathbb{R}^m \rightarrow \mathbb{R}$, tale che:

$$F(x_i) = d_i$$

cioè che dato un input m dimensionale (x_i) restituisca il corretto output reale (d_i). Si ha quindi una *funzione di interpolazione*.

Come sappiamo, dato un polinomio di grado sufficientemente alto possiamo rappresentare una qualsiasi figura e quindi possiamo ottenere la funzione interpolatrice che soddisfi quanto appena detto. Per ottenere ciò il grado della funzione deve essere pari al grado degli input, ovvero N .

Ogni neurone hidden definisce un punto, ergo dovremmo avere N neuroni hidden (cioè tanti quanti i pattern in input) per poter ottenere una funzione interpolatrice perfetta (priva di errore). Per ora poniamoci nella situazione di avere tutti questi N neuroni hidden, anche se chiaramente questa è una richiesta estremamente costosa: di fatto avremo una funzione interpolatrice approssimata, con un numero di neuroni in generale molto minore di N . Usare meno neuroni porta ad un vantaggio computazionale (e non solo, come vedremo in seguito).

6.3.1 Una rappresentazione matriciale

Riprendiamo la nostra definizione di output della rete:

$$F(x) = \sum_{i=1}^N w_i \varphi(\|x - x_i\|)$$

dove con x (senza pedici) rappresentiamo il vettore di input (di dimensione m) e i vari x_i sono i centri dei vari neuroni. La stessa equazione può essere riscritta in forma matriciale:

$$\begin{bmatrix} \varphi(\|x_1 - x_1\|) & \cdots & \varphi(\|x_1 - x_N\|) \\ \vdots & \vdots & \vdots \\ \varphi(\|x_N - x_1\|) & \cdots & \varphi(\|x_N - x_N\|) \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ d_N \end{bmatrix}$$

Si ponga particolare attenzione alla prima matrice, la notazione può sembrare confusionaria: abbiamo una riga per ogni pattern in input (da 1 ad N) e una colonna per ogni unità hidden (sempre da 1 a N). Dunque, la prima x si riferisce a quale degli N input consideriamo e la seconda x si riferisce al centro di dell'unità hidden a cui ci stiamo riferendo. Come già affermato, in questo caso abbiamo tante unità hidden quanti sono gli input per avere una interpolazione perfetta.

Vediamo poi \vec{w} , il vettore dei pesi fra il livello hidden ed i neuroni di output. Il risultato che vogliamo è \vec{d} , la lista dei valori a cui vogliamo associare i vari input. Si noti che qualora gli input siano più di uno, \vec{w} e \vec{d} diventano delle matrici che specificano tutti i pesi da ogni unità hidden verso ogni unità di output.

Se chiamiamo Φ la matrice iniziale, abbiamo tradotto il nostro problema in una banale equazione matriciale:

$$\Phi \cdot \vec{w} = \vec{d}$$

Il teorema di Michelli ci dice che Φ (proprio per la sua costruzione) gode di una particolare proprietà: è una matrice non singolare, ovvero con determinante diverso da zero. La matrice è anche quadrata ($N \times N$) e quindi grazie a queste due condizioni sappiamo che la matrice è invertibile¹. Questa proprietà ci sarebbe utile: potremmo infatti ottenere molto agilmente \vec{w} se potessimo invertire Φ (si risolve semplicemente il sistema lineare). Di fatto noi non useremo mai matrici quadrate, poiché impiegheremo molte meno di N unità hidden. Toneremo su questo discorso a breve.

6.3.2 Il numero di unità hidden

Abbiamo più volte accennato al fatto che avere tante unità hidden quanti sono gli elementi del dataset sia una soluzione impraticabile. Si avrebbe infatti un enorme costo computazionale, ma soprattutto si correrrebbe il rischio di *overfitting*. Avremmo infatti una rete in grado di riconoscere perfettamente il dataset (una funzione interpolatrice perfetta, appunto) ma non in grado di generalizzare a causa della sua troppa precisione sugli esempi dati.

Ecco quindi che il “problema” di dover usare meno neuroni di quanto sia il dataset non risulti in realtà un vero problema, ma anzi un passo obbligato.

La funzione di interpolazione si trasforma quindi in una approssimazione, ovvero avremo che:

$$F(x_i) \approx d_i$$

Data questa definizione è evidente che si richieda anche la delinazione di una tolleranza intorno ai vari d_i , entro la quale considereremo il valore in output uguale a d_i .

Usare meno neuroni hidden ha anche conseguenze sulla nostra matrice Φ , che, non essendo più quadrata (ma rettangolare, con molte più righe rispetto alle colonne) non gode più delle proprietà portate dal teorema di Michelli e quindi non è più invertibile. È un vero peccato poiché come abbiamo accennato l'invertibilità poteva esserci utile per il calcolo di \vec{w} : affronteremo in breve questo problema².

Unità hidden e trasformazioni di spazio

Grazie a quanto appena studiato, abbiamo la possibilità di precisare qualcosa che avevamo imparato fin dalle reti back propagation: l'introduzione di unità hidden permette di risolvere problemi non linearmente separabili poiché il livello hidden effettua una trasformazione di spazio, cercando di spostare gli input in

¹Una matrice quadrata e non singolare è certamente invertibile. Qualora una delle due condizioni non sia verificata, abbiamo certezza che la matrice *non* sia invertibile.

²Per i più frettolosi: useremo una *pseudo-inversa*.

modo che i neuroni di output siano in grado di separarli. Lo scopo delle unità hidden è quindi quello di rimappare gli input per rendere il problema posto ai neuroni di output come linearmente separabile.

6.4 Diversi algoritmi di apprendimento

Riassumendo quanto appena detto, abbiamo una equazione matriciale che ci descrive il funzionamento della rete. Ci sono diversi parametri da impostare per inizializzare la rete, più precisamente abbiamo:

- I centri delle funzioni RBF (in pratica i pesi dei neuroni hidden).
- Il valore di spread della funzione gaussiana usata come funzione di attivazione.
- I pesi che vanno dal livello hidden a quello di output (\vec{w}).

Diversi algoritmi di apprendimento imposteranno diversamente queste variabili. Vedremo tre diverse versioni dell'algoritmo di apprendimento.

6.4.1 Il bias

I più attenti si saranno chiesti: *dove è finito il bias?*

Siamo ora in grado di definire più precisamente il ruolo che ha avuto il bias nelle precedenti architetture. Esso, infatti, funge da *intercetta* che permette di spostare la soluzione (più precisamente l'iperpiano tracciato dalla soluzione) dall'origine.

Capiamo dunque che nel caso delle RBFN il bias ha senso soltanto a livello di output, mentre non ne ha a livello hidden (il centro è definito da n coordinate e non da $n + 1$, se n è la dimensione dello spazio in cui si trovano i neuroni). Pertanto, i neuroni a livello output avranno il bias mentre quelli a livello hidden no.

6.5 Algoritmo di apprendimento (1)

La prima versione dell'algoritmo di apprendimento definisce in maniera abbastanza banale i primi due parametri:

1. Il centro di ogni neurone è scelto casualmente fra i punti di input: ogni neurone è quindi piazzato su un pattern del dataset. Si cerca di evitare di piazzare due neuroni sullo stesso pattern.

2. Lo spread è definito come una sorta di normalizzazione:

$$\sigma = \frac{\text{Massima distanza fra due centri}}{\sqrt{\text{Numero di centri}}} = \frac{d_{\max}}{\sqrt{m_1}}$$

Dunque la funzione di attivazione per il neurone i con centro t_i sarà definita come:

$$\varphi_i(\|x - t_i\|^2) = \exp\left(-\frac{m_1}{d_{\max}^2} \cdot \|x - t_i\|^2\right)$$

La definizione dei pesi \vec{w} è invece decisamente più complessa.

6.5.1 La pseudo-inversione

Consideriamo la nostra equazione:

$$\begin{bmatrix} \varphi(\|x_1 - x_1\|) & \cdots & \varphi(\|x_1 - x_{m_1}\|) \\ \vdots & \vdots & \vdots \\ \varphi(\|x_N - x_1\|) & \cdots & \varphi(\|x_N - x_{m_1}\|) \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \vdots \\ w_{m_1} \end{bmatrix} \approx \begin{bmatrix} d_1 \\ \vdots \\ d_N \end{bmatrix} \Rightarrow \Phi \cdot \vec{w} \approx \vec{d}$$

Come si vede, ci sono solo m_1 neuroni presenti. Inoltre, l'equivalenza è diventata una approssimazione (per comodità torneremo ad usare il simbolo di uguale da ora in poi, ma rimane sempre vero che si tratta di una interpolazione approssimata).

Se Φ fosse una matrice quadrata e con determinante diverso da zero, potremmo farne l'inversa e ottenere in un batter d'occhio \vec{w} . Purtroppo, dato che Φ non è quadrata (a causa del numero di unità hidden molto minore rispetto al numero di pattern nel dataset, cioè $m_1 \ll N$), siamo certi che non sia invertibile.

Per risolvere il problema, proviamo a modificare l'equazione matriciale per cercare di isolare \vec{w} . Moltiplichiamo da sinistra entrambi i membri per la trasposta di Φ :

$$\Phi^T \Phi \cdot \vec{w} = \Phi^T \vec{d}$$

Dopodiché, facciamo l'inversa di $\Phi^T \Phi$ e la moltiplichiamo da sinistra su entrambi i lati dell'equazione:

$$(\Phi^T \Phi)^{-1} \Phi^T \Phi \cdot \vec{w} = (\Phi^T \Phi)^{-1} \Phi^T \vec{d}$$

Otteniamo così l'identità a sinistra, e abbiamo:

$$\vec{w} = (\Phi^T \Phi)^{-1} \Phi^T \vec{d}$$

Per semplicità, definiamo $(\Phi^T \Phi)^{-1} \Phi^T$ come Φ^\dagger , ottenendo la formula finale:

$$\vec{w} = \Phi^\dagger \vec{d}$$

Φ^\dagger è detta matrice *pseudo-inversa* di Φ .

L'utilità di questa trasformazione consta nel fatto che $\Phi^T \Phi$ è una matrice quadrata e quindi è *sperabilmente* una matrice non singolare. In tal caso, potremmo calcolarne l'inversa e impiegarla in Φ^\dagger per ottenere il nostro \vec{w} . Qualora però $\Phi^T \Phi$ sia singolare, il problema si ripropone: per ora ce ne dimentichiamo, ma lo affronteremo in futuro studiando la *Extreme Learning Machine*, che utilizzerà la regolarizzazione per risolvere il problema.

Si noti che non solo le matrici singolari sono problematiche, bensì tutte quelle matrici "quasi singolari", ovvero con un determinante prossimo allo zero.

Tralasciando questo (non) dettaglio, appuriamo di aver trovato una equazione sensazionale! Infatti, mediante questo calcolo matriciale è possibile ottenere in un solo passo l'intero vettore dei pesi \vec{w} . Chiaramente, tale vettore dipende dalla posizione dei centri che mano a mano vengono aggiornati durante l'apprendimento, ma intanto abbiamo trovato un modo per calcolare i pesi dalle unità hidden verso quelle di output con una operazione soltanto.

6.5.2 Ottimizzare il calcolo della pseudo-inversa

La matrice Φ può essere molto grande, e per calcolarne la pseudo-inversa è necessario mantenerla tutta in memoria. Tale requisito può essere problematico, pertanto, studiamo un metodo per poter effettuare il calcolo della pseudo-inversa senza coinvolgere direttamente tutta la matrice Φ .

Supponiamo di esplodere la nostra matrice Φ in una somma di matrici, come indicate in Figura 6.3. Ora sostituiamo nella nostra definizione di pseudo-inversa

$$\Phi = \sum_{i=1}^Q A_i$$

The diagram shows a large matrix Φ being decomposed into a sum of three smaller matrices, A_1 , A_2 , and A_3 . The matrices A_1 , A_2 , and A_3 have the same number of rows as Φ but fewer columns. The non-zero elements in Φ are mapped to the corresponding columns in A_1 , A_2 , and A_3 . All other elements are zero.

Figura 6.3: La matrice Φ è suddivisa in una somma di Q matrici, ognuna delle quali mantiene un numero *arbitrario* di righe della matrice iniziale (ma ogni riga è mantenuta in una sola matrice). Le matrici sono tutte di dimensioni pari a quella di Φ , e tutti gli elementi che non provengono dalla matrice iniziale sono posti a zero. In questo esempio, $Q = 3$.

la nuova Φ :

$$\vec{w} = (\Phi^T \Phi)^{-1} \Phi^T \vec{d} = \left(\sum_{j,i=1}^Q A_j^T A_i \right)^{-1} \left(\sum_{j=1}^Q A_j^T \right) \vec{d}$$

Calcoliamo separatamente le due parentesi.

Primo moltiplicando (H). Definiamo H :

$$H = \sum_{j,i=1}^Q A_j^T A_i$$

Il prodotto delle varie A_j^T e A_i è particolare. Infatti, per costruzione, ognuna delle due matrici avrà molte righe pari a zero. Dunque, l'unico di modo di ottenere un prodotto diverso da zero è che le righe diverse da zero della matrice A_i vengano moltiplicate per le colonne diverse da zero di A_j^T , ma questo accade solamente nel caso in cui A_i sia proprio A_j . In altre parole, se $j \neq i$ il prodotto fra $A_j^T A_i$ sarà pari a zero, mentre avremo un risultato diverso da zero solo se $j = i$, cioè se stiamo moltiplicando le matrici che mantengono lo stesso "pezzo" della Φ di partenza. Possiamo quindi modificare la nostra definizione di H :

$$H = \sum_{i=1}^Q A_i^T A_i$$

Il risultato di ogni moltiplicazione $A_i^T A_i$ produrrà quindi una matrice con tante righe diverse da zero quante quelle della A_i di partenza, e tutti gli altri elementi a zero. La somma dei vari prodotti, dunque, ci porterà ad avere H . Ma perché non rimuovere tutti quegli zeri (che occupano spazio in memoria)?

Definiamo una serie di matrici ridotte (\hat{A}_i), generate a partire dalle varie A_i rimuovendo però tutte le righe a zero. La dimensione di ogni \hat{A}_i sarà quindi diversa dalla dimensione di partenza di Φ : A_i avrà tante righe quante sono le righe che mantiene della matrice di partenza Φ , e avrà lo stesso numero di colonne di Φ .

La definizione di H che abbiamo appena ottenuto ci permette di sviluppare un algoritmo *parallelo* e che risparmia memoria per il calcolo di $\Phi^T \Phi$. Infatti possiamo:

1. Memorizzare \hat{A}_i (che ha molte meno righe di Φ).
2. Computare \hat{A}^T e quindi $\hat{A}_i^T \hat{A}_i$.
3. Aggiungere il risultato della computazione ad una lista, che, dopo Q step conterrà l'intera H .

In sostanza abbiamo rimosso tutti gli elementi a zero di ogni matrice, calcolato ogni addendo separatamente e poi *accodato* i vari risultati per ricostruire H . Le matrici \hat{A}_i sono storate in memoria una alla volta, e di fatto ogni moltiplicazione $\hat{A}_i \hat{A}_i^T$ è eseguibile su una macchina diversa.

Secondo moltiplicando. Prendiamo la seconda parte dell'equazione che stavamo calcolando:

$$\left(\sum_{j=1}^Q A_j^T \right) \vec{d}$$

È evidente che si possa applicare una tecnica simile a quanto fatto per il calcolo di H .

Andiamo ad introdurre Q vettori \vec{d}_i , ognuno dei quali riporta solo alcuni elementi del vettore \vec{d} di partenza (è ovviamente necessario che il "taglio" degli elementi sia coerente fra le varie A_j e i vari \vec{d}_i). Vediamo in Figura 6.4 tale separazione.

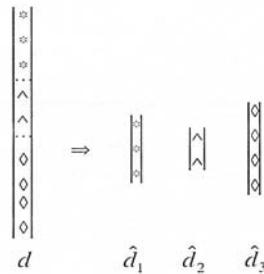


Figura 6.4: Il vettore \vec{d} è suddiviso in diverse parti, rispettando la suddivisione effettuata su Φ .

Come già fatto in precedenza, possiamo notare che gli unici prodotti che daranno un risultato diverso da zero sono quelli per cui le righe non nulle di A_j^T e \vec{d}_i corrispondono, dunque abbiamo che:

$$\sum_{j=1, i=1}^Q A_j^T \vec{d}_i = \sum_{i=1}^Q A_i^T \vec{d}_i$$

Anche qui possiamo definire vari \hat{d}_i ridotti, e computare quindi in maniera parallela ogni prodotto per poi accodare i risultati in una lista:

1. Memorizzare \hat{A}_i .
2. Computare \hat{A}^T e quindi $\hat{A}_i^T \hat{d}_i$.
3. Aggiungere il risultato della computazione ad una lista.

Dopo Q iterazioni avremo ottenuto il risultato di questa secondo moltiplicando.

Possiamo poi moltiplicare questo risultato per H^{-1} , che a sua volta sarà stata calcolata parallelamente, così da ottenere \vec{w} .

Si noti che questo approccio è decisamente flessibile: non solo possiamo parallelizzare il problema e risparmiare memoria, ma il numero di matrici tramite le quali esplodiamo Φ è anche arbitrario (cioè è arbitrario il numero di righe che ogni matrice A_i serba).

6.5.3 Riflessioni ed uso dell'algoritmo

L'algoritmo che abbiamo appena visto è decisamente interessante poiché permette, senza alcun procedimento iterativo, di ottenere il risultato desiderato. È evidente che il posizionamento casuale dei neuroni hidden possa fortemente condizionare le capacità di ragionamento della rete.

L'apprendimento mediante l'uso di questo algoritmo viene solitamente effettuato in maniera incrementale, ovvero, l'algoritmo è applicato più e più volte con un numero incrementale di unità hidden, finché non si ottiene un risultato accettabile in termini di discostamento dei valori ottenuti da quelli attesi sul test set.

Nonostante questo approccio a tentativi, non possiamo considerare l'algoritmo iterativo (anche perché questo aumento incrementale delle hidden unit viene applicato con ogni versione dell'algoritmo di apprendimento), ma si tratta sempre di un procedimento immediato con un risultato ottenuto in un passo soltanto.

6.6 Algoritmo di apprendimento (2)

È evidente che il primo algoritmo di apprendimento che abbiamo studiato dedichi molto lavoro al calcolo dei pesi \vec{w} concentrando poco sul centro dei neuroni, scelti randomicamente a inizio algoritmo.

La seconda versione dell'algoritmo concentra invece le sue energie sul calcolo dei centri dei neuroni.

6.6.1 I centri dei neuroni hidden

Per trovare il centro dei neuroni hidden si sfrutta un tipico algoritmo di clustering.

1. I centri vengono inizializzati randomicamente.
2. Si considera un input e lo disegna nello spazio in cui si trovano i neuroni hidden.

3. Si trova il centro più vicino all'input considerato (BMU):

$$k(x) = \min_k (\|x(n) - t_k(n)\|)$$

Ovvero il neurone k con centro t_k che ha la minima distanza dall'input x . Il parametro n indica l'iterazione del processo di clustering.

4. Si modifica il centro del neurone k (la best matching unit):

$$t_k(n+1) = t_k(n) + \eta(x(n) - t_k(n))$$

tutti gli altri neuroni rimangono invariati.

5. Ripetere dal punto 2 fino alla condizione di terminazione, ad esempio ogni BMU di ogni input è sufficientemente vicino all'input (vi sarà una soglia minima).

I più attenti avranno notato che l'aggiornamento dei centri è molto simile all'aggiornamento dei pesi della SOM, di fatto l'obiettivo che vogliamo ottenere è simile: rendere il neurone BMU in grado di rispondere per un determinato input.

6.6.2 Definizione degli spreads e dei pesi per il livello output

Una volta calcolati i centri mediante l'algoritmo di clustering, gli spread vengono definiti mediante normalizzazione, proprio come accade della prima versione dell'algoritmo che abbiamo studiato.

I pesi che collegano i neuroni hidden al livello di output sono invece calcolati mediante discesa del gradiente, ovvero calcolando la variazione di errore al variare dei pesi (come viene fatto nelle reti back propagation per l'ultimo livello).

6.6.3 Riflessioni ed uso dell'algoritmo

Questa versione dell'algoritmo è evidentemente più dispendiosa in termini di risorse: il processo è iterativo (sia per il clustering e sia per la discesa del gradiente) e dunque richiede (mediamente) più tempo.

Come già osservato per il primo algoritmo di apprendimento, il corretto numero di unità hidden da utilizzarsi viene trovato per tentativi, aumentando mano a mano il quantitativo di unità fino a ottenere risultati soddisfacenti. Per questa versione dell'algoritmo, però, si rende anche necessario definire il learning rate e questo implica ulteriori tentativi per trovare un valore appropriato.

6.7 Algoritmo di apprendimento (3)

L'ultima versione dell'algoritmo di apprendimento estende l'idea di usare la discesa del gradiente *su ogni parametro della rete*. Gli update si effettueranno quindi come segue:

- Modifica dei centri:

$$\delta t_j = -\eta_{t_j} \frac{\partial E}{\partial t_j}$$

- Modifica degli spread:

$$\delta \sigma_j = -\eta_{\sigma_j} \frac{\partial E}{\partial \sigma_j}$$

- Modifica dei pesi dell'ultimo livello:

$$\delta w_{ij} = -\eta_{w_{ij}} \frac{\partial E}{\partial w_{ij}}$$

6.8 Combinare gli algoritmi

Ogni versione dell'algoritmo ha i suoi vantaggi ed i suoi svantaggi. Il calcolo dei tre parametri si può però effettuare combinando i diversi algoritmi. Una delle versioni più usate consta ad esempio nel calcolare i centri mediante clustering, lo spread mediante normalizzazione e i pesi di output mediante pseudo-inversa.

6.9 Comparazione RBF-FFNN

Concludiamo questo capitolo comparando le reti RBF con quelle FFNN (ad esempio le reti back propagation). In realtà anche le reti RBF sono reti FFNN (poiché non presentano loop), ma sono abbastanza differenti dalle classiche FFNN poiché i due livelli svolgono una sorta di pre-processing abbastanza distinto rispetto a ciò che fa il secondo livello. In ogni caso, confrontandone le RBF rispetto al resto delle FFNN abbiamo che:

- Entrambe le tecniche sono esempi di reti non-lineari a livelli feed-forward.
- Entrambe sono approssimatori universali.
- **Architettura:** le reti FFNN possono avere più hidden layer mentre le RBF ne hanno solo uno.
- **Modello dei neuroni:** nelle reti FFNN i neuroni a livello hidden hanno un significato diverso rispetto ai neuroni a livello di output, cosa che nelle reti FFNN non accade. Inoltre, nelle reti FFNN ogni neurone (a qualsiasi livello appartenga) effettua una combinazione non-lineare, mentre nelle RBF si ha il livello hidden che è non-lineare e quello di output che è lineare.

- **Funzione di attivazione:** entrambe le reti sfruttano una misura di similarità. Le hidden unit delle reti RBF considerano la distanza euclidea tra il centro dell'unità e l'input, ma anche le reti FFNN effettuano una sorta di distanza, calcolata mediante prodotto scalare (che è infatti la definizione della cosine similarity).
- **Approssimazione:** Le reti RBF utilizzano una funzione gaussiana, dunque effettuano una approssimazione locale. Al contrario, le reti FFNN sfruttano la discesa del gradiente, impiegando quindi una approssimazione globale.

Capitolo 7

Extreme Learning Machine

7.1 Introduzione

L'*Extreme Learning Machine* (ELM) è un interessante tipologia di rete, che unisce le caratteristiche di generalità delle reti back propagation con le (veloci) tecniche di addestramento implementate nelle RBFN.

Guang-Bin Huang, nel suo articolo "Extreme learning machine: Theory and applications" definisce le ELM in termini di reti SLFN (single-hidden layer feedforward neural network) il cui addestramento viene effettuato mediante pseudo-inversa. Tecnicamente, con ELM indichiamo l'algoritmo di apprendimento per l'architettura SLF della rete, ma per comodità useremo ELM per definire sia l'architettura che l'algoritmo di apprendimento.

Non esiste più il concetto di centro dei neuroni, né di distanza radiale: la funzione di attivazione è la stessa su tutti i neuroni e può essere una qualsiasi (sigmoide, gaussiana, ecc.). L'articolo indica però come requisito necessario per il funzionamento della rete che la funzione di attivazione scelta sia *infinitamente derivabile*, requisito che verrà smentito in una successiva pubblicazione dello stesso Guang-Bin Huang.

Riassumendo, le ELM sono reti SLFN i cui pesi del livello hidden possono essere inizializzati tutti in maniera randomica, dopodiché, risolvendo analiticamente il sistema che presenta la pseudo-inversa (originata a partire dall'output dell'hidden layer) è possibile trovare la corretta assegnazione dei pesi fra il livello hidden e quello di output.

I più accorti si ricorderanno che la pseudo-inversa non sempre è calcolabile: introdurremo anche il concetto di *regolarizzazione* della pseudo-inversa al fine di rendere invertibili matrice prossime alla singolarità.

7.2 L'architettura SLFN

Conosciamo già l'architettura di questa rete, che è identica a quanto già studiato per le reti RBF. L'unica differenza è la presenza del bias anche nel livello hidden.

In Figura 7.1 vediamo un esempio di SLFN con due neuroni di output.

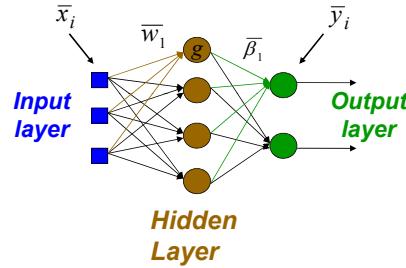


Figura 7.1: Rete SLF. In questo caso abbiamo due livelli di output. L'input è identificato da \bar{x} , la funzione di attivazione è g , i pesi fra input e neuroni hidden sono \bar{w} , i pesi fra i neuroni hidden e quelli di output sono β e l'output è identificato da \bar{y} .

Generalizzando, otteniamo l'equazione matriciale in Figura 7.2.

$$\begin{bmatrix} g(\bar{x}_1 \cdot \bar{w}_1) & \dots & g(\bar{x}_1 \cdot \bar{w}_{\tilde{N}}) \\ \vdots & & \vdots \\ g(\bar{x}_N \cdot \bar{w}_1) & \dots & g(\bar{x}_N \cdot \bar{w}_{\tilde{N}}) \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \dots & \beta_{1m} \\ \vdots & & \vdots \\ \beta_{\tilde{N}1} & \dots & \beta_{\tilde{N}m} \end{bmatrix} = \begin{bmatrix} y_{11} & \dots & y_{1m} \\ \vdots & & \vdots \\ y_{N1} & \dots & y_{Nm} \end{bmatrix}$$

Figura 7.2: Con \tilde{N} indichiamo il numero di neuroni hidden, mentre con N indichiamo il numero di pattern in input. L'equazione è del tutto simile a quella già vista per le RBFN, ma con output generalizzato ad m .

Dopo aver inizializzato a caso i pesi dei neuroni hidden, il nostro intento è trovare una corretta configurazione di β affinché gli output vengano ben riconosciuti.

Chiamando la prima matrice H (era Φ nel capitolo precedente), potremmo facilmente risolvere il sistema per ottenere β :

$$H\beta = y \implies \beta = H^{-1}y$$

Ma giacché H è una matrice non quadrata (poiché \tilde{N} è molto minore di N) risolveremo in realtà:

$$H\beta \approx T \implies \beta = H^\dagger T$$

ovvero una versione approssimata (T è circa y). H^\dagger è definita come abbiamo visto nel capitolo precedente (avremo occasione di riprendere il discorso in seguito).

Nei paragrafi a seguire formalizzeremo questo ragionamento, fornendo a supporto le nozioni teoriche riportate nel paper di Guang-Bin Huang.

7.3 I teoremi

Il lavoro di Guang-Bin Huang riporta due importanti teoremi.

7.3.1 Il teorema di interpolazione

Il teorema di interpolazione afferma che dati N pattern di input e una assegnazione casuale di pesi per N neuroni hidden è sempre possibile trovare una configurazione di pesi β tale per cui $\|H\beta - T\| = 0$. Cioè, possiamo risolvere l'equazione matriciale per trovare esattamente la matrice β che ci da il corretto target T . La Figura 7.3 illustra il teorema completo.

Theorem 2.1. *Given a standard SLFN with N hidden nodes and activation function $g : R \rightarrow R$ which is infinitely differentiable in any interval, for N arbitrary distinct samples $(\mathbf{x}_i, \mathbf{t}_i)$, where $\mathbf{x}_i \in \mathbf{R}^n$ and $\mathbf{t}_i \in \mathbf{R}^m$, for any \mathbf{w}_i and b_i randomly chosen from any intervals of \mathbf{R}^n and \mathbf{R} , respectively, according to any continuous probability distribution, then with probability one, the hidden layer output matrix \mathbf{H} of the SLFN is invertible and $\|\mathbf{H}\beta - \mathbf{T}\| = 0$.*

Figura 7.3: Teorema di interpolazione. Si noti che in realtà non c'è bisogno che la funzione di attivazione sia differenziabile (lo screenshot è preso dal paper originale).

7.3.2 Il teorema di approssimazione

Il teorema di interpolazione ci garantisce la presenza di soluzione avendo N unità hidden, ma noi ne abbiamo in realtà \tilde{N} , un numero ben minore. Il teorema di approssimazione garantisce che risolvendo il sistema usando la matrice rettangolare (o meglio la sua pseudo-inversa, supponendo che questa sia invertibile, e se non lo è vedremo in seguito come risolvere il problema) si ottiene comunque la soluzione con una tolleranza pari ad ϵ . La Figura 7.4 illustra il teorema completo.

Theorem 2.2. *Given any small positive value $\epsilon > 0$ and activation function $g : R \rightarrow R$ which is infinitely differentiable in any interval, there exists $\tilde{N} \leq N$ such that for N arbitrary distinct samples $(\mathbf{x}_i, \mathbf{t}_i)$, where $\mathbf{x}_i \in \mathbf{R}^n$ and $\mathbf{t}_i \in \mathbf{R}^m$, for any \mathbf{w}_i and b_i randomly chosen from any intervals of \mathbf{R}^n and \mathbf{R} , respectively, according to any continuous probability distribution, then with probability one, $\|\mathbf{H}_{N \times \tilde{N}} \beta_{\tilde{N} \times m} - \mathbf{T}_{N \times m}\| < \epsilon$.*

Figura 7.4: Teorema di approssimazione.

7.4 Algoritmo di apprendimento

Sfruttando i due teoremi appena riportati, possiamo implementare l'algoritmo di apprendimento che è il cuore di ELM. Sarà infatti sufficiente calcolare:

$$\beta = H^\dagger T$$

dove:

$$H^\dagger = (H^T H)^{-1} H^T$$

Ogni elemento di H sarà ottenuto applicando la funzione di attivazione su un determinato input (riga) da parte di un determinato neurone (colonna).

Ad esempio, se la funzione di attivazione fosse una sigmoide, avremmo per l'input i -esimo entrante nel neurone j -esimo (che pesi \bar{w}_j) il valore:

$$h_{ij} = g(\bar{x}_i; \bar{w}_j) = \frac{1}{1 + \exp\left(\frac{-\bar{x}_i \cdot \bar{w}_j}{\sigma}\right)}$$

7.4.1 Quando la matrice è prossima alla singolarità

Abbiamo accennato più e più volte a questa problematica: se la pseudo-inversa ($H^T H$) è prossima alla singolarità, il nostro algoritmo risulta impraticabile. Per risolvere questa problematica Guang-Bin Huang propone l'introduzione di un termine detto di *regolarizzazione* che, al prezzo di rendere la soluzione più imprecisa, allontana la pseudo-inversa dalla singolarità e permette quindi di effettuare l'apprendimento.

Si noti che utilizzando dati sperimentali è difficile ottenere una matrice quasi singolare, ma d'altronde, se le misure acquisite sono molto fini le differenze fra i dati potrebbero risultare minime (o nulle, a livello macchina) e quindi la singolarità potrebbe essere un problema col quale dover avere a che fare.

La regolarizzazione consta nell'aggiunta di uno scalare λ moltiplicato per la matrice identità I :

$$H^\dagger = (H^T H + \lambda I)^{-1} H^T$$

Dato che la matrice identità dispone di 1 sulla diagonale principale e di zero sul resto degli elementi, di fatto andiamo a sommare sulla diagonale principale di $H^T H$ la quantità λ . Giacché il calcolo del determinante è fortemente dipendente dal valore della diagonale principale, aggiungere il termine λ ne impedisce l'avvicinamento a zero. La definizione di λ è chiaramente dipendente da quanto "errore" vogliamo introdurre nella misura finale: non ci addentreremo così in profondità nel capire come definire un appropriato λ .

Un modo alternativo per considerare l'introduzione di questo λ è esplicitando l'errore che si vuole minimizzare:

$$E = E_D + \lambda E_W = \frac{1}{2} \sum_{j=1}^N \sum_{k=1}^Q \left(\left(t_j^k - \sum_{i=1}^M w_{ki} g(c_i \cdot x_j + b_i) \right)^2 + \frac{\lambda}{2} \sum_{i=1}^M \|w_{ki}\|^2 \right)$$

Nota bene: c'è stato un cambio notazionale: con w indichiamo i pesi dell'ultimo livello (dall'hidden all'output) mentre con c_i identifichiamo i pesi dall'input verso i neuroni hidden.

La formula indica con E_D l'errore derivante dalla computazione della rete (cioè la sottrazione fra il target e l'output della rete) e ad esso aggiunger E_W , il termine che permette la regolarizzazione. Tale termine, infatti, contiene la norma al quadrato del vettore dei pesi fra il livello hidden ed il livello di output: il nostro intento è ridurre E , di cui E_W è componente, dunque, vogliamo anche ridurre la norma al quadrato del vettore dei pesi. Questo significa, in altre parole, che vogliamo impedire che all'interno del vettore dei pesi ci sia grande distanza fra i vari elementi che questo contiene.

Imporre tale condizione ha senso perché è come dire che vogliamo una superficie "smooth", non inasprita dalle grande differenze all'interno di w . Tali differenze, infatti, equivalgono a disegnare un iperpiano che fitta perfettamente gli esempi che abbiamo fornito in fase di training, cosa che *non vogliamo* (onde evitare l'overfitting).

La regolarizzazione porta quindi due vantaggi intrinseci: i problemi instabili sono più facilmente risolvibili e la soluzione ottenuta è più generale.

Il termine di regolarizzazione può avere molte forme diverse, ma scegliere di minimizzare la norma dei pesi è risultato particolarmente vantaggioso; infatti, il teorema di Bartlett afferma che: "minore è la norma dei pesi, maggiore è la capacità di generalizzazione della rete".

Per tutte queste ragioni la regolarizzazione viene talvolta utilizzata anche nelle reti che abbiamo già studiato, come ad esempio le reti back propagation.

Terminiamo questa discussione mettendo in luce il fatto che sebbene l'introduzione di λ porti ad una imprecisione nel risultato (che teoricamente andrebbe sommata a ϵ , l'imprecisione data dalla presenza di \tilde{N} neuroni hidden), porta comunque ad una soluzione che *altrimenti non avremmo avuto*. Inoltre, non è "così grave" l'imprecisione del risultato: ci allontana dall'overfitting, che è sempre cosa gradita.

7.4.2 Calcolare H mediante Single Value Decomposition

Un modo alternativo per calcolare la H (attenzione, non la pseudo-inversa) consiste nello sfruttare la *Single Value Decomposition* (SVD).

Questa tecnica (che è usata in moltissime applicazioni) permette di scindere una matrice in tre diverse matrici:

$$H = U \cdot S \cdot V^T$$

La prima e l'ultima matrice sono matrici rettangolari rispettivamente di dimensioni $n \times m$ ed $m \times n$. La matrice centrale, invece, è una matrice quadrata costituita da zeri, tranne sulla diagonale principale. S è di fondamentale importanza poiché ordina dall'alto verso il basso le componenti più "importanti" della matrice di partenza.

U e V^T sono matrici costituite da righe (colonne per V^T) ortonormali (linearmente indipendenti e con norma pari ad 1).

Applicando la SVD ad H^\dagger (tralasciamo per semplicità tutta la trafia aritmetica) si ottiene:

$$H^\dagger = V \Sigma^\dagger U^T$$

La matrice centrale, Σ^\dagger , è "parente" della S iniziale. Infatti, se S conteneva sulla sua diagonale principale gli elementi σ_{ii} , Σ^\dagger mantiene sulla sua diagonale principale gli elementi $\frac{1}{\sigma_{ii}}$.

È chiaro che durante l'esposizione di SVD abbiamo dimenticato il discorso della regolarizzazione: è comunque possibile integrare le due tecniche.

Infatti, trasformare l' H mediante SVD non permette affatto di risolvere i problemi in merito alla singolarità: se H è prossima alla singolarità, gli elementi di Σ^\dagger saranno molto grandi (poiché i σ_{ii} saranno molto piccoli). Ecco quindi che la regolarizzazione risulta comunque necessaria.

7.5 Quando usare ELM e che vantaggi porta

Sempre secondo Guang-Bin Huang, utilizzare ELM permette di velocizzare di più di 100 volte l'addestramento di una rete.

ELM viene per queste ragioni preferito in molte applicazioni, ma, quando i dati da trattare sono molti (pensiamo ad esempio ai dati che possono essere ottenuti dal web) si preferisce ancora utilizzare la discesa del gradiente (gestire matrici di grosse dimensioni risulta impraticabile).

7.5.1 Qualche risultato

In Figura 7.5 vediamo i *Root Medium Square Error* ottenuti dalla computazione di reti BP e ELM su un insieme di benchmark. Tali valori sono ottenuti prima di tutto normalizzando i valori di input (cioè riportando dati possibilmente eterogenei in uno stesso intervallo) dopodiché per ogni pattern input si

calcola la differenza fra il valore atteso e quello ottenuto dalla rete (al quadrato). Una volta ottenute tutte le differenze al quadrato le si mette sotto radice e si ha l'RMSE.

Ricordiamo che questo tipo di valutazione viene effettuata per problemi di *regression*.

| Data sets | BP | | ELM | |
|--------------------|----------|---------|----------|---------------|
| | Training | Testing | Training | Testing |
| Abalone | 0.0785 | 0.0874 | 0.0803 | 0.0824 |
| Delta ailerons | 0.0409 | 0.0481 | 0.0423 | 0.0431 |
| Delta elevators | 0.0544 | 0.0592 | 0.0550 | 0.0568 |
| Computer activity | 0.0273 | 0.0409 | 0.0316 | 0.0382 |
| Census (house8L) | 0.0596 | 0.0685 | 0.0624 | 0.0660 |
| Auto price | 0.0443 | 0.1157 | 0.0754 | 0.0994 |
| Triazines | 0.1438 | 0.2197 | 0.1897 | 0.2002 |
| Machine CPU | 0.0352 | 0.0826 | 0.0332 | 0.0539 |
| Servo | 0.0794 | 0.1276 | 0.0707 | 0.1196 |
| Breast cancer | 0.2788 | 0.3155 | 0.2470 | 0.2679 |
| Bank domains | 0.0342 | 0.0379 | 0.0406 | 0.0366 |
| California housing | 0.1046 | 0.1285 | 0.1217 | 0.1267 |
| Stocks domain | 0.0179 | 0.0358 | 0.0251 | 0.0348 |

Figura 7.5: Confronto del RMSE ottenuto su diversi benchmark utilizzando Back Propagation ed ELM. I valori in grassetto indicano i benchmark sui quali l'ELM "si comporta meglio". Tale definizione porta con sé un significato statistico: ogni valore in tabella è ottenuto in realtà calcolando l'RMSE medio ottenuto mediante molte esecuzioni sullo stesso benchmark. Di tale distribuzione di valori si studia poi la deviazione standard, e, vengono segnati in grassetto soltanto quei valori che sono migliori rispetto alla controparte e che distano almeno la deviazione standard rispetto alla controparte. I valori riportati sono ancora soggetti alla normalizzazione (solitamente si riportano all'unità di partenza, ma Guang-Bin Huang non si è sporcati le mani).

In Figura 7.6 abbiamo invece un confronto fra tempi per gli stessi benchmark.

| Data sets | BP ^a (s) | | ELM ^a (s) | |
|--------------------|---------------------|------------|----------------------|------------|
| | Training | Testing | Training | Testing |
| Abalone | 1.7562 | 0.0063 | 0.0125 | 0.0297 |
| Delta ailerons | 2.7525 | 0.0156 | 0.0591 | 0.0627 |
| Delta elevators | 1.1938 | 0.0125 | 0.2812 | 0.2047 |
| Computer activity | 67.44 | 0.0688 | 0.2951 | 0.172 |
| Census (house8L) | 8.0647 | 0.0457 | 1.0795 | 0.6298 |
| Auto price | 0.2456 | $<10^{-4}$ | 0.0016 | $<10^{-4}$ |
| Triazines | 0.5484 | $<10^{-4}$ | $<10^{-4}$ | $<10^{-4}$ |
| Machine CPU | 0.2354 | $<10^{-4}$ | 0.0015 | $<10^{-4}$ |
| Servo | 0.2447 | $<10^{-4}$ | $<10^{-4}$ | $<10^{-4}$ |
| Breast cancer | 0.3856 | $<10^{-4}$ | $<10^{-4}$ | $<10^{-4}$ |
| Bank domains | 7.506 | 0.0466 | 0.6434 | 0.2205 |
| California housing | 6.532 | 0.0469 | 1.1177 | 0.3033 |
| Stocks domain | 1.0487 | 0.0063 | 0.0172 | 0.0297 |

^aRun in MATLAB environment.

Figura 7.6: Confronto dei tempi ottenuto su diversi benchmark utilizzando Back Propagation ed ELM.

7.5.2 Gli effetti della regolarizzazione

Per completezza, riportiamo l'andamento dell'RMSE al variare delle unità hidden sul domino *Abalone* nel caso in cui si utilizzi o meno la regolarizzazione.

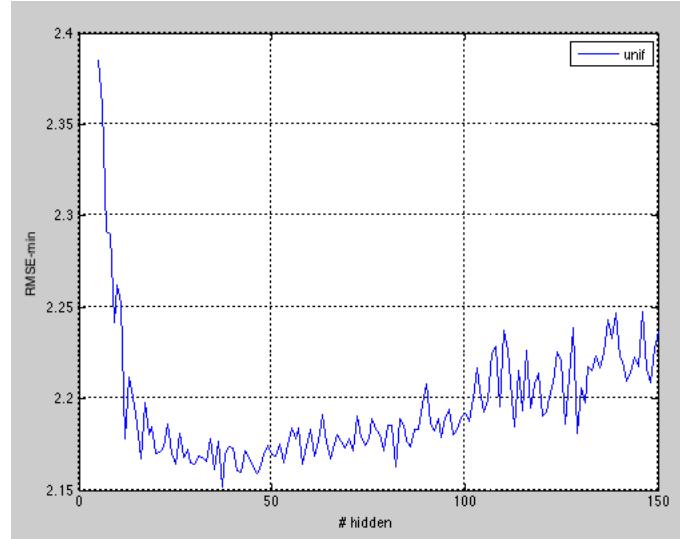


Figura 7.7: Andamento del RMSE rispetto ai nodi di hidden *senza usare la regolarizzazione* durante il calcolo della pseudo-inversa.

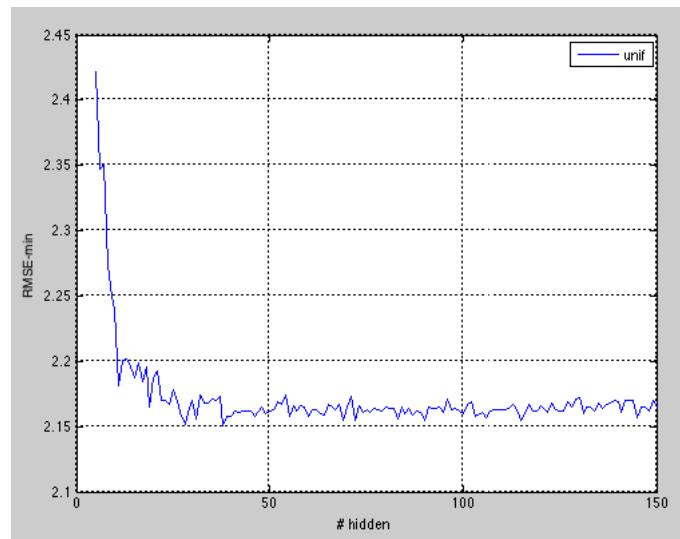


Figura 7.8: Andamento del RMSE rispetto ai nodi di hidden *usando la regolarizzazione* durante il calcolo della pseudo-inversa.

Senza utilizzare la regolarizzazione si ha che all'aumento dei nodi hidden la rete non è più in grado di "gestire" le sue unità computazionali ed il calcolo della pseudo-inversa risulta sempre più impreciso (ovvero la matrice sempre

più prossima alla singolarità ed i calcoli sballati). Al contrario, utilizzando la regolarizzazione l'errore tende a stabilizzarsi indipendentemente dal numero di hidden units utilizzate.

Capitolo 8

Hopfield Networks

In questo capitolo spostiamo la nostra attenzione verso una tipologia di rete decisamente diversa rispetto alle feedforward neural networks studiate in precedenza: le *Hopfield networks* sono infatti la più semplice istanza di *recurrent neural networks* (RNN). La fondamentale caratteristica delle RNN è il fatto che l'output della rete funga da input per la rete stessa (da cui l'appellativo "recursive").

Sarà quindi nostro obiettivo capire le differenze che vi sono fra le due tipologie di reti, andando a studiare in dettaglio come funzionano le Hopfield networks.

8.1 Introduzione

Le reti di Hopfield si basano su una architettura *ricorrente* e fanno uso di un addestramento (quasi del tutto) *non supervisionato*. Si tratta di un modello un po' obsoleto e che soffre di diverse problematiche (di cui parleremo debitamente in seguito) ma fornisce ottimi spunti didattici.

8.1.1 Memorie associative

Le reti di Hopfield vengono sfruttate come *memorie associative*, ovvero strumenti impiegati al fine di ricondurre un determinato input ad un pattern precedentemente cablato nella rete (operazione detta *associazione*). Il punto interessante è che l'input fornito può essere anche alterato o corrotto rispetto al prototipo mantenuto dalla rete: starà proprio alla rete saper abilmente associare l'input corrotto al corretto prototipo. In Figura 8.1 vediamo tre esempi. I prototipi mantenuti all'interno della rete a cui vorremo associare i vari input durante l'uso della rete vengono detti *memorie fondamentali* o *attuatori* e come impareremo tali memorie verranno cablate mediante dei pesi sugli archi che collegano i neuroni.

Scopo della "fase di apprendimento" (detta in realtà *storage*) di queste re-



Figura 8.1: Tre esempi d'uso di una rete di Hopfield. A destra abbiamo il prototipo mentre a sinistra abbiamo l'input: attraverso una serie di passaggi (al centro) l'input verrà ricondotto al corretto prototipo.

ti è quello di trovare la adeguata combinazione di pesi affinché la rete possa correttamente associare gli input alle memorie fondamentali.

8.2 L'architettura

8.2.1 Da cosa è costituita la rete

Ora che abbiamo un'idea di cosa vogliamo faccia la nostra rete, andiamo ad elencare le sue componenti.

Supponendo di avere un input codificato in N valori, si avranno N neuroni *binari*; si ha perciò che ogni neurone rappresenta una componente dell'input. Sia l'input della rete che l'uscita dei neuroni sono codificati mediante valori binari (assumono valore 1 oppure -1).

Si ha poi che l'output (binario) di ogni neurone funge da input per tutti gli altri (ma non per se stesso). Questa connessione fra i neuroni è *pesata* mediante particolari valori che sono ottenuti in fase di storage (lo abbiamo accennato sopra). I pesi rappresentano quindi la correlazione delle varie componenti (ognuna rappresentata dal suo neurone) relativamente alle memorie fondamentali impiegate in fase di storage. Si avrà dunque che un peso molto grande indicherà il fatto che le due componenti sono spesso "simili" nelle memorie fondamentali.

È evidente che i pesi siano simmetrici, ovvero l'arco uscente dal neurone i ed entrante in j ha lo stesso peso dell'arco uscente da j ed entrante in i .

In Figura 8.2 vediamo una rappresentazione di quanto appena descritto.
È evidente che la struttura ricorsiva della rete debba prevedere un concetto di *istante*: se non imponiamo una sorta di temporizzazione non saremo in

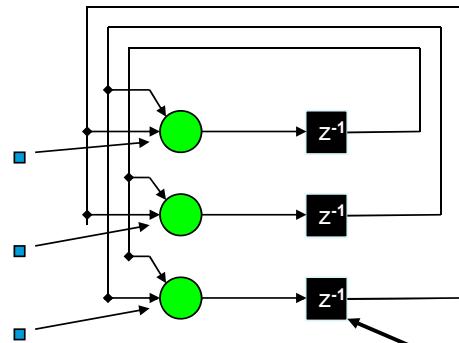


Figura 8.2: L'architettura di una rete di Hopfield con input a tre componenti. In verde abbiamo i neuroni, mentre i quadratini blu rappresentano l'input. L'output di ogni neurone è collegato all'input di tutti gli altri (ma non di sé stesso). In nero abbiamo dei particolari operatori di cui parleremo in seguito.

grado di leggere nulla dalla rete (si ha infatti una sorta di loop infinito). Più precisamente definiamo:

- Lo *stato di un neurone* al tempo n come il suo valore di output a quell'istante (sarà un valore binario).
- Lo *stato della rete* al tempo n come l'insieme degli stati dei vari neuroni a quell'istante (sarà una lista di valori binari).

I due punti sembrano banali, ma in alcuni modelli più complessi lo stato di un neurone può essere funzione dell'output o viceversa: non necessariamente i due concetti sono coincidenti.

Il concetto di evoluzione temporale a cui eravamo abituati con le reti FFNN è ribaltato nelle reti di Hopfield: nelle FFNN la fase di apprendimento era iterativa e l'uso delle reti era one-shot, al contrario qui abbiamo uno storage immediato e un uso della rete "iterativo". Come impareremo, nelle reti di Hopfield il concetto di iterazione è legato al concetto di *livello energetico*.

Si noti che l'input che viene fornito alla rete in fase di utilizzo *non è pesato* (ma una volta entrato in circolo nella rete gli output che quell'input ha generato verranno chiaramente pesati).

8.3 Il funzionamento della rete

Vediamo ora come funziona la rete, ovvero come è possibile computare i pesi cablando le memorie fondamentali e come è possibile sfruttare la rete per sapere a quale prototipo è stato associato il nostro input.

Un po' di notazione

Definiamo:

- N è la dimensione dell'input, quindi avremo N neuroni e output di dimensione N .
- M è la quantità di memorie fondamentali che vogliamo cablare nella rete.
- $f_{\mu i}$ è la i -esima componente (su N) della μ -esima memoria fondamentale (su M).
- $x_i(n)$ è lo stato dell' i -esimo neurone al tempo n . Come sappiamo, può valere -1 oppure 1.

8.3.1 Fase di storage

Abbiamo già anticipato che la fase di storage è il corrispettivo in ambiente RNN della fase di apprendimento per le reti FFNN. In questa vase vengono definiti tutti i pesi che collegano fra loro i neuroni.

Ricordiamo che non vi sono self-loop e i pesi sono simmetrici: in pratica la matrice dei pesi è una matrice simmetrica con diagonale principale a 0.

Definendo come f_1, f_2, \dots, f_M le N -dimensionali memorie fondamentali che vogliamo cablare nella rete, definiremo ogni peso come:

$$w_{ji} = \begin{cases} \frac{1}{M} \sum_{\mu=1}^M f_{\mu i} f_{\mu j} & \text{se } j \neq i, j = 1, \dots, N \\ 0 & \text{se } j = i \end{cases}$$

In pratica il peso della connessione fra i neuroni i e j (se i è diverso da j , no self-loops) è ottenuto come il valore medio della moltiplicazione fra i valori assunti da ogni memoria fondamentale sulle due componenti. Ricordiamo che le varie f assumono valori binari, dunque, il valore dei pesi non potrà che variare fra -1 e 1.

Al termine di questa fase i pesi sono fissati e non verranno più modificati.

8.3.2 Uso della rete

Vediamo ora come viene utilizzata la rete al fine di ricondurre un certo input x_{probe} ad uno dei prototipi memorizzati durante la fase di storage.

Inizializzazione. L'algoritmo è inizializzato settando su ogni neurone j :

$$x_j(0) = x_{\text{probe},j} \quad \text{con } j = 1, \dots, N$$

cioè ogni neurone riceve come input (stato all'istante 0) il valore della corrispondente componente dall'input fornito alla rete (sarebbero i quadrati blu di Figura 8.2). Come avevamo già accennato, questi valori non vengono pesati ma vengono direttamente forniti ai neuroni così come sono.

Iterazione fino alla convergenza. Non resta che lasciar "fluire" i valori all'interno della rete, ovvero, leggere i vari $x(n)$ incrementando mano a mano gli n . Questa operazione può essere compiuta sia simultaneamente (ovvero tutti i neuroni passano da $x(n)$ ad $x(n+1)$ insieme) che in maniera asincrona, neurone per neurone (con un ordinamento random oppure scelto a monte). Nel nostro caso effettuiamo un update asincrono randomico, usando la formula:

$$x_j(n+1) = \text{sign} \left[\sum_{i=1}^N w_{ji} x_i(n) \right] \quad \text{con } j = 1, \dots, N$$

Si computa quindi la somma di tutti i valori entranti in j (i vari neuroni i) debitamente moltiplicati col peso che collega j ad i . Se tale somma è positiva, j produrrà valore $+1$, altrimenti produrrà valore -1 (questo è implementato mediante la funzione sign). Si noti che fra i vari i considerati vi sarà anche j , ma per come abbiamo definito i pesi nella fase di storage w_{ji} varrà 0 se i e j sono coincidenti e dunque l'intero addendo varrà 0.

Abbiamo detto che questa operazione di aggiornamento deve essere compiuta fino alla convergenza, ovvero fino a che il vettore x non cambia più (in nessuna delle sue componenti, ovvero lo stato di nessun neurone muta ulteriormente). È evidente che per poter accettare questa condizione di terminazione si debbono avere delle certezze in merito alla convergenza. Per ora ci limitiamo a dire che l'algoritmo *converge sempre*, e lo dimostreremo nei paragrafi a seguire.

Per poter computare in maniera asincrona i vari valori prodotti dalla rete si fa uso di *operatori di sincronizzazione* (i quadrati neri in Figura 8.2) che non fanno altro che permettere di leggere i dati impedendo la cortocircuitazione della rete.

Leggere l'output. Non resta che leggere il nostro output, cioè il punto fisso della rete, l'insieme di valori contenuti in $x(n)$ che non varia più col procedere delle iterazioni (si ha cioè che $x(n+1) = x(n)$). Questo insieme è detto *stato stabile*. Se chiamiamo x_{fixed} questo vettore, avremo che l'output della rete sarà:

$$y = x_{\text{fixed}}$$

Sperabilmente, il punto fisso della rete sarà equivalente ad una delle memorie fondamentali che abbiamo cablato nella rete e ciò indicherà che il nostro input è stato associato a quel prototipo. Vedremo in seguito che ciò potrebbe non essere vero (ed è ovviamente un problema).

8.3.3 Un esempio

Vediamo ora un esempio d'uso delle reti di Hopfield. Osserviamo la Figura 8.3.

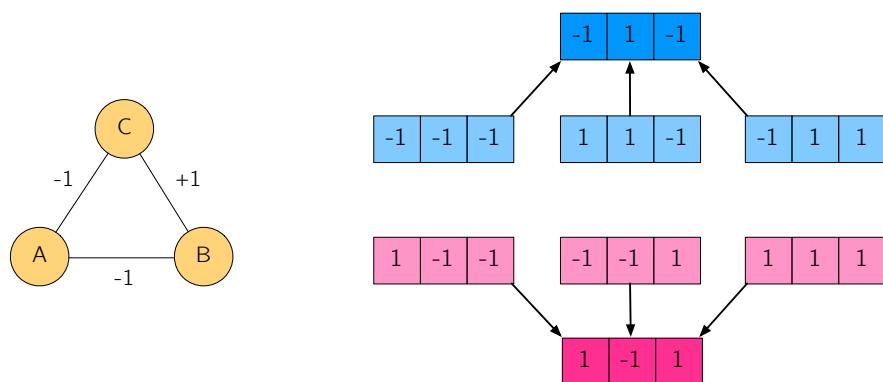


Figura 8.3: A sinistra abbiamo la configurazione dei pesi della rete: in arancione i neuroni e sugli archi i pesi che li collegano. A destra abbiamo invece rappresentato diverse cose: $[-1, 1, -1]$ e $[1, -1, 1]$ sono due stati stabili (ovvero le memorie fondamentali con le quali abbiamo effettuato la fase di storage). Sono poi rappresentati tutti i possibili stati e, come si vede, ognuno di essi "tende" dopo qualche iterazione a diventare uno dei due stati stabili. Questo fenomeno di convergenza ad uno dei due stati stabili (cioè a una delle memorie fondamentali) lo spiegheremo nel prossimo paragrafo parlando delle garanzie di convergenza.

8.4 In merito alla convergenza

È evidente che senza un teorema di convergenza un'algoritmo di questo tipo sarebbe di poco uso. Fortunatamente è dimostrabile che la rete tende sempre *ad uno stato stabile*, che, nella nomenclatura che stiamo per introdurre, equivale ad uno stato di *minima energia*.

8.4.1 Il concetto di energia

Uno dei modi per interpretare il funzionamento della rete e l'evolversi degli stati di output è quello di affiancare ad ogni stato un determinato *livello di energia*. Dunque, l'energia è una funzione che mappa ogni stato in un determinato valore, ovvero una funzione $2^N \rightarrow \mathbb{R}$ (ricordiamo che avendo N valori binari in ogni stato vi sono esattamente 2^N stati).

Il livello di energia è definito come segue:

$$E(x) = -\frac{1}{2} \sum_i^N \sum_j^N w_{ji} x_i x_j$$

Giacché gli stati stabili sono quelli a livello di energia minimo (lo dimostreremo in seguito) il nostro intento è quello di dimostrare che la transizione fra uno stato x verso uno stato x' porta ad una diminuzione di energia, ovvero, accade sempre che $E(x') < E(x)$. Se riusciamo a dimostrare questo passaggio, dato il fatto che abbiamo un numero finito di stati, avremo che certamente l'algoritmo terminerà non appena giungerà allo stato con livello di energia minimo (uno stato stabile, appunto).

Gli stati stabili *non sono* tutti allo stesso livello minimo di energia (ovvero non sono tutti ad un minimo numerico assoluto) e questo potrebbe comportare dei problemi se l'immagine di partenza è troppo distante dalla memoria stabile a cui vogliamo che la rete l'associ. Infatti, data l'immagine di partenza, l'algoritmo si avvicinerà sempre più ad uno stato stabile e, una volta raggiunto, non potrà transire ad uno stato stabile a livello di energia ancora più basso a meno che quello stato stabile si "a distanza uno" da quello raggiunto (in questo caso sarebbe improprio chiamare "stato stabile" il primo). Il concetto di "distanza" fra stati può essere pensato come distanza di Hamming, ovvero la quantità di neuroni che devono avere output diverso per passare dal primo stato al secondo.

Si tenga conto di un particolare: la modifica di x accade *localmente* (poiché noi modifichiamo in maniera asincrona lo stato di un neurone alla volta) ma in realtà questo ha effetto sull'energia globale ($E(x)$).

Gli stati stabili hanno un basso valore di energia

Una memoria fondamentale stabilisce (o almeno contribuisce a stabilire) la correlazione che intercorre fra due neuroni, ovvero, neuroni tendono avere lo stesso stato se sono concordi nella memoria fondamentale e stato opposto se sono discordi nella memoria fondamentale.

Se dunque w_{ji} è un valore grande ci si aspetta che le caratteristiche rappresentate dai neuroni i e j si comportino in maniera similare, al contrario, un valore molto piccolo (negativo) di w_{ji} indicherebbe una correlazione opposta fra le caratteristiche.

Ma allora $\sum_{i,j} w_{ji} x_i x_j$ è *sempre* un valore grande se x è una memoria associativa. Questo è banalmente vero, poiché se $x_i x_j > 0$ allora w_{ji} sarà maggiore di zero ed il tutto sarà maggiore di zero, al contrario se $x_i x_j < 0$ allora w_{ji} sarà minore di zero, riportando l'intero prodotto alla positività. Ma, giacché nella

definizione di energia noi prendiamo tutta la somma negata, avremo che *gli stati stabili hanno sempre un basso valore di energia.*

8.4.2 Il teorema di convergenza

Vogliamo dimostrare che la modifica di stato di un neurone diminuisce sempre l'energia E definita come sopra. Chiamiamo ℓ il neurone di cui stiamo aggiornando lo stato (operazione detta *fire*) e chiamiamo x_ℓ lo stato di partenza di ℓ , mentre x'_ℓ è lo stato aggiornato.

Supponendo che il fire di ℓ modifichi il suo stato, abbiamo solo due casistiche da prendere in considerazione:

1. Lo stato di x_ℓ passa da -1 a 1. Ciò implica che:

$$\sum_{i=1}^N w_{\ell i} x_i > 0$$

(per come abbiamo definito l'update dei vari neuroni).

2. Lo stato di x_ℓ passa da 1 a -1. Ciò implica che:

$$\sum_{i=1}^N w_{\ell i} x_i < 0$$

(per come abbiamo definito l'update dei vari neuroni).

Perciò possiamo dire che:

$$(x'_\ell - x_\ell) \sum_{i=1}^N w_{\ell i} x_i > 0$$

Il fatto che questa quantità sia positiva ci tornerà utile in seguito.

Andiamo ora a lavorare sulla formula di E al fine di fattorizzare tutte le componenti che dipendono da x_ℓ . Sappiamo che:

$$E(x) = -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N w_{ji} x_i x_j$$

Volendo, possiamo portare a fattor comune x_i dalla sommatoria più interna:

$$E(x) = -\frac{1}{2} \sum_{i=1}^N x_i \sum_{j=1}^N w_{ji} x_j$$

Andiamo poi ad estrapolare il caso in cui $i = \ell$ nella sommatoria esterna:

$$E(x) = -\frac{1}{2}x_\ell \sum_{j=1}^N w_{j\ell} x_j - \frac{1}{2} \sum_{\substack{i=1 \\ i \neq \ell}}^N x_i \sum_{j=1}^N w_{ji} x_j$$

Abbiamo quindi sdoppiato la sommatoria esterna (che contiene anche la seconda) esplicitando il caso $i = \ell$. Si noti che nella sommatoria del primo addendo avremmo anche potuto escludere il caso $j = \ell$, dato che in quel caso si avrà il peso $w_{j\ell}$ pari a zero, ma onde evitare di complicare la notazione non l'abbiamo esplicitato.

Proseguiamo con la fattorizzazione di ℓ , che è ancora presente nel loop relativo a j . Andiamo ad estrarporarlo anche da lì (separiamo il caso in cui $j = \ell$):

$$E(x) = -\frac{1}{2}x_\ell \sum_{j=1}^N w_{j\ell} x_j - \frac{1}{2} \sum_{\substack{i=1 \\ i \neq \ell}}^N x_i w_{\ell i} x_\ell - \frac{1}{2} \sum_{\substack{i=1 \\ i \neq \ell}}^N x_i \sum_{\substack{j=1 \\ j \neq \ell}}^N w_{ji} x_j$$

Si noti che anche in questo caso avremmo potuto escludere $i = \ell$ nella sommatoria del secondo addendo: si avrà infatti $w_{i\ell}$ pari a zero per annullare l'addendo quando consideriamo il self-loop.

Osservando attentamente la nostra sommatoria, vediamo che vi sono tre addendi: i primi due coinvolgono x_ℓ mentre l'ultimo ne è indipendente. Possiamo quindi pensare alla nostra formula come qualcosa del genere:

$$E(x) = -\frac{1}{2}x_\ell \sum_{j=1}^N w_{j\ell} x_j - \frac{1}{2} \sum_{i=1}^N x_i w_{\ell i} x_\ell + \text{Termine indipendente da } x_\ell$$

Concentrandoci poi sui primi due fattori, possiamo vedere come siano in realtà la stessa cosa a causa della simmetria dei pesi. Ecco quindi la formula definitiva:

$$E(x) = -x_\ell \sum_{j=1}^N w_{j\ell} x_j + \text{Termine indipendente da } x_\ell$$

Ora che siamo riusciti ad estrapolare l'apporto fornito da x_ℓ all'energia, possiamo calcolare la differenza di energia prima e dopo il fire di ℓ ! Calcoliamo perciò:

$$E' - E = -x'_\ell \sum_{j=1}^N w_{j\ell} x_j + x_\ell \sum_{j=1}^N w_{j\ell} x_j$$

dove E' è l'energia dopo il fire di ℓ mentre E è l'energia prima del fire di ℓ . Ovviamente il fattore indipendente da x_ℓ viene annullato nella sottrazione

poiché è costante in E ed E' .

Con qualche semplice conto otteniamo che:

$$E' - E = -(x'_\ell - x_\ell) \sum_{i=1}^N w_{\ell i} x_i$$

Ma abbiamo dimostrato a inizio paragrafo che la quantità seguita dal meno è *sempre positiva*, ergo, con il meno davanti abbiamo una quantità *sempre negativa*. Pertanto, abbiamo dimostrato che:

$$E' - E < 0$$

è sempre vero.

8.4.3 Risultato del teorema

Non resta che mettere insieme i due fatti di cui disponiamo:

- Sappiamo che $E' - E < 0$, ovvero ogni cambiamento di stato porta sempre ad una diminuzione del livello di energia.
- Il numero di stati è finito.

Se ne deduce che l'algoritmo termina poiché l'energia non può diminuire in eterno. Inoltre, sappiamo anche che alla terminazione si avrà uno stato con minimo livello di energia, cioè uno stato stabile (che speriamo essere corrispondente ad una memoria associativa).

8.5 Dati sperimentali

Esaminiamo un caso studio per comprendere le capacità ed i difetti delle reti di Hopfield.

Abbiamo delle immagini 10×12 codificate pixel per pixel. Disponiamo perciò di 120 neuroni ($N = 120$) e di 14.280 pesi ($N^2 - N$). La quantità di memorie fondamentali che vogliamo memorizzare è 8 ($M = 8$). Un pixel nero è codificato con il valore +1, mentre un pixel bianco è codificato col valore -1.

In Figura 8.4 vediamo quali memorie fondamentali sono state adoperate nella fase di storage.

Esperimento in due fasi

L'esperimento si è svolto in due fasi.

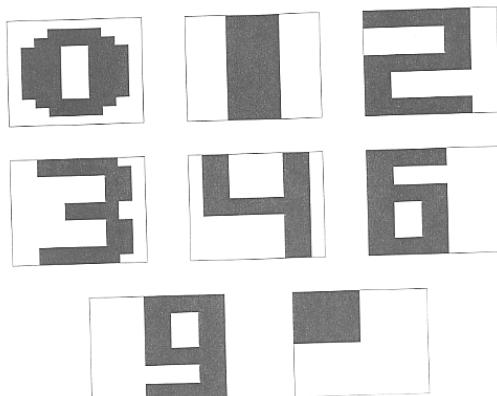


FIGURE 14.17 Set of handcrafted patterns for computer experiment on the Hopfield network.

Figura 8.4: Le immagini presentate alla rete per lo storage (memorie fondamentali).

La prima fase consiste nel presentare alla rete le memorie fondamentali e vedere se questa è in grado o meno di restituire la memoria fondamentale stessa. I risultati in questa fase sono stati strabilianti: il pattern desiderato è stato prodotto dalla rete in una sola iterazione.

La seconda fase, invece, prevede l'introduzione di una *probabilità di flip* P : viene presa una memoria fondamentale e ogni pixel dell'immagine viene flippato (inverto) con probabilità P . Nel nostro caso la probabilità di flip è stata settata al 25%.

I risultati sono stati accettabili: la rete si comporta come previsto e ci mette 31 iterazioni a trovare la memoria fondamentale associata. In Figura 8.5 vediamo un esempio di esecuzione sulla rete per il retrieval della memoria fondamentale "3".

8.5.1 I problemi delle reti di Hopfield

Nella maggior parte dei casi il mapping fra input e memoria fondamentale si è rivelato corretto. In alcuni casi, però, l'associazione si è rivelata erronea (Figura 8.6).

Un problema ancora più grave consiste nel fatto che alcuni input siano stati ricondotti a *memorie fondamentali non esistenti*. In Figura 8.7 vediamo 108 attrattori a cui sono stati associati input ottenuti con una probabilità di flip pari a 0.25. Sono stati effettuati 43097 test per ottenere tali attrattori. Questo fenomeno è spiegabile se si studia con attenzione ciò che viene fatto durante

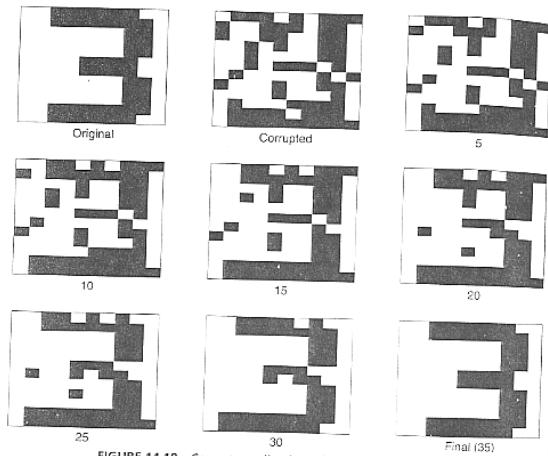


FIGURE 14.18 Correct recollection of corrupted pattern 3.

Figura 8.5: Il pattern 3 corrotto viene correttamente ricondotto alla sua memoria principale.

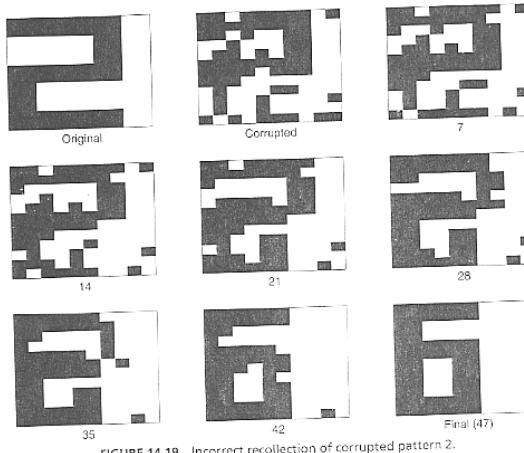


FIGURE 14.19 Incorrect recollection of corrupted pattern 2.

Figura 8.6: Il pattern 2 corrotto viene erroneamente ricondotto al pattern 6.

lo storage:

$$w_{ji} = \begin{cases} \frac{1}{M} \sum_{\mu=1}^M f_{\mu i} f_{\mu j} & \text{se } j \neq i, j = 1, \dots, N \\ 0 & \text{se } j = i \end{cases}$$

Il valore dei pesi viene ottenuto mediante moltiplicazioni e somme, ma esistono modi diversi di ottenere lo stesso valore mediante una somma o una moltiplicazione. Ecco quindi che tutte le varianti vengono considerate memorie fondamentali accettabili a cui ricondurre un input.

Ad esempio, la prima delle immagini in Figura 8.7 è l'inverso del 6, tale attrattore è ottenuto semplicemente invertendo $f_{\mu i} f_{\mu j}$ nel calcolo del peso: dato

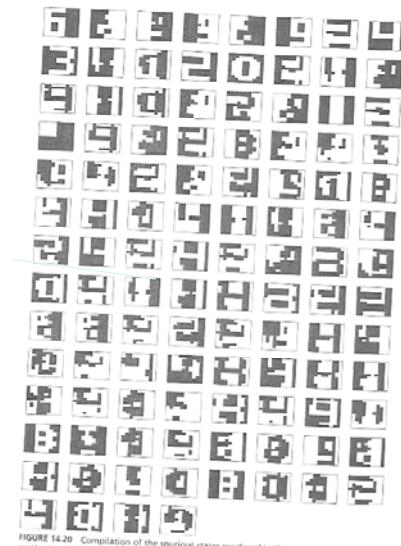


Figura 8.7: 108 attrattori (fra cui non vi sono le 8 memorie fondamentali) a cui sono stati associati gli input di 43097 test.

che la moltiplicazione gode della proprietà commutativa, il risultato è lo stesso rispetto alla memoria principale originale, ma ovviamente il senso cambia.

8.6 La capacità delle reti di Hopfield

Date le problematiche enunciate, è fondamentale dimensionare la rete di Hopfield in maniera adeguata alla quantità di memorie fondamentale che si vogliono storare. La capacità è definita come:

$$C = \frac{\text{numero di memorie principali}}{\text{numero di neuroni della rete}} = \frac{M}{N}$$

È teoricamente dimostrabile che l'upper bound di capacità è pari a:

$$C = \frac{1}{2 \ln N}$$

Dunque, avendo Max memorie fondamentali da storare e volendo un riconoscimento "accettabile" (in termini probabilistici) si avrà:

$$Max = \frac{N}{2 \ln N}$$

Capitolo 9

Boltzmann machines

Ci apprestiamo ora a studiare le Macchine di Boltzmann, particolari architetture di reti Neurali decisamente sofisticate: proprio a causa della loro complessità, le macchine di Boltzmann sono state sì teorizzate e realizzate, ma di fatto hanno avuto una minima applicazione.

Decisamente più significativo è invece il loro impiego nelle *Deep Neural Networks*, argomento del prossimo capitolo (e ragione per cui le introduciamo). Per ora ci basti sapere che le macchine di Boltzmann vengono utilizzate per inizializzare i pesi di una DNN.

Partendo quindi dal presupposto che stiamo per studiare qualcosa di decisamente complesso che verrà usato in una forma semplificata in un'altra architettura, affrontiamo l'argomento prima da un punto di vista prettamente teorico per poi portarlo nelle poche applicazioni che queste reti hanno avuto.

9.1 Introduzione

Come abbiamo appurato, le reti di Hopfield soffrono di un grave difetto: la discesa verso il livello di energia minimo è monotona, quindi non è *mai* possibile transire ad uno stato che abbia un livello maggiore di quello a cui ci si trova.

Questa limitazione ci impedisce di trovare il minimo globale: il fatto non va giù ai teorici Hinton & Sejnowski, che, a metà degli anni '80 creano la prima rete multilivello ispirata da meccaniche statistiche per risolvere il problema. L'idea di usare elementi statistici all'interno di una rete neurale risale però al 1954, quando Cragg e Temperly pubblicano i loro primi lavori sul tema.

Le macchine di Boltzmann cercano quindi di sfruttare le probabilità per permettere *in alcuni casi* di violare il principio di discesa monotona dell'energia e quindi, eventualmente, trovare il minimo globale. Però, come stiamo per im-

parare, il tutto è immerso in un algoritmo ancora più complesso la cui struttura è ispirata al processo fisico della *ricottura* (in inglese, *annealing*).

9.2 Studiare un sistema fisico

Dimentichiamo per ora che stiamo parlando di reti neurali, e parliamo di fisica.

Definiamo come *sistema fisico* la porzione di universo oggetto dell'indagine scientifica. All'interno di questo sistema saremo in grado di distinguere diverse variabili, e l'insieme delle combinazioni dei valori di dominio di tali variabili definirà l'insieme dei possibili stati in cui un determinato sistema fisico potrà trovarsi (un sistema fisico avrà tanti gradi di libertà quante sono le sue variabili).

Pensiamo ad esempio ad un sistema in cui si misuri l'energia, la velocità e la posizione: la combinazione dei vari valori di dominio delle tre variabili produrrà i vari stati in cui il sistema si può trovare.

In casi reali le variabili sono spesso non indipendenti fra loro e sono su un dominio continuo: noi, fortunatamente, ci limiteremo a variabili binarie e indipendenti.

Ciò che cercheremo di fare nei paragrafi a seguire è studiare il sistema al fine di trovare uno stato nel quale la sua quantità di energia sia minima.

9.2.1 Ad ogni stato una probabilità

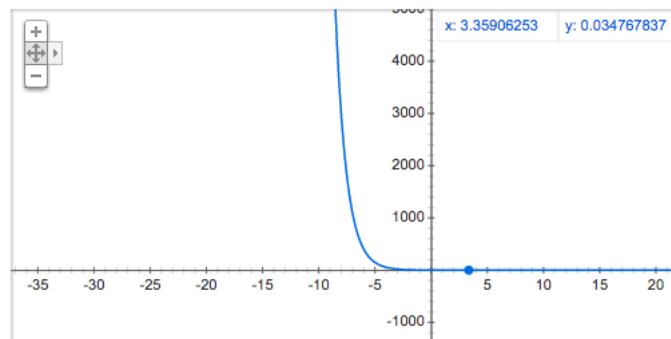
Iniziamo con l'associare ad ogni stato i una certa probabilità p_i di occorrenza. Ovviamente, trattandosi di probabilità, avremo che $p_i \geq 0 \quad \forall i$ e che $\sum_i p_i = 1$.

Ogni stato ha poi un valore di energia E_i che sfruttiamo per definire p_i . Ecco quindi l'equazione che ci dice qual è la probabilità di occorrenza di uno stato:

$$p_i = \frac{1}{Z} \exp\left(-\frac{E_i}{k_B T}\right) \quad (9.1)$$

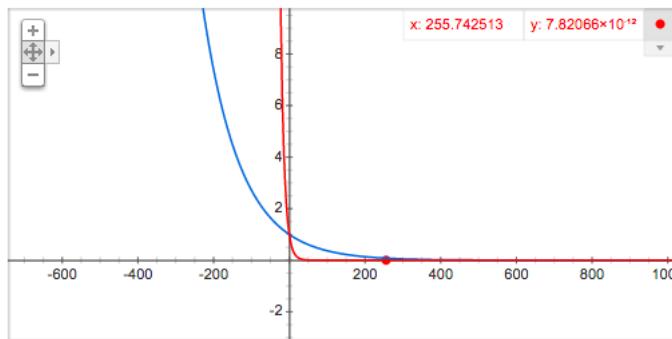
Il grosso della formula è costituito da un esponenziale decrescente (e^{-x}) e dunque la probabilità avrà un'andamento simile a quello riportato in Figura 9.1.

Il numeratore dell'esponente è l'energia relativa allo stato, mentre a denominatore abbiamo la costante di Boltzmann (k_B) e T , la temperatura del sistema. Nel campo delle reti neurali k_B viene impostato ad 1, mentre la temperatura T è una importante variabile il cui settaggio permette di inclinare la funzione di probabilità. Perciò, a bassi valori di energia è associata una alta probabilità, mentre ad alti valori di energia è associata una bassa probabilità. A parità di

Figura 9.1: Grafico di e^{-E} .

energia, però, questo concetto di “alta e bassa probabilità” può però essere modulato mediante T .

Più precisamente, se T è bassa, la curva sarà molto schiacciata ed i valori ad alta probabilità saranno pochissimi (e quindi solo quelli a bassissimo livello di energia avranno una probabilità alta); al contrario con una T alta la curva di probabilità sarà più distesa, e quindi anche stati ad energia relativamente alta avranno una probabilità associata abbastanza grande. Vediamo tutto questo in Figura 9.2 dove sono paragonate le funzioni $e^{\frac{-x}{10}}$ (blu) ed $e^{\frac{-x}{100}}$ (rosso).

Figura 9.2: Il valore di T (in questo caso 10 per la funzione rossa e 100 per la funzione blu) può inclinare la curva esponenziale.

Si può quindi associare al concetto di alta temperatura una maggiore “lassività” della funzione esponenziale, mentre quando le temperature sono basse la funzione esponenziale sarà spietatamente precisa e quindi si avranno valori di alta probabilità solo per stati ad energia molto bassa.

La variabile Z di cui non abbiamo ancora parlato non è altro che una variabile

di normalizzazione e vale:

$$Z = \sum_i \exp\left(-\frac{E_i}{k_b T}\right)$$

ovvero la sommatoria su tutti gli stati dei vari valori di probabilità. Dato che Z è a denominatore, è un po' come se avessimo reso la nostra p_i una classica formula casi favorevoli su casi totali.

9.2.2 Lo spazio degli stati

Consideriamo ora un sistema la cui evoluzione è descritta mediante un processo stocastico definito dalle variabili random X_n con $n = 1, 2, \dots$. Ogni variabile X_n assumerà quindi un valore x_n che definisce *lo stato del sistema* al tempo n (la definizione si riferisce ad un sistema con una variabile soltanto, ma se questa variabile fosse una sequenza di valori, avremmo considerato un sistema con molte variabili).

Lo spazio di tutti i possibili valori che le variabili random possono assumere è detto *spazio degli stati* del sistema.

Supponendo di avere N oggetti binari (tutti descritti in una X_n), lo spazio delle soluzioni sarà chiaramente 2^N (come avevamo già appreso dalle reti di Hopfield), ovvero esisteranno 2^N possibili configurazioni di X_n .

9.3 L'algoritmo di Simulated Annealing

Nell'algoritmo di *Simulated Annealing* (d'ora in poi SA) andremo a sfruttare le definizioni che abbiamo appena imparato per cercare di trovare il minimo di energia di un sistema fisico. Vediamo il singolo passo dell'algoritmo (che è a sua volta un algoritmo, l'algoritmo di Metropolis) e poi studiamone la struttura per intero.

Finalmente, dopo aver capito come funziona SA, vedremo come tale algoritmo ci potrà essere utile applicato ad una rete neurale.

9.3.1 Algoritmo di Metropolis

L'algoritmo di Metropolis è un algoritmo di Monte Carlo modificato che permette la simulazione dell'evoluzione stocastica di un sistema fisico fino al giungere del suo equilibrio termico.

Disponiamo di un sistema fisico ad una data temperatura T e vogliamo trovarne l'equilibrio termico (minimo di energia). Per fare ciò, partiamo da uno stato casuale i (per ora, nel SA non sarà così) e proviamo a transire in un altro

stato j .

Definiamo poi il fondamentale valore:

$$\Delta E = E_j - E_i$$

che denota la variazione di energia ottenuta compiendo una transizione dallo stato x_i ad uno stato x_j .

A seconda del segno di ΔE avremo due comportamenti diversi:

1. Se $\Delta E < 0$ (ovvero lo stato in cui siamo arrivati ha energia minore del precedente, cioè $E_j < E_i$), la transizione viene accettata e si procede con un nuovo tentativo.
2. Se $\Delta E > 0$ (ovvero lo stato in cui siamo arrivati ha energia maggiore del precedente, cioè $E_j > E_i$), la transizione può essere accettata, seguendo un ragionamento probabilistico.

La grande novità consta chiaramente nel secondo caso: nelle reti di Hopfield non era *mai* accettata (di fatto era impossibile) una transizione verso uno stato con livello di energia maggiore del precedente!

La scelta probabilistica viene regolata mediante l'uso di una formula che ricorda molto p_i (Equazione 9.1) come l'abbiamo definita in precedenza, ma si tratta di una formula diversa. Infatti, invece di usare E_i adoperiamo ΔE : questo fatto è importante, poiché non si considera in maniera assoluta il livello di energia, bensì quanto la transizione sia in grado di "sbalzare" il valore energetico del sistema. Di fatto, comunque, i ragionamenti sull'andamento della probabilità effettuati in precedenza rimangono immutati.

Da un punto di vista implementativo, si estrae un numero ε in maniera uniforme nel range $[0, 1]$, dopodiché se è vero che $\varepsilon < p_i$ (ricordiamo che si usa ΔE al posto di E_i) la transizione viene accettata, altrimenti, è rifiutata. L'impiego di questo ε non modifica i ragionamenti effettuati in precedenza, ma aggiunge un elemento probabilistico che in certi casi faciliterà la transizione (ε prossimo a 0) mentre in altri la complicherà (ε prossimo ad 1).

Quando una transizione è rifiutata se ne prova un'altra. I criteri di terminazione possono essere diversi, ne parleremo in dettaglio studiando come l'algoritmo di Metropolis venga impiegato all'interno della singola iterazione del SA.

Chiaramente questo algoritmo non garantisce di raggiungere il minimo effettivo (dipende dalla condizione di terminazione), e nemmeno di restituire il minimo incontrato (magari all'ultimo passo si sceglie di accettare una transizione "sfortunata" ed il livello di energia si alza).

9.3.2 Diminuire la temperatura in ogni iterazione

Vediamo ora come l'algoritmo di Metropolis venga impiegato dall'algoritmo di SA per trovare lo stato con il valore minimo globale di energia.

Definiamo prima di tutto una *serie di temperature* man mano decrescenti, i cui valori (in termini di quantità e distanza fra di essi) vengono definiti a monte. Dopodiché, per ogni valore di temperatura che abbiamo definito eseguiamo l'algoritmo di Metropolis: l'output di ogni algoritmo di Metropolis ci dirà lo stato di equilibrio termico ad una certa temperatura, e proprio quello stato fungerà da input per la successiva applicazione dell'algoritmo di Metropolis (che avverrà ad una temperatura più bassa).

Questo procedimento è del tutto simile al processo chimico/fisico di annealing (ecco da dove deriva il nome *Simulated Annealing*).

Il SA è un algoritmo di ottimizzazione che differisce dai suoi simili per due ragioni fondamentali:

- L'algoritmo non si blocca, dato che le transizioni intorno ad un minimo sono sempre possibili quando operiamo a temperature diverse da zero. Ovviamente, la probabilità di transire quando si è intorno ad un minimo è estremamente improbabile (il valore di T sarà molto piccolo quindi la curva sarà estremamente ripida, ovvero si accettano solamente minime transizioni "controproducenti").
- L'algoritmo è adattivo: con una temperatura alta si permette di esplorare maggiormente le soluzioni, ma con il diminuire della temperatura l'algoritmo si concentra sulla ricerca del minimo in maniera precisa (un comportamento che ricorda le SOM, che avevano una fase di auto-organizzazione ed una di convergenza).

Garanzie sulla convergenza al minimo

Dal punto di vista teorico abbiamo una bella garanzia: se le temperature scelte scendono *logaritmicamente*, allora abbiamo garanzia che SA trovi lo stato con il minimo globale di energia.

Purtroppo, la teoria ci fornisce un'informazione che non possiamo sfruttare: utilizzare un'algoritmo così costoso con temperature che si distanziano logaritmicamente fra loro è assolutamente impraticabile (tant'è che solitamente si usa una distribuzione esponenziale per definire le temperature).

9.3.3 Implementazione e criteri di stop

Per implementare SA necessitiamo quindi di una cosiddetta *coocking schedule*, cioè la serie di temperature su cui far lavorare l'algoritmo. Tale schedule

definirà quindi quante iterazioni verranno eseguite (nel senso di applicazioni dell'algoritmo di Metropolis).

Sempre nella coocking schedule sarà necessario definire un valore ξ di “minime transizioni accettate” da usarsi come limite per la terminazione di SA (vediamo dopo come).

Oltre che definire ξ si devono imporre dei criteri di stop per l'algoritmo di Metropolis: si può pensare che nelle prime iterazioni si imponga un limite alle transizioni effettuabili, e che nelle iterazioni a temperatura più bassa l'algoritmo si fermi quando si hanno ρ transizioni rifiutate di seguito.

Ecco quindi come si svolge l'algoritmo una volta che si dispone di una coocking schedule:

1. *Scelta del valore iniziale di temperatura*: tale valore dovrà essere scelto sufficientemente alto per permettere a tutte le possibili transizioni di essere “accettabili” per l'algoritmo almeno nella prima iterazione. Lo stato di partenza è definito randomicamente.
2. *Decremento della temperatura*: solitamente il raffreddamento viene effettuato esponenzialmente (seguendo la schedule). Una possibile funzione di decremento è:

$$T_k = \alpha T_{k-1}$$

dove α è un valore random compreso fra 0.8 e 0.99.

3. *Valore finale di temperatura*: il sistema viene fermato e l'annealing è completo quando per tre iterazioni successive (cioè tre successivi valori di temperatura) non si sono compiute almeno ξ transizioni. In altre parole, se per tre volte di seguito l'algoritmo di Metropolis non è riuscito a fare nemmeno ξ transizioni, SA termina. Si ponga attenzione a non confondere questo criterio di stop con quelli usati per il singolo algoritmo di Metropolis.

In alternativa, è possibile richiedere che il ratio di successo (transizioni accettate/transizioni provate) non sia mai sotto un certo valore predefinito (che sarebbe quindi alternativo a ξ).

9.4 Reti neurali stocastiche

Ora che abbiamo appreso il funzionamento dell'algoritmo di Simulated Annealing, vediamo come possiamo sfruttarlo nel nostro ambito.

Introduciamo innanzi tutto il concetto di rete neurale stocastica: una rete neurale stocastica è una particolare tipologia di rete neurale in cui o le funzioni

di transizione o i pesi sulle connessioni dei neuroni hanno un comportamento stocastico. Ad esempio, potremmo avere una funzione di attivazione che non fornisce un risultato deterministico a seconda dell'input, ma che darà un risposta anche legata ad una funzione di probabilità.

Vi sono principalmente due tipologie di reti neurali stocastiche:

- *Sigmoid Belief Nets* (Neal 1992) che sono costituite da neuroni binari stocastici connessi fra loro in grafo diretto aciclico.
- *Boltzmann machines* (Hinton & Sejnowski, 1983) che sono come le Sigmoid Belief Nets, ma le connessioni sono simmetriche (quindi il grafo non è diretto).

In Figura 9.3 vediamo le due reti a confronto.

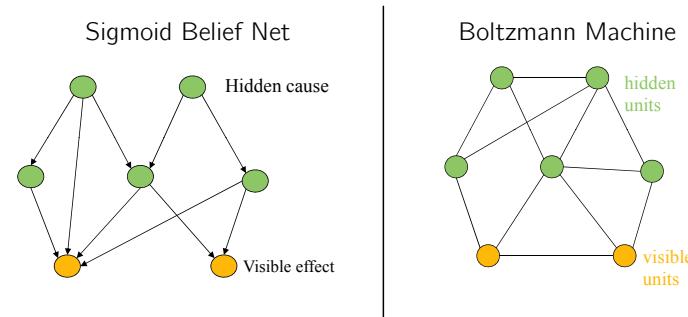


Figura 9.3: Le Sigmoid Belief Nets impongono un verso fra gli archi, mentre le macchine di Boltzmann no.

Noi ci concentreremo sulle Boltzmann machines.

9.5 Boltzmann machines

Le macchine di Boltzmann sono delle reti neurali che mirano a modellare pattern di input secondo una distribuzione di Boltzmann. In pratica, durante una prima fase si delineano i pesi sugli archi al fine di riconoscere determinati pattern (vedremo dopo come funziona questa fase: è abbastanza sofisticata) e, in fase di utilizzo, la rete dovrà riportare input simili a quelli imparati su stati simili (vedremo a fine capitolo cosa si intende).

Prima di entrare nel dettaglio del funzionamento di quest'architettura, ricordiamo che stiamo parlando di reti stocastiche in cui i neuroni sono neuroni stocastici binari (cioè che restituiscono +1 o -1 seguendo un comportamento probabilistico). I neuroni sono inoltre divisi in due gruppi, *visible* e *hidden* (ma di fatto fanno la stessa cosa) e le connessioni fra i neuroni sono simmetriche. In Figura 9.4 vediamo una macchina di Boltzmann.

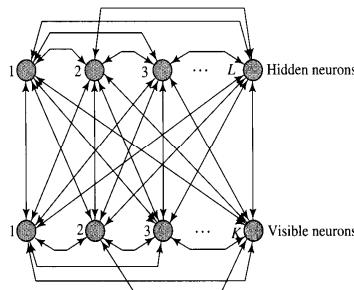


Figura 9.4: Schema dell'architettura di una rete di Boltzmann. K è il numero di neuroni visibili, L è il numero di neuroni nascosti.

Nelle sezioni a seguire andremo finalmente a studiare come vengono inizializzati i pesi mediante l'algoritmo di annealing e come viene poi utilizzata la macchina di Boltzmann per ottenere delle risposte in merito all'input datagli sui neuroni visibili.

9.5.1 Neuroni stocastici

Come detto poco fa, i neuroni di una macchina di Boltzmann sono stocastici e quindi il loro output è guidato dalla probabilità. Chiaramente, la probabilità sarà legata all'input del neurone stesso.

Più precisamente, la probabilità che un neurone assuma valore x_j (che può essere 1 o -1) è la seguente:

$$p(x = x_j) = \frac{1}{1 + \exp\left(\frac{-x_j}{T} \sum_{i \neq j} w_{ji} x_i\right)} \quad (9.2)$$

Come vediamo, a denominatore è presente la tipica sommatoria su tutti i valori di input (che chiamiamo v), che viene poi sfruttata al fine di ottenere la sigmoide.

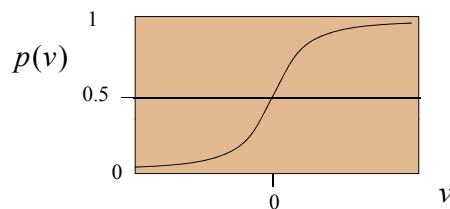


Figura 9.5: Andamento dell'output del neurone $p(v)$ rispetto all'input v . Tanto più è positivo l'input, tanto più alta sarà la probabilità di avere 1 come output; tanto più è negativo l'input e tanto più alta sarà la probabilità di avere -1 come output.

9.5.2 Il concetto di energia

Come già visto nelle reti di Hopfield, anche nelle macchine di Boltzmann il concetto di energia è fondamentale.

Definito che il vettore x denota lo stato di una macchina di Boltzmann (ogni componente x_i denota lo stato del neurone i , cioè il suo output) e che la connessione sinaptica da un neurone i ad un neurone j è definita come w_{ji} (ricordando che i pesi sono simmetrici e non esistono self-loop, ovvero $w_{ii} = 0$ per ogni i), allora abbiamo che l'energia associata ad uno stato x è definita come:

$$E(x) = -\frac{1}{2} \sum_i \sum_j w_{ji} x_i x_j$$

mentre la probabilità che la macchina di Boltzmann si trovi ad un determinato stato x è definito da:

$$P(X = x) = \frac{1}{Z} \exp\left(-\frac{E(x)}{T}\right)$$

con

$$Z = \sum_x \exp\left(-\frac{E(x)}{T}\right)$$

Per ora non abbiamo detto nulla di nuovo: abbiamo ripreso la formula dell'energia dalle reti di Hopfield e abbiamo ricordato la probabilità di trovarsi in un determinato stato per un sistema fisico (nel nostro caso, la macchina di Boltzmann).

9.5.3 L'algoritmo di apprendimento

Entriamo ora nel vivo della questione: come funziona l'apprendimento?

La likelihood

Il nostro intento è quello di massimizzare una funzione di *likelihood* (in italiano *verosimiglianza*). La funzione di verosimiglianza in statistica è una funzione di probabilità condizionata, considerata come funzione del suo secondo argomento, mantenendo fissato il primo argomento. In gergo colloquiale spesso “verosimiglianza” è usato come sinonimo di “probabilità”, ma in campo statistico vi è una distinzione tecnica precisa.

Questo esempio chiarisce la differenza tra i due concetti: una persona potrebbe chiedere “Se lanciassi una moneta non truccata 100 volte, qual è la probabilità che esca testa tutte le volte? ” oppure “Dato che ho lanciato una moneta 100 volte ed è uscita testa 100 volte, qual è la verosimiglianza (likelihood) che la moneta sia truccata? ”. Invertire nelle due frasi verosimiglianza e probabilità

sarebbe errato.

Noi siamo quindi interessati a calcolare la likelihood di un insieme di pesi che rappresenti al meglio l'insieme di training set (\mathfrak{T}). Se chiamiamo x_α l'insieme degli output sulle unità visibili regolato dalla variabile aleatoria X_α (quindi x_α è una porzione di x), avremo che \mathfrak{T} conterrà diverse configurazioni di x_α , ognuna delle quali rappresenta uno dei pattern che vogliamo apprendere.

Noi vogliamo però apprendere tutti i pattern contemporaneamente, dunque, la distribuzione totale di probabilità che utilizziamo è la distribuzione fattoriale:

$$\prod_{x_\alpha \in \mathfrak{T}} P(X_\alpha = x_\alpha)$$

cioè, semplicemente, per ogni x_α contenuto in \mathfrak{T} ne prendiamo la probabilità di occorrenza e poi le moltiplichiamo tutte insieme. In questo modo si ottiene la distribuzione totale di probabilità relativa all'apprendimento dell'insieme \mathfrak{T} .

Siamo riusciti ad ottenere una buona definizione di likelihood, poiché noi vogliamo massimizzare la probabilità che abbiamo appena calcolato (ovvero vogliamo che la distribuzione di probabilità sia appresa "al meglio"), perciò, possiamo dire:

$$L(w) = \prod_{x_\alpha \in \mathfrak{T}} P(X_\alpha = x_\alpha)$$

dove L è appunto la nostra funzione di likelihood.

Per ragioni di calcolo prendiamo però la log-likelihood (semplicemente il suo logaritmo): da un punto di vista teorico non cambia nulla, ma i calcoli ci saranno più facili.

Ecco quindi la definizione finale della nostra likelihood:

$$\log L(w) = \log \prod_{x_\alpha \in \mathfrak{T}} P(X_\alpha = x_\alpha) = \sum_{x_\alpha \in \mathfrak{T}} \log P(X_\alpha = x_\alpha)$$

Calcolare la likelihood

Abbiamo definito la likelihood ma ovviamente dobbiamo essere in grado di calcolarla al fine di massimizzarla, ovvero, dobbiamo calcolare $P(X_\alpha = x_\alpha)$ per ogni x_α del nostro training set. Noi sappiamo che la probabilità di un certo stato x di occorrere sulla macchina di Boltzmann è pari a:

$$P(X = x) = \frac{1}{Z} \exp\left(-\frac{E(x)}{T}\right)$$

ma x definisce al suo interno anche i valori di output per i neuroni hidden (chiamiamo questa porzione x_β), mentre x_α definisce solo i valori sulle unità

visibili. Allora, calcoliamo:

$$P(X_\alpha = x_\alpha) = \frac{1}{Z} \sum_{x_\beta} \exp\left(-\frac{E(x)}{T}\right)$$

ovvero, la probabilità di avere x_α sulle unità visibili è data dal fatto di avere x_α con una qualsiasi combinazione di x_β sulle unità hidden (ecco il perché della sommatoria sulle possibili x_β). Perciò, lo stato x utilizzato come parametro di E sarà sempre costituito da x_α nella parte che concerne le unità visibili, mentre avrà una combinazione diversa di x_β ad ogni loop della sommatoria.

In altre parole abbiamo sommato le probabilità $P(X = x)$ di tutti quegli stati x che hanno x_α sulla parte delle unità visibili e una qualsiasi altra combinazione su x_β .

Non resta che sostituire la definizione nella nostra formula di likelihood per ottenere:

$$\log L(w) = \sum_{x_\alpha \in \mathfrak{T}} \left(\log \frac{1}{Z} \sum_{x_\beta} \exp\left(-\frac{E(x)}{T}\right) \right)$$

Ricordando però che Z vale:

$$Z = \sum_x \exp\left(-\frac{E(x)}{T}\right)$$

otteniamo:

$$\log L(w) = \sum_{x_\alpha \in \mathfrak{T}} \left(\log \left(\frac{\sum_{x_\beta} \exp\left(-\frac{E(x)}{T}\right)}{\sum_x \exp\left(-\frac{E(x)}{T}\right)} \right) \right)$$

e per le proprietà dei logaritmi (il logartimo della divisione è la sottrazione fra logartimo del numeratore e il logartimo del denominatore) otteniamo:

$$\log L(w) = \sum_{x_\alpha \in \mathfrak{T}} \left(\log \left(\sum_{x_\beta} \exp\left(-\frac{E(x)}{T}\right) \right) - \log \left(\sum_x \exp\left(-\frac{E(x)}{T}\right) \right) \right) \quad (9.3)$$

Update dei pesi

Finalmente la nostra likelihood è calcolabile: non resta che "lavorarci sopra" al fine ottenere una qualche formula che permetta la modifica dei pesi (ovviamente tenendo conto che vogliamo massimizzare $L(w)$). Di fatto, vogliamo calcolare:

$$\Delta w_{ji} = \varepsilon \frac{\partial L(w)}{\partial w_{ji}}$$

ovvero, vogliamo calcolare come varia la likelihood al variare dei pesi e mediante questo valore effettuare la modifica dei pesi stessi (infatti calcoliamo Δw_{ji}). Il ragionamento è identico a quello che abbiamo fatto per minimizzare l'errore nelle reti back-propagation, ma in quel caso calcolavamo il variare dell'errore al variare dei pesi. Il parametro ε è un ulteriore parametro di tuning (non è lo stesso ε che abbiamo visto nell'algoritmo SA).

A questo punto dovremmo imperviarci nella complessa risoluzione della derivata di $L(w)$, cosa non semplice (dato che si tratta di una sommatoria con logaritmi all'interno). Risparmiamoci quindi la fatica ed enunciamo direttamente il risultato:

$$\Delta w_{ji} = \eta(\rho_{ji}^+ - \rho_{ji}^-)$$

dove η è il solito learning rate, definito come:

$$\eta = \frac{\varepsilon}{T} \quad (9.4)$$

e quindi comprende la temperatura T (ricordiamone il coinvolgimento nell'algoritmo di annealing!). I due ρ_{ji} sono invece definiti come segue:

$$\rho_{ji}^+ = \langle x_j x_i \rangle_{\text{clamped}} \quad \rho_{ji}^- = \langle x_j x_i \rangle_{\text{free}}$$

Per comprendere il significato delle due ρ dobbiamo tornare alla $L(w)$ (Equazione 9.3): come vediamo, abbiamo la sottrazione fra due logaritmi. Questi due termini ci portano ad ottenere rispettivamente ρ^+ e ρ^- .

Il primo termine è una sommatoria che esplora tutte le possibili combinazioni di x_β , ovvero le combinazioni di valori sui neuroni hidden mantenendo però fissi i neuroni visibili (a x_α). Il secondo termine, invece, non ha questa limitazione e la sommatoria coinvolge tutti i possibili stati calcolandone l'energia.

Possiamo così distinguere due diverse fasi: la prima fase, detta *clamped*, avrà valori fissi sulle unità visibili e valori variabili sulle unità hidden, mentre la seconda fase, detta *free*, avrà valori variabili sia sulle unità hidden che sulle unità visibili.

9.5.4 Studio dell'algoritmo di apprendimento

Ecco quindi ciò che accade: una volta impostata una temperatura (e quindi un η) si considera una istanza alla volta all'interno di \mathfrak{T} e su quell'istanza si tenta di raggiungere l'equilibrio termico (algoritmo di Metropolis). Tale operazione viene compiuta però due volte: la prima (fase clamped) mantenendo fissi i valori di x_α secondo quanto definito dall'istanza considerata in \mathfrak{T} , e la seconda (fase free) senza bloccare alcun valore. Si noti che ogni fase comincia usando gli stessi pesi: le due fasi non si influenzano fra loro (e comunque i pesi

verranno aggiornati solo dopo aver considerato tutte le istanze, come stiamo per imparare).

Una volta terminate le due fasi, avremo ottenuto due stati x_{clamped} e x_{free} della macchina di Boltzmann (uno per ogni fase). Li memorizziamo, e consideriamo il pattern successivo. Anche per questo pattern si hanno le due fasi in cui si cerca di ottenere l'equilibrio termico e quindi si otterranno altri due stati x_{clamped} e x_{free} . Procediamo fino a che non abbiamo esaminato tutti i pattern in \mathfrak{T} .

Al termine di questa fase, se \mathfrak{T} constava di n pattern, avremo n stati x_{clamped} e n stati x_{free} .

Siamo finalmente pronti ad aggiornare i pesi: per fare ciò, dobbiamo calcolare ρ^+ e ρ^- su ogni coppia di neuroni. Non è un problema, disponiamo di tutti i dati all'interno dei vari x_{clamped} e x_{free} .

Prima di tutto scegliamo il peso per cui vogliamo calcolare i ρ che poi impiegheremo nella formula:

$$\Delta w_{ji} = \eta(\rho_{ji}^+ - \rho_{ji}^-)$$

In sostanza, quindi, fissiamo un neurone j ed un neurone i .

Calcoliamo quindi ρ_{ji}^+ scandendo tutti i x_{clamped} e consideriamo solo le componenti x_j ed x_i relative al peso che ci interessa. Ricordiamo che ogni x_{clamped} mantiene l'intero stato della macchina di Boltzmann, cioè tutti i valori di output su tutti i neuroni, mentre noi siamo interessati solo all'output di i e di j . Ecco quindi che per ogni x_{clamped} calcoliamo la semplice moltiplicazione $x_j x_i$ e poi ne facciamo la media fra tutti i valori ottenuti (dividendo per n).

Calcoliamo poi ρ_{ji}^- nello stesso modo, ma usando gli stati x_{free} precedente memorizzati (anziché x_{clamped}).

Una volta calcolati i due ρ_{ji} saremo pronti ad aggiornare il peso della sinapsi fra il neurone i e j . Si dovrà ripetere l'operazione *per ogni peso*, così da avere l'intera macchina di Boltzmann con i pesi aggiornati.

A partire da questa macchina, si diminuirà la temperatura e si ripeterà da capo l'intero algoritmo implementando così il Simulated Annealing.

In Listing 9.1 vediamo una traccia testuale dell'intero algoritmo.

- ```

1 Si hanno ξ neuroni .
2 La matrice w mantiene tutti i pesi della macchina .
3 Si ha un ε .
4 Si ha una serie di temperature $T = \{T_1, \dots, T_m\}$
5 Si ha un training set $\mathfrak{T} = \{\mathfrak{T}_1, \dots, \mathfrak{T}_n\}$
```

```

6 Inizializza tutti i pesi casualmente.
7
8 /*
9 Strutture per mantere gli stati in equilibrio termico
10 (risultati di metropolis)
11 mantengono n array (uno per ogni pattern)
12 grandi ξ (una cella per ogni neurone)
13 */
14 xclamped[n][ξ];
15 xfree[n][ξ];
16
17 foreach($T_k \in T$) { //per ogni temperatura (indice k)
18 $\eta = \epsilon/T_k$;
19
20 foreach($\mathfrak{T}_u \in \mathfrak{T}$) // per ogni pattern (indice u)
21
22 // -- Fase clamped --
23 //equilibrio termico per pattern \mathfrak{T}_u e temperatura T_k
24 xclamped[u] = metropolis(T_k , \mathfrak{T}_u);
25
26 // -- Fase free --
27 //equilibrio termico per pattern \mathfrak{T}_u e temperatura T_k
28 xfree[u] = metropolis(T_k , \mathfrak{T}_u);
29
30 // -- Calcolo aggiornamento pesi --
31 foreach(neurone i) {
32 foreach(neurone $j \neq i$) {
33 //lavoro sul peso w_{ji}
34
35 // calcolo di ρ^+
36 $\rho^+ = 0$;
37 for($h = 1$; $h \leq n$; $h++$) //accumulo $x_j x_i$
38 $\rho^+ += x_{\text{clamped}}[h][i] * x_{\text{clamped}}[h][j]$;
39
40 $\rho^+ = \rho^+/n$; // media
41
42 // calcolo di ρ^-
43 $\rho^- = 0$;
44 for($h = 1$; $h \leq n$; $h++$) // accumulo $x_j x_i$
45 $\rho^- += x_{\text{free}}[h][i] * x_{\text{free}}[h][j]$;
46
47 $\rho^- = \rho^-/n$; // media
48
49 // uso i ρ per aggiornare il peso
50 w[j][i] = w[j][i] + $\eta * (\rho^+ - \rho^-)$

```

```

51 }
52 } // fine aggiornamento pesi
53 } //fine pattern
54 } //fine temperature

```

Listing 9.1: Pseudocodice per l'apprendimento di una Boltzmann machine. Si suppone che il metodo *metropolis* abbia a disposizione la struttura della macchina.

### 9.5.5 Algoritmo di Metropolis: come funziona la transizione

Nel precedente codice abbiamo supposto la presenza di una procedura chiamata *metropolis* che, dato un pattern ed una temperatura, è in grado di portare il “sistema fisico” (la macchina di Boltzmann) in equilibrio termico. Lo stato della macchina in equilibrio termico viene poi utilizzata per calcolare i  $\rho$  ed aggiustare i pesi. Abbiamo studiato nelle sezioni precedenti come funziona l’algoritmo di Metropolis in senso generico, ma vediamo ora come è implementata la transizione fra uno stato e l’altro all’interno di una macchina di Boltzmann.

**Transizioni nella fase clamped.** Come sappiamo, nella fase clamped vengono inseriti sui neuroni visibili le varie componenti di un pattern facente parte del training set. Non appena questi valori vengono messi sui neuroni visibili, i neuroni hidden “reagiscono” di conseguenza producendo uno stato che sia coerente con quanto posto sui neuroni visibili. Da questo momento, a causa del fatto che ogni neurone è collegato coi suoi fratelli, parte una serie di possibili transizioni (che corrispondono alle transizioni casuali dell’algoritmo di Metropolis). In sostanza ogni neurone propone una transizione di stato (ricordiamo che i neuroni sono stocastici, ergo l’output è non *deterministico*) la quale verrà rifiutata o accettata seguendo i parametri che abbiamo studiato nella Sezione 9.3.1. Ogni transizione genererà a sua volta delle modifiche negli altri neuroni hidden (dato che sono tutti collegati fra loro) i quali proporranno altre transizioni, che verranno sempre accettate o rifiutate. L’intero processo non coinvolge mai i neuroni visibili, che sono clamped e quindi non possono variare il loro stato. Si noti quindi che la transizione accade *neurone per neurone*.

Dopo un certo numero di transizioni o di rifiuti (ricordiamo le diverse possibilità per i criteri di stop indicate in Sezione 9.3.3) si raggiunge l’equilibrio termico.

**Transizioni nella fase free.** Nella fase free l’algoritmo si comporta in modo molto simile a quanto visto nella fase clamped, ma dopo aver piazzato il

pattern sui neuroni visibili ed aver suscitato una transizione nei neuroni hidden, anche i neuroni visibili possono modificare il loro stato proponendo quindi delle transizioni. Possibilmente, questa fase può durare di più rispetto alla precedente.

## 9.6 Usare le macchine di Boltzmann

Una volta terminato l'algoritmo di apprendimento tutti i pesi sulle sinapsi sono valorizzati e quindi la rete si può finalmente utilizzare. Supponiamo di avere un certo input che si vuole ricondurre ad uno dei pattern del training set: tale input viene piazzato sulle unità visibili, dopodiché si lascia "ragionare" la rete, ovvero la si lascia transire liberamente. Per fare ciò si ripete l'algoritmo di SA a tutte le temperature definite nella coocking schedule ma chiaramente non si modifica alcun peso. La macchina di Boltzmann passerà così da uno stato iniziale in cui l'input era fornito sui neuroni visibili e i neuroni hidden non erano valorizzati ad uno stato finale che avrà sui neuroni visibili uno dei pattern utilizzati in fase di learning (o comunque una configurazione molto simile, che ci riporterà ad uno dei pattern del training set).

## 9.7 Commenti sulle macchine di Boltzmann

È sufficiente dare uno sguardo all'algoritmo in Listing 9.1 per capire quanto sia sofisticato e costoso l'apprendimento di una macchina di Boltzmann. Ecco perché queste macchine non sono mai state utilizzate così come le abbiamo studiate (nella loro forma "pura") ma soltanto in versione ristretta nelle reti profonde.

La presenza di molti neuroni hidden può infatti comportare un'esecuzione molto lenta dell'algoritmo di metropolis, il quale viene eseguito due volte per ogni pattern ad ogni temperatura. La fase free è inoltre tendenzialmente più costosa a causa dei maggiori gradi di libertà dei neuroni (non dimentichiamo che tutti i neuroni sono collegati fra loro: avere bloccati tutti i neuroni visibili è un vantaggio!).

Vedremo come la versione ridotta delle macchine di Boltzmann permette di risolvere gran parte di questi problemi computazionali.



# Capitolo 10

## Deep Neural Networks

L'ultima tipologia di reti che andiamo a studiare sono le *Deep Neural Networks* (d'ora in poi DNN). Si tratta di particolari FNN a più livelli che durante la fase di learning non sfruttano l'algoritmo di back-propagation così come l'abbiamo studiato fin'ora, bensì impiegano particolari macchine di Boltzmann (dette macchine di Boltzmann ridotte) e la discesa del gradiente in maniera combinata.

### 10.1 Introduzione

#### 10.1.1 Perché DNN

Le reti back-propagation a due livelli che abbiamo studiato sono in grado di risolvere praticamente qualsiasi problema di interesse che si sia presentato fin'ora. Perché allora aumentare il numero di livelli?

In realtà non vi sono prove che utilizzare un maggiore numero di livelli fornisca qualche vantaggio sulla qualità del risultato riportato dalle reti, tranne nel caso di task particolarmente complessi il cui processing può essere suddiviso in livelli. Pensiamo ad esempio al riconoscimento di lettere: all'inizio potremmo dividere le lettere in grandi categorie (ad esempio quelle che hanno dei tondi e quelli che non ne hanno) e poi suddividere ulteriormente i vari insiemi per una categorizzazione più precisa (fino, presumibilmente, al riconoscimento del singolo carattere).

Un ulteriore motivo per cui reti a tre livelli o più possono interessanti riguarda la somiglianza rispetto al cervello umano: un processing man mano più raffinato (cioè livello per livello) dell'informazione sarebbe più vicino a ciò che ritengiamo faccia il la mente umana.

Abbiamo quindi che ogni livello contribuisce a formare una rappresentazione dell'entità in input: tale entità viene mano a mano classificata "sempre meglio", fino al livello di output dove ci si aspetta che l'input venga rappresentato

nella sua forma più semplice (e quindi immediatamente classificabile). Si ha in sostanza una transizione dal concetto nella sua rappresentazione astratta verso il concetto rappresentato in maniera più pratica (di basso livello).

Anche se ogni problema è risolvibile con una rete a due livelli potrebbe essere necessario impiegare moltissime hidden units: in una rete a più livelli è possibile utilizzare un quantitativo di hidden units più limitato poiché le unità a livelli avanzati lavorano su un input già raffinato e quindi è come se le hidden units venissero sfruttate in maniera più “intelligente”.

### L'inadeguatezza di back-propagation

Una volta appurato che può essere interessante avere reti a più di due livelli, perché non usare l'algoritmo back-propagation per l'apprendimento? La banale risposta è che *non funziona*. Non si è ancora trovata una precisa spiegazione matematica del perché l'algoritmo di back-propagation si comporti male con reti a tre o più livelli ma sta di fatto che la fase di correzione dei pesi risulta essere poco efficace e così, i pesi inizializzati in maniera randomica, non riescono a convergere verso una configurazione utile al riconoscimento degli input.

Come vedremo, il problema consta nella randomizzazione dei pesi a inizio algoritmo: almeno da un punto di vista intuitivo, inizializzare i pesi totalmente a caso non permette alla rete di computare correttamente la variazione di errore rispetto all'assegnamento dei pesi stessi e quindi la rete va fuori controllo anziché raggiungere un errore minimo.

Pertanto, dato che nelle DNN non è possibile utilizzare direttamente l'algoritmo di back-propagation con pesi randomici, si è deciso di fare un *preprocessing dei pesi* mediante l'impiego di macchine di Boltzmann ridotte. Questa fase di preprocessing dei pesi permette di trovare una inizializzazione sufficientemente sensata da far sì che l'algoritmo di back-propagation non impazzisca ma si comporti in maniera adeguata.

#### 10.1.2 L'architettura

Introduciamo l'architettura delle DNN e un po' di notazione. In Figura 10.1 vediamo la struttura di una DNN.

In sostanza abbiamo un primo livello (più in basso) che è il nostro solito input  $x$ . Seguono  $\ell$  livelli hidden (in figura ve ne sono 3) che sono collegati mediante i soliti pesi come abbiamo già studiato per le reti a due livelli. Ogni livello dispone anche del bias. Infine, abbiamo l'ultimo livello che espone l'output della rete.

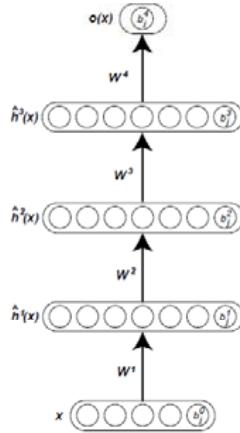


Figura 10.1: Struttura di una DNN.

Indichiamo l'output dell' $j$ -esimo neurone a livello  $i$  come:

$$h_j^i(x) = \text{sigmoid}(a_j^i)$$

dove  $a$  è definito come al solito, ovvero come somma pesata dell'input del neurone:

$$a_j^i = b_j^i + \sum_k W_{jk}^i h_k^{i-1}(x)$$

Avendo  $\ell$  livelli nascosti,  $i$  andrà da 1 ad  $\ell$ . Distinguiamo poi il livello di input:

$$h^0(x) = x$$

e quello di output:

$$h^{\ell+1}(x) = o(x)$$

In questo caso abbiamo utilizzato una funzione sigmoide su ogni neurone, ma volendo è possibile utilizzarne altre (e magari utilizzare sull'ultimo livello una semplice combinazione lineare, come nelle RBFN).

### 10.1.3 L'apprendimento: idea introduttiva

Approfondiamo il funzionamento dell'algoritmo di apprendimento (vedremo solo alla fine la sua effettiva implementazione).

Per allenare le DNN si fa uso di un apprendimento ibrido, ovvero di un apprendimento che consta sia di una fase supervisionata che di una parte non supervisionata.

In pratica abbiamo:

1. Fase di pre-training dei pesi: ogni coppia di livelli della rete viene considerata come una macchina di Boltzmann (teoricamente ridotta, ma per ora possiamo pensare di implementarne una completa). Durante questa fase si effettua quindi un calcolo greedy dei pesi mediante un approccio non supervisionato. I livelli vengono presi da quello di input verso quello di output a due a due, dunque, ogni macchina di Boltzmann fornisce il livello visibile per quella successiva (il primo livello visibile è dato dall'input). In altre parole, i primi due livelli della DNN costituiscono la prima macchina di Boltzmann. Al termine dell'apprendimento di questa prima macchina (che fa scorrere sui suoi neuroni visibili i pattern del training set) tutti i pesi che collegano il primo livello col secondo saranno valorizzati. Per l'apprendimento della seconda macchina di Boltzmann (costituita dal secondo e dal terzo livello della DNN) si piazzerà un pattern di input sul primo livello della DNN, poi i pesi appresi dalla prima macchina di Boltzmann lo processeranno e faranno produrre un risultato sul secondo livello della DNN (che è il livello visibile della seconda macchina di Boltzmann). Sarà quindi questo pattern "processato" a fungere da input per la seconda macchina di Boltzmann. La terza macchina di Boltzmann, conseguentemente, vedrà sui suoi neuroni visibili ogni pattern di input processato dai pesi del primo-secondo e secondo-terzo livello della DNN.
2. Fase di fine-tune: utilizzando una classica back-propagation i pesi vengono "rifiniti". Chiaramente questa fase è supervisionata.

Alcuni esperimenti suggeriscono che questo tipo di inizializzazione migliori il classico assegnamento random, permettendo alla fase di finale di partire da una zona già vicina ad un minimo locale e quindi garantendone il funzionamento.

In merito alla generalità della soluzione utilizzare un apprendimento di questo tipo sembra essere vantaggioso. Quando abbiamo parlato di regolarizzazione (in ELM) abbiamo già spiegato come ottenere una soluzione generale sia alla fine più importante che ottenere una soluzione perfettamente fittante il training set. In questo caso il learning non supervisionato fa qualcosa di molto simile ma in maniera ancora più interessante rispetto alla regolarizzazione. In quel caso, infatti, il parametro  $\lambda$  viene scelto "a priori", indipendentemente dal dataset, mentre ciò che facciamo con le macchine di Boltzmann è ovviamente dipendente dal dataset e quindi è più fine/preciso.

Riassumendo, il nostro apprendimento consta in una prima fase greedy per inizializzare i pesi affinché le unità hidden abbiano una sorta di "guida" su che cosa apprendere. La fase supervisionata che segue permette poi la minimizzazione di una funzione di costo globale (l'errore, ad esempio). In Figura 10.2 abbiamo uno sketch grafico che illustra l'apprendimento.

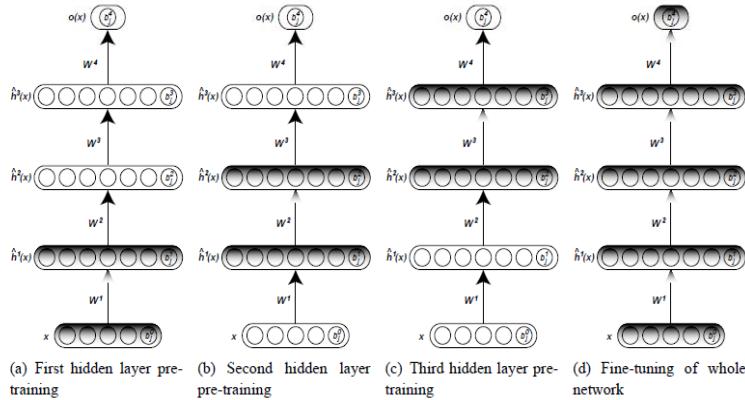


Figura 10.2: L'apprendimento nelle DNN.

## 10.2 Restricted Boltzmann machines

Per comprendere come è implementato l'apprendimento dobbiamo introdurre le *Restricted Boltzmann machines* (RBM), che vengono impiegate nella prima fase di learning non supervisionato.

Come abbiamo studiato, le macchine di Boltzmann sono praticamente inutilizzabili a causa dell'enormità del costo dell'apprendimento. È evidente che farne un uso così massiccio solo per inizializzare dei pesi risulti ulteriormente impraticabile. Ecco perché si sfruttano le RBM, che hanno una *connettività più semplice*.

Nelle macchine di Boltzmann ristrette i neuroni sono connessi solo fra livelli e non intra-livello.

Questo fatto ha una conseguenza immediata sull'algoritmo di apprendimento nella fase clamped: una volta fissato un valore sulle unità visibili si ha una modifica delle unità hidden e *basta*. La dinamica è immediatamente ferma. In una macchina di Boltzmann completa, invece, l'interazione fra le unità hidden avrebbe richiesto ulteriori step dell'algoritmo di Metropolis per raggiungere l'equilibrio termico. Come stiamo per imparare, questa conseguenza avrà un impatto importantissimo nell'algoritmo di apprendimento.

### 10.2.1 Pesi, energia e probabilità

#### L'energia

Riprendiamo il caro vecchio concetto di energia che abbiamo introdotto con le macchine di Boltzmann complete. Riscriviamo però la nostra definizione di energia in funzione di  $v$  ed  $h$ , cioè rispettivamente la configurazione sui neuroni

visibili e hidden. In pratica passiamo da questa definizione:

$$E(x) = -\frac{1}{2} \sum_i \sum_j w_{ji} x_i x_j$$

a questa definizione:

$$E(v, h) = -\sum_i \sum_j w_{ji} v_i h_j$$

Dove  $v_i$  rappresenta lo stato del neurone binario visibile  $i$  e  $h_j$  rappresenta lo stato del neurone binario  $j$ .

Abbiamo quindi scorporato  $x$ , calcolando l'energia di uno stato considerando la correlazione fra tutti i neuroni visibili (indice  $i$ ) e tutti i neuroni hidden (indice  $j$ ). Si noti che questa definizione ha senso grazie al fatto che essendo una macchina di Boltzmann ristretta *non esistono più* archi fra neuroni hidden e fra neuroni visibili, ergo non può esistere una sinapsi fra i neuroni il cui stato è descritto in  $v$  e fra quelli il cui stato è descritto in  $h$ .

### La probabilità

Ogni possibile configurazione di  $h$  e  $v$  ha quindi un'energia correlata, calcolata come sopra. L'energia di una configurazione determina anche la probabilità di occorrenza di quella configurazione, secondo la legge:

$$P(v, h) \propto \exp(-E(v, h))$$

Ancora una volta non stiamo dicendo niente di nuovo. Nelle macchine di Boltzmann complete avevamo affermato:

$$P(X = x) = \frac{1}{Z} \exp\left(-\frac{E(x)}{T}\right)$$

Possiamo definire con più precisione la probabilità di occorrenza di un determinato stato.

Se si tratta di uno stato nella fase clamped, avremo:

$$P(v, h) = \frac{\sum_h \exp(-E(v, h))}{\sum_{u,g} \exp(-E(v, h))}$$

invece nella fase free avremo:

$$P(v, h) = \frac{\exp(-E(v, h))}{\sum_{u,g} \exp(-E(v, h))}$$

Il denominatore corrisponde alla nostra vecchia  $Z$  (fattore di normalizzazione), che infatti ha una sommatoria su tutte le unità (cioè su tutti i possibili stati). Poi, nella fase clamped abbiamo a numeratore tutte le combinazioni di unità hidden sommate (essendo una probabilità composta) mentre nella fase free abbiamo una precisa configurazione di  $h$  (e quindi non c'è la sommatoria). Il ragionamento da tenere a mente è quello che abbiamo fatto per la likelihood delle macchine di Boltzmann complete.

## 10.3 L'algoritmo di apprendimento

Andiamo finalmente a vedere nel dettaglio come funziona l'algoritmo di apprendimento.

Per ora sappiamo che tale algoritmo è costituito di due grandi fasi:

1. Preprocessing dei pesi della rete mediante apprendimento non supervisionato, sfruttando macchine di Boltzmann ridotte costruite sui vari livelli della DNN presi a coppie.
2. Rifinitura dei pesi mediante apprendimento supervisionato utilizzando l'algoritmo di back-propagation sull'intera DNN.

La seconda fase non è particolarmente interessante, si tratta della solita discesa del gradiente per minimizzare l'errore come l'abbiamo studiata per le reti multilivello.

Concentriamoci invece sulla prima fase, l'impiego di macchine di Boltzmann ridotte sui vari livelli della DNN. Ragioniamo ora sull'apprendimento di una singola macchina di Boltzmann ristretta, ricordando che tale processo viene poi ripetuto per ogni coppia di livelli della DNN.

L'apprendimento di una macchina di Boltzmann si divide a sua volta in due parti: fase clamed e fase free. Studiamole separatamente.

### 10.3.1 Fase clamped su RBM

Lo abbiamo già accennato: giacché le RBM hanno una connettività ridotta, la fase clamped dell'algoritmo di apprendimento è pressoché immediata: si pongono i valori sulle unità visibili e le unità hidden produrranno uno stato adeguato sfruttando i pesi delle sinapsi. Non esiste ulteriore dinamica, poiché l'assunzione di uno stato da parte delle unità hidden non influenza in alcun modo le altre unità hidden (né tantomeno le unità visibili, che sono clamped).

Pertanto, la fase clamped produce uno stato  $x_{\text{clamped}}$  in una sola iterazione.

La definizione di  $\rho^+$  dunque non cambia:

$$\rho_{ji}^+ = \langle v_i h_j \rangle_{\text{clamped}}$$

dovremo quindi ripetere il processo per tutte le istanze del training set (eventualmente processate dai livelli precedenti della DNN se la macchina di Boltzmann che stiamo considerando non è la prima) e poi ottenere la media dei valori fra le varie moltiplicazioni  $v_i h_j$ .

In definitiva, la grande semplificazione consta nel fatto che l'equilibrio termico è raggiunto *in una sola iterazione* grazie alla connettività ridotta delle RBM rispetto alle macchine di Boltzmann complete.

### 10.3.2 Fase free su RBM

Il vantaggio ottenuto dalla connettività ridotta nella fase clamped non porta ad un'altrettanta allettante semplificazione nella fase free.

Infatti, una volta piazzati i valori sulle unità visibili, le unità hidden assumeranno dei valori adeguati, i quali saranno però in grado di modificare nuovamente i valori sulle unità visibili, fatto che genererà ulteriori transizioni nei neuroni hidden, e così via.

Risultati empirici ci vengono però incontro: a quanto pare, non risulta particolarmente efficace permettere più di quattro transizioni totali nella fase free. Ecco quindi che anche la fase free risulta incredibilmente semplificata e ridotta in quattro passi:

1. Il pattern del training set viene messo sulle unità visibili.
2. Le hidden units vengono aggiornate in parallelo conseguentemente a quanto presente sulle unità visibili.
3. Le unità visibili vengono aggiornate in parallelo conseguentemente a quanto presente sulle unità hidden (questo passo è detto *reconstruction*).
4. Le unità hidden vengono nuovamente aggiornate in parallelo conseguentemente a quanto presente sulle unità visibili.

Perciò, sebbene l'algoritmo originale preveda una serie di aggiornamenti come quelli sopra riportati da perpetrarsi fino all'effettivo equilibrio termico, noi ci limitiamo a quattro passi.

La semplificazione apportata non segue la discesa logaritmica della likelihood, ma funziona comunque bene (non dimentichiamoci che si tratta di una fase di preprocessing dei pesi!).

Anche in questo caso la definizione di  $\rho^-$  non cambia:

$$\rho_{ji}^- = \langle v_i h_j \rangle_{\text{free}}$$

ma ciò che cambia è il tempo (in termini di transizioni) necessario per completare l'algoritmo di Metropolis.

### 10.3.3 L'update dei pesi

Ciò che abbiamo ottenuto è in pratica una grande semplificazione dell'algoritmo di Metropolis, che quando applicato nella fase clamped consta di una sola transizione e quando applicato nella fase free consta di quattro transizioni (teoricamente tre).

Se chiamiamo  $\langle v_i h_j \rangle^0$  e  $\langle v_i h_j \rangle^1$  la media degli stati sui neuroni  $i$  e  $j$  risultanti dall'applicazione dell'algoritmo di Metropolis sui vari pattern del training set rispettivamente nella fase clamped e nella fase free, possiamo scrivere la nostra formula di aggiornamento dei pesi come segue:

$$\Delta w_{ji} = \eta(\langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^1)$$

In definitiva, la miglioria computazionale consta nel fatto che i due  $\rho$  (ora chiamati  $\langle v_i h_j \rangle^0$  e  $\langle v_i h_j \rangle^1$ ) vengono calcolati molto più velocemente, poiché ogni stato in equilibrio termico da utilizzarsi per il calcolo della media è calcolato molto più velocemente, sia nella fase clamped che nella fase free.

## 10.4 Prestazioni delle DNN (MNIST dataset)

Concludiamo il nostro excursus sulle DNN valutandone le prestazioni.

È evidente che strutture di questo tipo siano assai più complesse di una tipica rete a due livelli, dunque sorge spontanea la domanda: "ne vale veramente la pena"? Per rispondere a questa giusta obiezione dobbiamo fare un po' di esperimenti.

Ci baseremo sui risultati riportati nel paper che ha introdotto le DNN.

### 10.4.1 Il MNIST dataset

Parliamo prima di tutto del problema che vogliamo risolvere e dunque del dataset correlato.

Siamo nell'ambito del riconoscimento di caratteri e il nostro intento è quello di categorizzare cifre scritte a mano in una delle possibili 10 cifre (da 0 a 9). In Figura 10.3 vediamo una porzione del dataset.

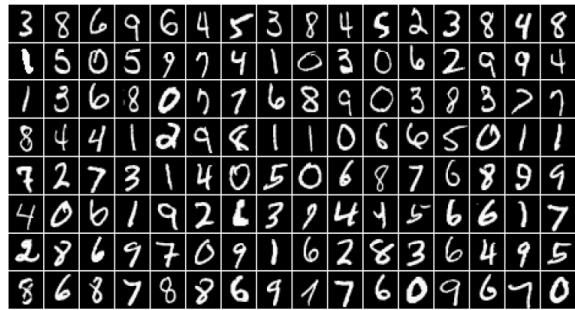


Figura 10.3: Esempi dal dataset MNIST. Con un valore a 0 indichiamo un pixel nero, mentre con un valore ad 1 indichiamo un pixel bianco.

#### 10.4.2 L'architettura della DNN

I raffronti fatti da Yann LeCun coinvolgono una DNN con tre livelli nascosti, rispettivamente di 500, 500 e 2000 unità nascoste. Poi, giacché ogni immagine da processare è  $28 \times 28$  pixel, abbiamo un livello di input costituito da 784 unità. Il livello di output consta invece di 10 unità, una per ogni carattere che vogliamo categorizzare (ci aspettiamo quindi una sola unità di output che risponda ad 1).

Il training è stato effettuato su 50,000 istanze, la validazione<sup>1</sup> su 10,000 e il testing su altre 10,000.

La funzione di attivazione usata è la tangente iperbolica, anziché la sigmoide.

#### 10.4.3 Diversi modelli a confronto

Sono principalmente due le domande a cui vogliamo trovare risposta:

- Inizializzare i pesi con un approccio greedy è utile?
- Quanto è importante il fatto che il preprocessing dei pesi sia non supervisionato?

Per rispondere a queste domande dobbiamo avere dei modelli che portino dei risultati affinché si possano studiare le differenze prestazionali. Yann LeCun compara i risultati di 6 modelli, noi ci concentriamo su quattro di questi:

- *SRBM (stacked restricted Boltzmann machines network)*: si tratta delle DNN che abbiamo studiato. Il preprocessing dei pesi è fatto mediante macchine di Boltzmann ristrette, seguito da un affinamento globale con back-propagation.

---

<sup>1</sup>Ricordiamo che il set di validazione è utile per trovare una buona definizione delle variabili di tuning, senza far sì che queste vengano modellate sull'insieme di testing.

- *Deep network with supervised pre-training*: il preprocessing dei pesi viene effettuato con una tecnica supervisionata (non le macchine di Boltzmann). Per implementare tale pre-training viene utilizzato l'algoritmo di back-propagation su ogni coppia di livelli della DNN (in pratica si usa un terzo livello di output fittizio per ogni coppia di livelli e lì viene piazzato il risultato atteso dalla rete, al fine di implementare la back-propagation).
- *Deep network, no pre-training*: una DNN come quella studiata, ma senza alcun tipo di preprocessing dei pesi. I pesi sono quindi random (è il caso "impraticabile" di cui abbiamo parlato a inizio capitolo).
- *Shallow network, no pre-training*: una rete multilivello con soli due livelli (quindi non una DNN). Tale rete è utile per fare riflessioni in merito a quanto la complessità delle DNN possa portare beneficio o meno.

In tutte le reti la funzione di attivazione usata è la tangente iperbolica.

#### 10.4.4 Risultati (hidden units arbitrarie)

In Figura 10.4 vediamo i primi risultati. Non è stata posta alcuna limitazione alle hidden units utilizzabili.

| Models                                               | Train. | Valid. | Test  |
|------------------------------------------------------|--------|--------|-------|
| SRBM (stacked restricted Boltzmann machines) network | 0%     | 1.20%  | 1.20% |
| SAA (stacked autoassociators) network                | 0%     | 1.31%  | 1.41% |
| Stacked logistic autoregressions network             | 0%     | 1.65%  | 1.85% |
| Deep network with supervised pre-training            | 0%     | 1.74%  | 2.04% |
| Deep network, no pre-training                        | 0.004% | 2.07%  | 2.40% |
| Shallow network, no pre-training                     | 0%     | 1.91%  | 1.93% |

Figura 10.4: Errore di classificazione sul training, validazione e testing sul MNIST dataset.

I risultati sono interessanti. Innanzitutto vediamo che su ogni modello il training ha avuto successo totale, questo indica il fatto che la rete è in grado di "assorbire" una complessità come quella del dominio MNIST.

Confrontando poi le prestazioni della shallow network rispetto alla SRBM vediamo che non c'è una "grande" miglioria in senso assoluto, ma è pur vero che per certe applicazioni migliorare la precisione anche di poco può essere importante.

Non è smentito ciò che abbiamo affermato a inizio capitolo: non effettuare pre-training su una rete DNN porta ad un errore doppio rispetto alla stessa rete con preprocessing dei pesi con macchine di Boltzmann ridotte.

Troviamo quindi risposta alle nostre due domande:

- Il preprocessing dei pesi è fondamentale.

- Confrontando la prima e la quarta riga possiamo affermare che un'approccio supervisionato in fase di pre-training porta a risultati più scadenti rispetto ad uno non supervisionato poiché si perde in capacità di generalizzazione.

#### 10.4.5 Risultati (hidden units limitate)

Cosa succede se limitiamo le hidden units dell'ultimo livello? Vediamo i risultati in figura 10.5.

| Models                                    | Train. | Valid. | Test  |
|-------------------------------------------|--------|--------|-------|
| SRBM network                              | 0%     | 1.5%   | 1.5%  |
| SAA network                               | 0%     | 1.38%  | 1.65% |
| Deep network with supervised pre-training | 0%     | 1.77%  | 1.89% |
| Deep network, no pre-training             | 0.59%  | 2.10%  | 2.20% |
| Shallow network, no pre-training          | 3.6%   | 4.77%  | 5.00% |

Figura 10.5: Errore di classificazione sul training, validazione e testing sul MNIST dataset. Le hidden units dell'ultimo livello nascosto sono limitate a 20.

Limitando a 20 il numero di hidden units dell'ultimo livello, la shallow network ne risente moltissimo: la complessità del problema non è più s intentizzabile in 20 unità soltanto. Questo prova il fatto che avendo una rete con più livelli l'informazione viene trattata e scremata livello per livello e dunque l'ultimo livello, sebbene molto ridotto, riesce comunque a dare prestazioni buone poiché lavora su un'informazione già semplificata.

# Bibliografia

- [1] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.