Appunti di Reti Neurali & Deep Learning

Davide Giosa[1][2]

2019/2020

[1]Basati sulle lezioni della Prof.ssa Valentina Gliozzi e della Prof.ssa Rossella Cancelliere

[2]Con la collaborazione di Zhongli Filippo Hu

Indice

1 Introduzione alle Reti Neurali **3**
1.1 Principi base delle Reti Neurali 3
1.2 Funzionamento di base 3

2 Perceptron **5**
2.1 Problemi linearmente separabili 5
2.2 Struttura del Perceptron 6
2.2.1 Il problema dell'OR 7
2.3 Algoritmo di apprendimento del Perceptron 8
2.4 Teorema di convergenza 10
2.5 Limiti del Perceptron 12

3 Multilayer Perceptron **13**
3.1 Adaline 13
3.1.1 Discesa Incrementale del Gradiente 14
3.1.2 Regola di aggiornamento dei pesi 14
3.2 Multi Layer Feed-Forward Neural Network 15
3.2.1 L'algoritmo di Backpropagation 16
3.3 Design di una Rete Neurale 20
3.4 Rete Neurale come approssimatore di funzioni 21

4 Reti Neurali Radiali **22**
4.1 Architettura della Rete 22
4.2 Modello del Neurone Hidden 23
4.3 Il Problema dell'Interpolazione 24
4.4 Il problema dello XOR 25
4.5 Algoritmi di Apprendimento 26
4.5.1 Algoritmo di Apprendimento 1 - Matrice Pseudo-Inversa . 26
4.5.2 Algoritmo di Apprendimento 2 - Calcolo dei Centri 27
4.5.3 Algoritmo di Apprendimento 3 - Discesa del Gradiente . . 28
4.6 Confronto con le Reti FFNN 29

5 Extreme Learning Machines **30**
5.1 Features . . . 30
5.2 Teoremi . . . 31
5.3 Algoritmo di Apprendimento . . . 32

6 Reti Neurali Convoluzionali **34**
6.1 Introduzione alle Reti Profonde . . . 34
6.2 Architettura della Rete . . . 35
6.3 Livello Convolutivo . . . 36
6.3.1 Il Filtro e lo Stride . . . 37
6.4 Convoluzione 3D . . . 38
6.4.1 SoftMax e Cross-Entropy . . . 39
6.5 Training e Transfer Learning . . . 40

7 Self-Organizing Maps **42**
7.1 Funzionamento della rete . . . 42
7.2 Struttura della Rete . . . 43
7.3 Fase di Learning . . . 44
7.3.1 Calcolare il Best Matching Unit . . . 44
7.3.2 Aggiornamento dei pesi . . . 44
7.3.3 Regola di apprendimento . . . 45

8 Reti di Hopfield **46**
8.1 Il Pattern Completion . . . 46
8.2 McCulloch-Pitts . . . 47
8.3 Fase di Storage . . . 49
8.4 Teorema di Convergenza . . . 50
8.5 Considerazioni finali . . . 52
8.5.1 I pro . . . 52
8.5.2 I contro . . . 52

9 Reti di Boltzmann **53**
9.1 Dalle Reti di Hopfield . . . 53
9.2 Architettura della Rete . . . 54
9.3 Funzionamento della Rete . . . 54
9.4 Fase di Learning . . . 56
9.4.1 Restricted Boltzman Machine . . . 56
9.4.2 La rete in azione . . . 57

Capitolo 1

Introduzione alle Reti Neurali

1.1 Principi base delle Reti Neurali

Si possono vedere le Reti Neurali come delle strutture che simulano il comportamento del cervello umano. All'interno del cervello umano, l'unità base di computazione è il **neurone**. I neuroni, tra di loro, sono collegati attraverso le **sinapsi**. Questa struttura è riportata anche all'interno delle Reti Neurali. Sempre facendo un analogia con il cervello umano, possiamo dire che quest'ultimo *raccoglie*, *elabora* e *propaga* diversi segnali lungo il cervello. Il segnale propagato è proporzionale all'input ricevuto. Anche questo comportamento viene riproposto all'interno delle nostre Reti Neurali.

1.2 Funzionamento di base

In Figura 1.1 è presente uno schema abbastanza basilare di quella che può essere una Rete Neurale. Da questo schema si possono cogliere alcuni elementi importanti che fanno parte di questa struttura.

In primo luogo, si possono notare una serie di neuroni che sono all'ingresso della rete, questi vengono chiamati **neuroni di input**, che si occupano di raccogliere l'informazione da elaborare all'interno della rete. Dopo, possiamo vedere un neurone detto **neurone di output** che si occupa di restituire il risultato della computazione della rete stessa. L'elaborazione avviene, in questo caso, attraverso la somma pesata degli input. Per ogni segnale di input, è associato un peso w che il punto fondamentale delle Reti Neurali, in quanto è proprio attraverso questi pesi che possiamo restituire un risultato piuttosto che un altro. Inoltre,

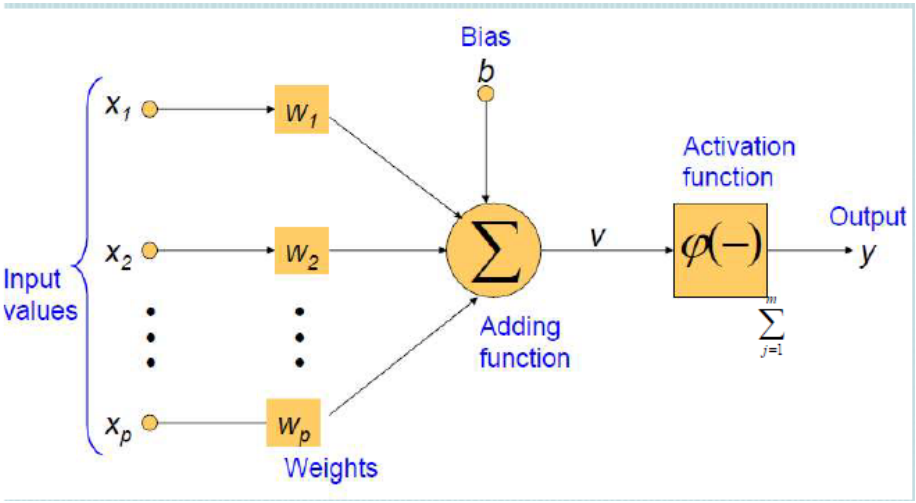


Figura 1.1: Schema di base di una Rete Neurale

al neurone di output è associato un ulteriore input chiamato **bias**: esso rappresenta la tendenza del neurone ad attivarsi oppure no. Ovviamente, maggiore è il bias, maggiore sarà la tendenza del neurone ad attivarsi, poiché aggiunge maggiore informazione alla somma pesata. Per semplificare i calcoli, il bias viene trattato con un ulteriore neurone di input con segnale di input sempre uguale a 1 e peso associato uguale al bias scelto.

A questo punto, il risultato della somma pesata passa attraverso una funzione di attivazione, che restituisce il risultato vero e proprio.

Capitolo 2

Perceptron

2.1 Problemi linearmente separabili

Il Perceptron[1] è il primo esempio di rete neurale. Introdotto nel 1958, esso è in grado di portare a termine dei semplici problemi di classificazione, con particolare attenzione al fatto che il problema sia *linearmente separabile*. In altre parole, affinché un problema sia di questo tipo, gli esempi rappresentati all'interno dello spazio devono poter essere separati in base alla loro classe semplicemente utilizzando una linea retta, chiamata **decision boundary** all'interno dello spazio stesso. In Figura 2.1 possiamo vedere un esempio di problema linearmente

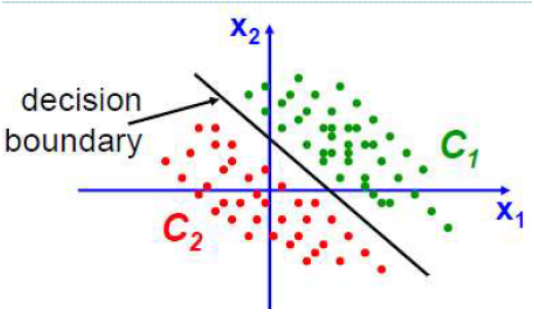


Figura 2.1: Esempio di problema linearmente separabile

separabile, in cui le classi C1 e C2 possono essere facilmente separate da una retta.

[1] in italiano, **percettrone**

2.2 Struttura del Perceptron

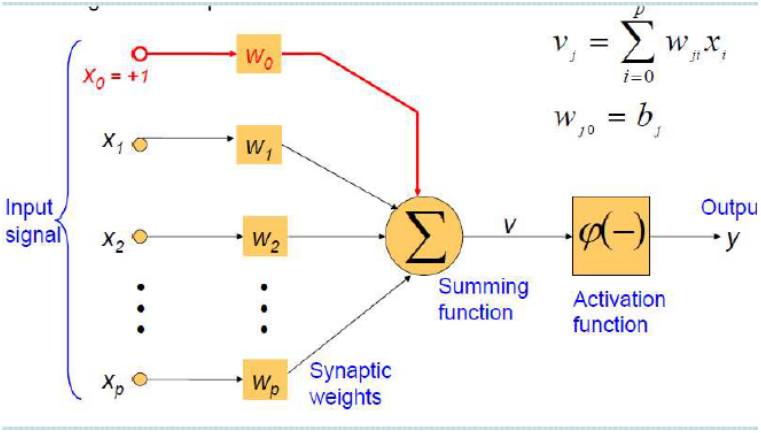


Figura 2.2: Struttura del Perceptron

In Figura 2.2 possiamo vedere la struttura del percettrone, del tutto analogo a quella vista nell'introduzione. Questa rete, quindi, presenta dei neuroni di input, uno per ogni feature degli esempi da trattare, a cui viene aggiunto il neurone di input rappresentante il bias. Ad ogni neurone è associato un peso. Il neurone di output effettua la somma pesata:

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

La funzione di attivazione ϕ è la funzione segno, quindi:

$$\phi(v_j) = \begin{cases} 1 & \text{se } v_j > 0 \\ -1 & \text{altrimenti} \end{cases}$$

Come detto nell'introduzione, il punto cruciale è l'aggiustamento del vettore dei pesi w, le cui componenti del vettore sono i singoli pesi associati ad ogni neurone di input. Più formalmente, possiamo dire che se $x^T \cdot w > 0$ allora l'output sarà 1, altrimenti, se $x^T \cdot w \leq 0$ l'output sarà -1.

Nel caso di due dimensioni, una volta trovato il vettore dei pesi, possiamo ricavare l'equazione della retta che separa correttamente le classi:

$$w_1 x_1 + w_2 x_2 + w_0 = 0$$

2.2.1 Il problema dell'OR

Possiamo fare un piccolo esempio con il problema dell'OR, ossia allenare il percettrone ad effettuare l'OR logico tra due input. Ovviamente non si tratta solo di far memorizzare la regola: in qualche modo è come se la stesse apprendendo da solo, in base agli esempi che gli vengono proposti. Gli esempi da proporre alla rete sono i seguenti:

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

In realtà gli input da fornire al percettrone sono tre, perché bisogna sempre ricordarsi dell'input di bias, in particolare questo compito è dato al primo input, che è sempre posto a 1, gli altri input rappresentano i possibili input dell'OR. Ad ogni esempio è anche fornito il risultato, che in questo caso è proprio il risultato dell'OR logico tra il secondo ed il terzo input.

Dopo che è stato avviato il processo di apprendimento della rete, possiamo vedere che il risultato è quello riportato in Figura 2.3, con i pesi correttamente settati. Graficamente, possiamo visualizzare il risultato in Figura 2.4, sapendo

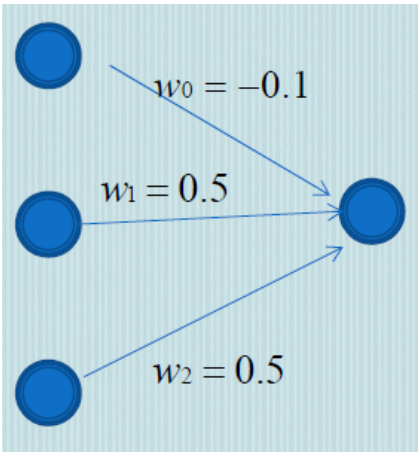


Figura 2.3: Problema dell'OR

che, utilizzando la formula precedentemente riportata, l'equazione della retta è $0.5 \cdot x_1 + 0.5 \cdot x_2 - 0.1 = 0$

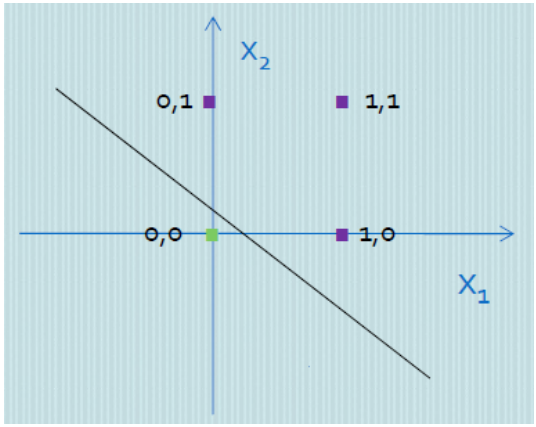


Figura 2.4: Soluzione grafica del problema dell'OR

2.3 Algoritmo di apprendimento del Perceptron

Quando ci soffermiamo sul problema dell'OR, è facile riflettere e trovare una combinazione di pesi che soddisfi il problema. Ma quando il problema è molto più grande? Possiamo applicare un algoritmo di apprendimento. Questo è un algoritmo iterativo che aggiusta i pesi ad ogni iterazione n. Le regole fondamentali per l'aggiornamento dei pesi sono le seguenti:

- se $w(n)^T x(n) > 0$ e $x(n) \in C_1$ allora $w(n+1) = w(n)$, quindi i pesi all'iterazione successiva rimangono invariati se la classificazione per la classe C_1 avviene correttamente
- se $w(n)^T x(n) \leq 0$ e $x(n) \in C_2$ allora $w(n+1) = w(n)$, in questo caso la classificazione per la classe C_2 è corretta, quindi i pesi rimangono invariati
- se $w(n)^T x(n) > 0$ e $x(n) \in C_2$ allora $w(n+1) = w(n) - \eta(n)x(n)$, in questo caso la classificazione per la classe C_2 è errata, quindi i pesi devono essere corretti
- se $w(n)^T x(n) \leq 0$ e $x(n) \in C_1$ allora $w(n+1) = w(n) + \eta(n)x(n)$, in questo caso la classificazione per la classe C_1 è errata, quindi i pesi devono essere corretti

Cerchiamo di entrare nel dettaglio, seguendo l'algoritmo 1. All'inizio i pesi vengono inizializzati in maniera casuale e si comincia ad esplorare lo spazio degli esempi effettuando la somma pesata tra il vettore dei pesi ed il vettore delle features di ogni esempio. Nel momento in cui si scova un esempio classificato erroneamente, allora si va a vedere la sua etichetta $d(n)$ ed in base al suo valore

si applica una certa correzione piuttosto che un'altra. Si continua così finché tutti gli esempi che vengono forniti vengono classificati correttamente.

Algorithm 1 Algoritmo di Apprendimento del Perceptron

```
1:  procedure LEARNING-PERCEPTRON
2:      n ← 0
3:      inizializza w(n) in maniera casuale
4:      while ci sono esempi classificati erroneamente do
5:          (x(n), d(n)) ← esempio mis-classificato
6:          if d(n) = 1 then
7:              w(n + 1) ← w(n) + ηx(n).
8:          if d(n) = −1 then
9:              w(n + 1) ← w(n) − ηx(n).
10:         n ← n + 1.
```

In conclusione, l'algoritmo di apprendimento utilizza i 4 casi appena riportati per il corretto aggiornamento dei pesi. A conti fatti, però, vengono utilizzati solo gli ultimi due, che corrispondono alle regole da utilizzare nel momento in cui un esempio non viene classificato correttamente. In particolare, viene utilizzata una quantità η chiamata **learning rate**. Questo è un parametro della rete che deve essere fornito in input all'algoritmo di apprendimento. In Figura 2.5

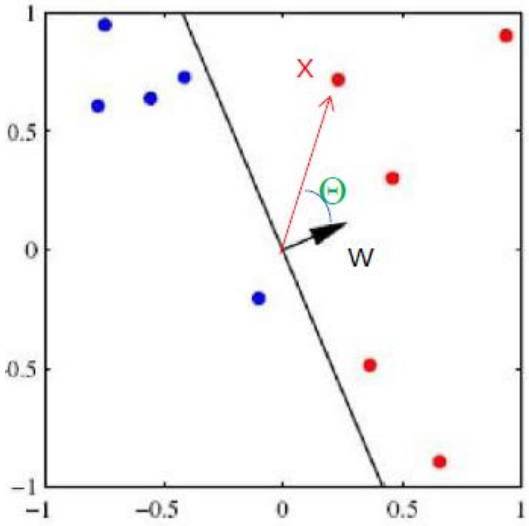


Figura 2.5: Aggiornamento dei pesi

possiamo vedere come l'applicazione dell'algoritmo di apprendimento, equivale all'aggiornamento del vettore dei pesi **w**. Con lo spostamento di questo vettore, non stiamo facendo altro che spostare il decision boundary, che si costruisce disegnando una retta perpendicolare al vettore dei pesi.

2.4 Teorema di convergenza

Attraverso questo teorema, se esiste una soluzione, possiamo stabilire che l'algoritmo di apprendimento termina, trovandola. Questo teorema, si basa sul fatto che esiste un limite superiore ed un limite inferiore alla lunghezza del vettore dei pesi: se si riesce a stabilire questo, allora si può anche concludere che l'algoritmo non può girare all'infinito, di conseguenza prima o poi deve terminare.

Per iniziare, assumiamo che $\eta = 1$ e $w(0) = 0$. Inoltre, diciamo che le classi C_1 e C_2 sono linearmente separabili. Creiamo una classe $C = C_1 \cup C_2$ rimpiazzando tutte le x della classe C_2 in $\neg x$. A questo punto il problema si riduce a trovare un vettore di pesi $\mathbf{w}$ tale che per ogni x in C, ${w^*}^T x > 0$, poiché abbiamo trasformato tutti gli esempi in esempi positivi.

Consideriamo $x(0)...x(k) \in C$ la sequenza di input utilizzati per correggere i pesi nelle prime k iterazioni. Avremo che:

$$\begin{aligned} w(1) &= w(0) + x(0) \\ w(2) &= w(1) + x(1) \\ &\vdots \\ w(k+1) &= w(k) + x(k) \end{aligned}$$

Seguendo il sistema di equazioni, effettuando una serie di sostituzioni, otteniamo che

$$w(k+1) = x(0) + ... + x(k)$$

Quindi il vettore dei pesi è stato aggiornato ogni volta ad ogni iterazione, aggiornando anche la norma del vettore stesso. Dovrebbe essere aggiunta anche la quantità $w(0)$, ma ricordiamo che all'inizio della dimostrazione, l'abbiamo posta al valore 0. Adesso bisogna dimostrare che la quantità k non è infinita.

Per ipotesi, abbiamo detto che esiste un w^* tale che ${w^*}^T x > 0$, $\forall x \in C$. La dimostrazione passa attraverso la scrittura delle seguenti espressioni:

$$w(k+1) = x(0) + \ldots + x(k) \tag{2.1}$$

$${w^*}^T w(k+1) = {w^*}^T (x(0) + ... + x(k)) \tag{2.2}$$

$${w^*}^T w(k+1) = {w^*}^T x(0) + ... + {w^*}^T x(k) \tag{2.3}$$

$${w^*}^T w(k+1) = {w^*}^T x(0) + ... + {w^*}^T x(k) \geq k\alpha \tag{2.4}$$

Partendo dall'equazione riportata in 2.1, il passo successivo è stato moltiplicare per ${w^*}^T x$ ad entrambi i membri dell'equazione, ottenendo 2.2. Dopo, la moltiplicazione è stata distribuita lungo tutti i membri della somma, ottenendo la forma in 2.3. A questo punto abbiamo trovato il lower bound, riportato nella formula 2.4. In particolare, α è il valore più piccolo tra ${w^*}^T x(0) \ldots {w^*}^T x(k)$.

Abbiamo concluso la prima parte della dimostrazione. Per la seconda, dato che $w(k+1) = w(k) + x(k)$, applicando l'operazione di norma, abbiamo che:

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

Dato che siamo nella fase di apprendimento all'iterazione k, siamo sicuri che $2w^T(k)x(k)$ è ≤ 0, dato che $x(k)$ non è stato classificato correttamente. Quindi stiamo sottraendo una quantità, derivando che:

$$||w(k+1)||^2 \leq ||w(k)||^2 + ||x(k)||^2 \tag{2.5}$$

Quindi, per arrivare ad una generalizzazione, possiamo dire che:

$$\begin{aligned} ||w(1)||^2 &\leq ||w(0)||^2 + ||x(0)||^2 \\ ||w(2)||^2 &\leq ||w(1)||^2 + ||x(1)||^2 \\ &\vdots \\ ||w(k+1)||^2 &\leq \sum_{i=0}^{k} ||x(i)||^2 \end{aligned}$$

Si può concludere con:

$$\beta = max||x(i)||^2 \quad (\forall x(i) \in C).$$

Facendo un ragionamento analogo al precedente, possiamo trovare un upper bound, ossia:

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

Avendo quindi trovato un upper bound ed un lower bound, possiamo dire che k non può essere più piccolo di una certa quantità e non può crescere oltre una certa soglia. Visto che l'algoritmo è iterativo, la quantità è costretta a crescere, visto che k rappresenta il numero di iterazioni. Essendoci un upper bound, prima o poi il numero di iterazioni dovranno fermarsi.

2.5 Limiti del Perceptron

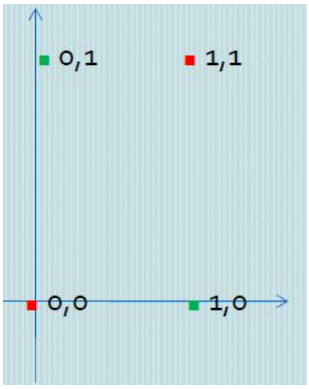


Figura 2.6: Problema dello XOR

All'inizio del capitolo, abbiamo fatto una ipotesi molto importante sul funzionamento del percettrone, ossia che può risolvere soltanto problemi linearmente separabili. Un classico esempio di problema non linearmente separabile è il problema dello XOR. Come si può vedere dalla Figura 2.6, le due classi non possono essere separate semplicemente da una linea retta. Questo problema ha portato ad una certa crisi, che ha fermato la ricerca per un po' di anni, fino a quando non è stata scoperta una nuova struttura per la Reti Neurali.

Capitolo 3

Multilayer Perceptron

3.1 Adaline

Per ora, non addentriamoci ancora nella struttura multilayer, ma rimaniamo su quella singlelayer. Adaline è una struttura che modifica il modo in cui avviene l'aggiornamento dei pesi. In particolare, si focalizza sulla riduzione dell'errore, dato dalla differenza tra il valore ottenuto ed il valore desiderato. Formalizzando, abbiamo in primo luogo:

$$(x^k, d^k)$$

Questa coppia rappresenta il $k - esimo$ esempio preso in considerazione all'interno del training set. In particolare, x^k rappresenta il vettore delle features attraverso le quali descriviamo l'esempio k, mentre d^k rappresenta il valore che deve essere attribuito all'esempio k. A questo punto possiamo definire il concetto di errore:

$$E^k(w) = \frac{1}{2}(d^k - y^k)^2 \tag{3.1}$$

$$= \frac{1}{2}(d^k - \sum_{j=0}^{m} x^k{}_j w_j)^2 \tag{3.2}$$

Con queste due formule, stiamo definendo l'errore per l'esempio k, ossia la differenza tra il valore desiderato d^k ed il valore ottenuto y^k. Il modo in cui si ottiene il valore dalla rete è sempre lo stesso, ossia effettuando la somma pesata degli input per il valore dei pesi degli archi che collegano i nodi: quello che è stato fatto in 3.2 è stata proprio questa sostituzione. Dopo aver definito l'errore

per il singolo esempio, definiamo:

$$E_{tot} = \sum_{k=1}^{N} E^k$$

ossia l'errore totale sulla rete, definito come la somma dei singoli errori su tutti gli esempi k.

3.1.1 Discesa Incrementale del Gradiente

L'algoritmo di apprendimento di Adaline si basa sulla **discesa incrementale del gradiente**. All'inizio i pesi vengono inizializzati in maniera randomica e tutto è da considerare in funzione dell'errore. Immaginando quindi la funzione dell'errore in due dimensioni, possiamo vedere l'asse x come il punto in cui vengono inizializzati i pesi e sull'asse y l'errore associato. A questo punto utilizziamo il gradiente della funzione errore, definito come:

$$\nabla E^k(w) = \left[\frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_m} \right]$$

Ricordando che il gradiente è definito come un vettore le cui componenti sono le derivate parziali rispetto agli argomenti della funzione. In questo caso si ha un elemento del gradiente per ogni elemento del vettore dei pesi.

Visto che il gradiente è un vettore che punta verso la massima crescita della funzione e noi vogliamo minimizzare l'errore, ci spostiamo esattamente nella direzione opposta, ottenendo:

$$w(k+1) = w(k) - \eta(\nabla E^k(w)) \tag{3.3}$$

Adesso dobbiamo concentrarci sul calcolo di ogni singolo componente del gradiente.

3.1.2 Regola di aggiornamento dei pesi

Ogni componente del gradiente è dato dalla derivata della funzione errore rispetto ad ogni componente del vettore pesi. Di conseguenza:

$$\frac{\partial E_{tot}}{\partial w_j} = \sum_{k=1}^{N} \frac{\partial E^k(w)}{\partial w_j} \tag{3.4}$$

$$= \sum_{k=1}^{N} \frac{\partial}{\partial w_j} \frac{1}{2} (d^k - \sum_{j=0}^{m} x^k{}_j w_j)^2 \tag{3.5}$$

Nell'equazione 3.4 è stata semplicemente riportata la formula dell'errore totale, definito attraverso l'errore di ogni singolo esempio k. Nell'equazione 3.5 è stata sostituita l'espressione del singolo errore con la differenza tra il valore reale dell'esempio k ed il valore ottenuto come somma pesata degli input per l'esempio k.

Bisogna notare, però, che non c'è una correlazione diretta tra l'errore totale (numeratore della frazione) e il peso w_j (denominatore della frazione): infatti, variando il peso w_j, vario di conseguenza l'uscita ottenuta per l'esempio k, di conseguenza variando quest'ultima, vario l'errore per il singolo esempio k, che andrà infine a variare l'errore totale. Si crea, quindi, questa sorta di catena immaginaria che collega l'errore totale alla variazione di un singolo peso. Alla luce di quanto appena detto, possiamo vedere l'errore totale in funzione del peso come una funzione composta. Dobbiamo applicare, quindi una derivata su funzione composta. La catena di equazioni diventa la seguente:

$$\frac{\partial E^k(w)}{\partial w_j} = \frac{\partial E^k(w)}{\partial y^k}\frac{\partial y^k}{\partial w_j} \tag{3.6}$$

$$= -(d^k - y^k){x_j}^k \tag{3.7}$$

Nell'equazione 3.6 è vi è l'applicazione della derivata di funzioni composte, sapendo che bisogna passare per l'uscita y^k. Banalmente, la si potrebbe vedere come una moltiplicazione per la frazione $\frac{\partial y^k}{\partial y^k}$ e poi scambiare i denominatori. In 3.7, invece, è stata applicata una semplice sostituzione, ricordando sempre che l'errore sul singolo esempio k è dato dalla differenza di ciò che volevamo e ciò che abbiamo ottenuto. Per concludere, la regola di aggiornamento dei pesi diventa la seguente:

$$w(k+1) = w(k) + \eta(d^k - y^k)x^k \tag{3.8}$$

Che è semplicemente la riscrittura della formula 3.3 esplicitando la computazione del gradiente nel caso di Adaline.

3.2 Multi Layer Feed-Forward Neural Network

Per gli amici **FFNN**, questa rete è di fatto lo step successivo al percettrone, in quanto permette di risolvere problemi non linearmente separabili.

In Figura 3.1 possiamo vedere la struttura di una FFNN. La novità di questa rete è che presenta un nuovo livello di neuroni, chiamato livello **hidden**. Come si può vedere dalla Figura 3.2, il numero di neuroni crea delle regioni convesse che permettono ognuna di definire una determinata classe di appartenenza degli esempi. La posizione e l'inclinazione delle rette varia in base ai pesi che vengono associati.

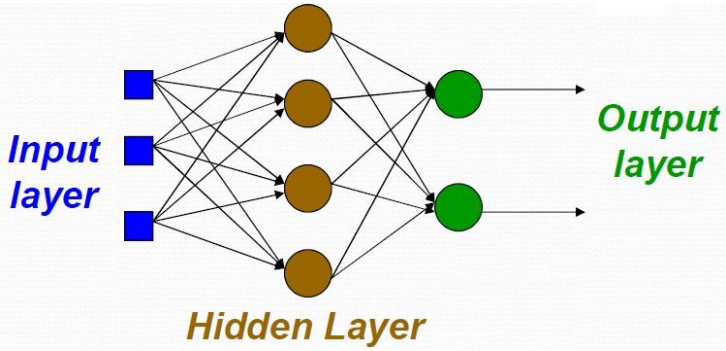


Figura 3.1: Struttura di una FFNN

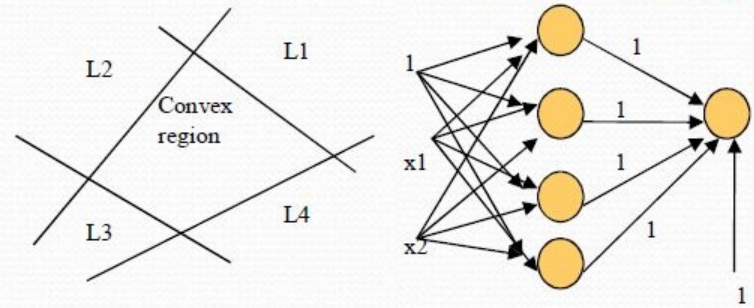


Figura 3.2: Creazione di regioni convesse

Tipicamente, la funzione di attivazione è la funzione sigmoide, che possiamo vedere in Figura 3.3. Prima di continuare nella trattazione, bisogna introdurre diverse notazioni:

- $\phi(v_j) = \dfrac{1}{1+e^{-av_j}}$ con $a > 0$, ossia la funzione di attivazione sigmoide
- $v_j = \sum w_{ji} y_i$ definita come l'uscita del neurone di output j,
- y_i uscita del neurone hidden i
- w_{ji} peso del link tra il neurone j ed il neurone i

3.2.1 L'algoritmo di Backpropagation

Questo è l'algoritmo di apprendimento di una FFNN. È diviso in due fasi:

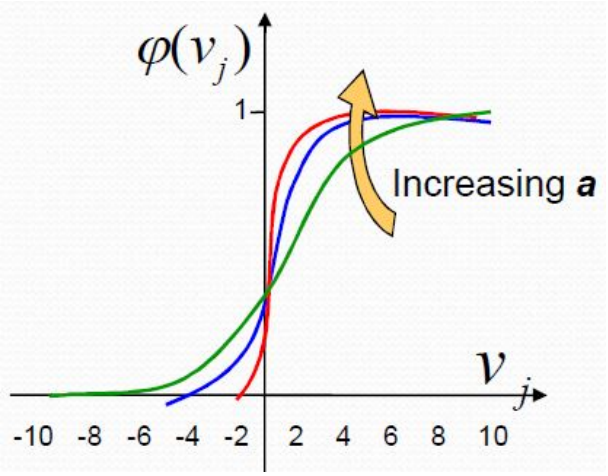


Figura 3.3: Funzione Sigmoide

- **forward pass**: fase in cui la rete viene attivata su uno specifico esempio, calcolando l'errore dei neuroni di output
- **backward pass**: fase in cui l'errore calcolato nella fase precedente viene utilizzato per correggere i pesi dei vari link

Prima di procedere nella trattazione, bisogna introdurre un nuovo tipo di errore, ossia l'errore del neurone di **output j** per l'esempio n:

$$e_j(n) = d_j(n) - y_j(n)$$

A questo punto, si può definire l'errore totale della rete come segue:

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

Infine, definiamo l'errore medio come segue:

$$E_{AV} = \frac{1}{N} \sum_{n=1}^{N} E(n)$$

Ossia la media degli errori ottenuti su tutto il training set. Anche in questo caso viene applicata la discesa del gradiente, ottenendo:

$$\begin{aligned} w_{ji} &= w_{ji} + \Delta w_{ji} \\ \Delta w_{ji} &= -\eta \frac{\partial E}{\partial w_{ji}} \end{aligned} \tag{3.9}$$

Proseguendo, possiamo riutilizzare la formula 3.6 per definire l'errore in funzione dei pesi, aggiungendo il fatto che ora abbiamo un vettore di pesi più complesso, le cui componenti sono w_{ji} e l'uscita dei neuroni di output è v_j:

$$\begin{aligned}\frac{\partial E}{\partial w_{ji}} &= \frac{\partial E}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}} \\ &= -\delta_j y_i \end{aligned} \tag{3.10}$$

Possiamo notare nell'equazione 3.10 che è stata definita la quantità **segnale di errore** $\delta_j = -\frac{\partial E}{\partial v_j}$. Alla fine, effettuando delle semplici sostituzioni all'equazione 3.9, otteniamo la formulazione:

$$\Delta w_{ji} = \eta \delta_j y_i \tag{3.11}$$

La particolarità di questo algoritmo sta nel fatto che la quantità Δw_{ji} è differente nei due casi, ossia quando ci troviamo nel livello hidden o quando ci troviamo nel livello di output. In particolare, dobbiamo conoscere il valore del segnale di errore δ_j, sapendo che modificare i pesi dei neuroni, significa modificarne gli input, per questo motivo ci concentriamo sugli input dei vari neuroni.

Aggiornamento dei pesi del livello di output

Questo è il caso più semplice, poiché il livello di output è in grado di sapere immediatamente l'errore che si sta commettendo. Partiamo dicendo che dobbiamo calcolare la quantità

$$\delta_j = -\frac{\partial E}{\partial v_j}$$

Anche in questo caso dobbiamo fare un discorso sullo sviluppo delle derivate di funzioni composte. Variando l'uscita v_j viene modificato anche il valore della funzione di trasferimento $y_j = \phi(v_j)$. Variando questo valore, varia anche l'errore commesso dal neurone di output indicato con e_j, ricordando che è possibile calcolarlo in questo livello. Passiamo attraverso la composizione di funzioni con altri due valori, ottenendo la seguente formulazione:

$$\begin{aligned}-\frac{\partial E}{\partial v_j} &= -\frac{\partial E}{\partial e_j}\frac{\partial e_j}{\partial y_j}\frac{\partial y_j}{\partial v_j} \\ &= -e_j(-1)\phi'(v_j) \end{aligned} \tag{3.12}$$

In conclusione, riprendendo l'equazione 3.11, sostituendo il valore δ_j appena calcolato otteniamo il risultato per il caso dei neuroni di output:

$$\Delta w_{ji} = \eta(d_j - y_j)\phi'(v_j)y_i \tag{3.13}$$

ricordandoci che:

$$e_j = d_j - y_j$$

Aggiornamento dei pesi del livello hidden

Anche in questo caso vogliamo calcolare la quantità

$$\delta_j = -\frac{\partial E}{\partial v_j}$$

La cosa che cambia in questo livello è che non siamo in grado di calcolare direttamente l'errore commesso dai neuroni. Quindi, utilizzando la regola di derivazione delle funzioni composte, otteniamo:

$$-\frac{\partial E}{\partial v_j} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}$$

Sappiamo che:

$$\frac{\partial E}{\partial y_j} = \sum_k \frac{\partial E}{\partial v_k}\frac{\partial v_k}{\partial y_j} \tag{3.14}$$

$$\frac{\partial y_j}{\partial v_j} = \phi'(v_j) \tag{3.15}$$

Con l'equazione 3.14, stiamo dicendo che variando l'uscita del neurone hidden stiamo andando a variare tutte le uscite v_k dei k neuroni del livello successivo, poiché ne sto modificando gli input. Con l'equazione 3.15, stiamo indicando il variare dell'uscita del neurone in base al suo ingresso. Effettuando le dovute sostituzioni alla formula 3.11, otteniamo il risultato per il caso dei neuroni hidden:

$$\Delta w_{ji} = \eta y_i \phi'(v_j) \sum_k \delta_k w_{kj} \tag{3.16}$$

Riassumendo: Delta Rule

La Delta Rule è la seguente:

$$\begin{aligned} w_{ji} &= w_{ji} + \Delta w_{ji} \\ \Delta w_{ji} &= \eta \delta_j y_i \end{aligned} \tag{3.17}$$

con

$$\delta_j = \begin{cases} \phi'(v_j)(d_j - y_j) & \text{se } j \text{ è un nurone di output} \\ \phi'(v_j) \sum_k \delta_k w_{kj} & \text{se } j \text{ è un neurone hidden} \end{cases}$$

con

$$\phi'(v_j) = \alpha y_j (1 - y_j)$$

Delta Rule Generalizzata

La formula 3.17 prende il nome di **delta rule** e si occupa di fornire la formula di aggiornamento dei pesi. Possiamo definirne una formula più generale, come segue:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j y_i \tag{3.18}$$

Nella formula 3.18 viene aggiunto un quantitativo α chiamato **termine di momento**: insieme al learning rate veicola l'aggiustamento dei pesi. Infatti, il Δw all'iterazione attuale, ora dipende anche dal Δw all'iterazione precedente. Il quantitativo dell'iterazione precedente viene pesato attraverso il termine di momento. Il problema di questi due quantitativi è che devono essere settati prima di avviare l'algoritmo di apprendimento della rete. Per questo c'è bisogno di una fase di validazione di questi due parametri al valore migliore. Per questo viene utilizzato un set di esempi chiamato **validation test**, simile al training set ma con degli scopi diversi.

Fortunatamente esistono alcune euristiche che guidano il settaggio di questi valori, in particolare del termine η, in particolare,

- ogni peso ha la sua η
- il termine η è fisso per tutti i pesi e può variare da iterazione ad iterazione

L'algoritmo di apprendimento può continuare ad aggiustare i pesi finché non viene soddisfatta una certa condizione stabilita a priori, come per esempio la discesa dell'errore di classificazione sotto una certa soglia.

3.3 Design di una Rete Neurale

Quando si crea una Rete Neurale, bisogna tenere conto di alcuni fattori:

- il modo in cui i dati devono essere rappresentati, ad esempio effettuando una normalizzazione
- numero di layer e numero di neuroni per layer, in particolare si può partire da una rete sovradimensionata per poi ridurne la complessità o viceversa
- i parametri, come il learning rate ed il termine di momento, ma anche come l'inizializzazione dei pesi nello stato iniziale

3.4 Rete Neurale come approssimatore di funzioni

Di fatto, una Rete Neurale può essere vista come una approssimazione di una funzione che effettua una interpolazione di tutti i punti del training set all'interno dello spazio. La funzione si costruisce spostandola all'interno dello spazio in modo da minimizzare l'errore, dato dalla distanza dei punti dalla funzione.

Capitolo 4

Reti Neurali Radiali

4.1 Architettura della Rete

Questo tipo di Reti Neurali utilizzano un metodo di classificazione basato su distanza. In particolare, ogni punto nello spazio che si prende in considerazione viene visto come distante da alcuni vettori che sono dei "recettori" di determinate classi. L'algoritmo di apprendimento si occupa di spostare questi recettori nello spazio degli esempi. In Figura 4.1 possiamo vedere un esempio di quanto

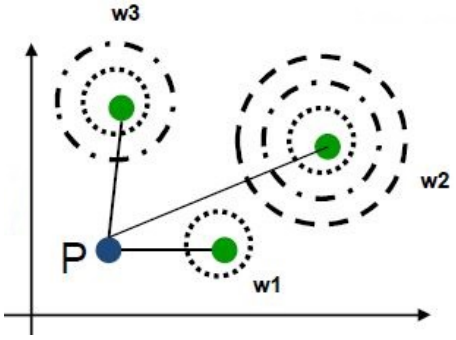


Figura 4.1: Vettori w

appena detto, in particolare i tre diversi vettori w_1, w_2 e w_3 rappresentano 3 diversi centri all'interno dello spazio ed il punto P viene interpolato calcolando la distanza del punto dai tre diversi centri. La differenza con le reti neurali multi-layer sta nel fatto che le reti radiali tendono ad avere una specializzazione locale, mentre quelle multi-layer hanno una specializzazione più globale, basti pensare al percettrone che, una volta stabilito il decision boundary, è in grado

di classificare anche esempi molto lontani dal boundary, cosa che una rete radiale (immaginando il concetto di decision boundary per una rete radiale) non saprebbe fare. Ad ogni modo, la struttura di una rete radiale è riportata in

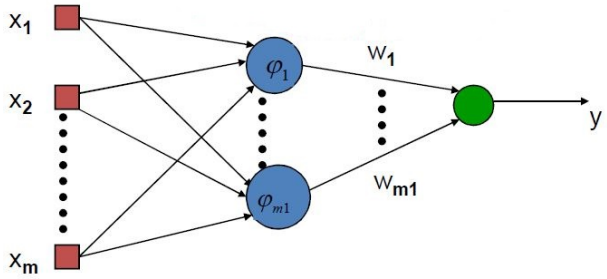


Figura 4.2: Struttura di una Rete Radiale

Figura 4.2. Ciò che cambia in questa rete sono i neuroni del livello hidden: ciascun neurone, infatti, rappresenta una funzione ϕ che prende come argomento una distanza, ne consegue che bisogna inserire all'interno della rete un neurone per ogni centro che si vuole inserire all'interno dello spazio. Il livello di output, invece, ha una funzione di attivazione. In particolare, l'output y è calcolato nel seguente modo:

$$y = w_1\phi_1(||x - t_1||) + ... + w_{m1}\phi_{m1}(||x - t_m||) \tag{4.1}$$

Bisogna stare attenti all'utilizzo della funzione ϕ_n che prende come argomento la distanza dell'esempio x dal vettore t.

4.2 Modello del Neurone Hidden

Vediamo ora la particolare struttura di un neurone hidden. In Figura 4.3, pos-

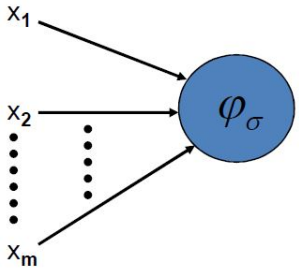


Figura 4.3: Neurone Hidden

siamo vedere che all'interno di un neurone del livello hidden è presente una

funzione ϕ_σ, definita in questo modo:

$$\phi_\sigma(||x - t||) \tag{4.2}$$

L'output di questa funzione dipende dalla distanza di x dal centro t. Inoltre, la quantità σ definisce lo **spread**, ossia quello che in Figura 4.1 è segnato come i cerchi intorno ai punti al centro dei vettori w. All'inizio, queste due quantità sono dati in input alla rete, in realtà alcuni algoritmi di apprendimento fanno in modo di valorizzare anche quelle.

Di fatto, un neurone hidden è più sensibile ai punti vicino al suo centro. Per le Reti Neurali con funzione ϕ gaussiana, per esempio, la sensibilità può essere impostata modificando lo spread σ: più grande è questo valore, minore sarà la sensibilità.

4.3 Il Problema dell'Interpolazione

Dato un insieme di N punti differenti $\{x_i \in \mathcal{R}^m, i = 1...N\}$ ed un insieme di N numeri reali $\{d_i \in \mathcal{R}, i = 1...N\}$, trovare una funzione $F : \mathcal{R}^m \Rightarrow \mathcal{R}$ che soddisfi la condizione di interpolazione: $F(x_i) = d_i$.

In altre parole, bisogna trovare una funzione che dato un esempio x_i, restituisca il target d_i.

Se $F(x) = \sum_{i=1}^{N} w_i \phi(||x - x_i||)$, cioè la somma dell'equazione 4.1, abbiamo:

$$\begin{bmatrix} \phi(||x_1 - x_1||) & \dots & \phi(||x_1 - x_N||) \\ \dots & \dots & \dots \\ \phi(||x_N - x_1||) & \dots & \phi(||x_N - x_N||) \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ \dots \\ w_N \end{bmatrix} = \begin{bmatrix} d_1 \\ \dots \\ d_N \end{bmatrix} \Rightarrow \Phi w = d \tag{4.3}$$

Nell'equazione 4.3 stiamo dicendo che la matrice delle funzioni radiali moltiplicato per il vettore colonna dei pesi risulta essere il vettore colonna dei valori desiderati. In particolare, si stanno prendendo in considerazione le distanze tra tutti gli esempi $x_1, ..x_N$ con lo stesso numero di funzioni ϕ. Bisogna porre attenzione ad un particolare. L'input di ogni funzione ϕ è rappresentato da una differenza. Il primo termine della sottrazione è l'input, mentre il secondo termine è il centro: la conclusione è che si sta prendendo un numero di centri (e di conseguenza, anche funzioni) pari al numero degli esempi e, dato che ogni centro rappresenta un neurone hidden, si sta prendendo un numero di neuroni hidden pari al numero degli esempi. Ovviamente questo non è tollerabile, ma è ciò da cui si deve partire. Il motivo per cui non è tollerabile è intuitivo: dato che si sta costruendo una rete con un neurone per ogni esempio nel training set, si sta specializzando la rete solo per quell'insieme di esempi, quindi non sarà in

grado di generalizzare, ma addestrerà ogni neurone a riconoscere un particolare esempio, generando, di fatto, overfitting.

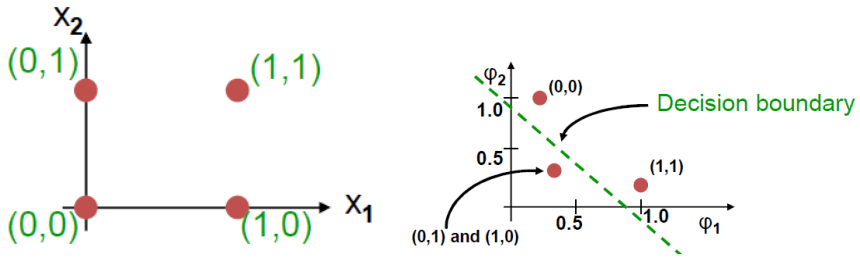
Ci sono diversi tipi di funzioni Φ, quella più usata è la gaussiana, che, per completezza, viene riportata di seguito:

$$\phi(r) = exp(-\frac{r^2}{2\sigma^2}) \tag{4.4}$$

con $\sigma > 0$

4.4 Il problema dello XOR

Attraverso l'esempio dello XOR, vediamo come si comporta di fatto una Rete Neurale Radiale. Le istanze del problema sono rappresentate in Figura 4.4a.



(a) *Prima* (b) *Dopo*

Figura 4.4: Spostamento delle istanze nel problema dell'OR

Possiamo definire le seguenti due funzioni:

$$\phi_1(||x - t_1||) = e^{-||x-t_1||^2}$$
$$\phi_2(||x - t_2||) = e^{-||x-t_2||^2}$$

con $t_1 = (1, 1)$ e $t_2 = (0, 0)$. Lo spazio delle features, viene mappato all'interno di un nuovo spazio che possiamo vedere in Figura 4.4b, in cui si può notare che gli assi non sono più x_1 e x_2 bensì ϕ_1 e ϕ_2: questo vuol dire che viene effettuato uno spostamento dei punti all'interno dello spazio degli esempi. Questo spostamento permette al problema di essere linearmente separabile. A questo punto, i neuroni di output si comportano come un normale classificatore che risolve problemi linearmente separabili. Questa caratteristica può essere vista in Figura 4.5 in cui vengono aggiunti i due neuroni hidden corrispondenti ai centri t_1 e t_2, collegati ad un singolo neurone di output che si occupa della classificazione lineare. Nel nostro particolare caso, abbiamo che:

$$y = -e^{-||x-t_1||^2} - e^{-||x-t_2||^2} + 1$$

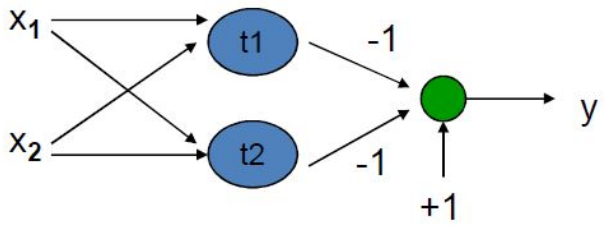


Figura 4.5: Piccola Rete Neurale Radiale

4.5 Algoritmi di Apprendimento

Data una particolare struttura di una RNN, ciò che dobbiamo apprendere attraverso gli algoritmi di apprendimento sono i valori dei centri, degli spreads e dei pesi. In letteratura sono presenti diversi algoritmi di apprendimento, ne vediamo tre.

4.5.1 Algoritmo di Apprendimento 1 - Matrice Pseudo-Inversa

In questo algoritmo i centri sono valorizzati casualmente, scelti dal training set, mentre lo spread è definito come:

$$\sigma = \frac{d_{max}}{\sqrt{m_1}} \tag{4.5}$$

dove d_{max} è la massima distanza calcolata tra ogni coppia di centri, mentre m_1 è il numero di centri, definito dall'utente. La funzione di attivazione cambia di conseguenza:

$$\phi_i(||x - t_i||^2) = exp(-\frac{m_1}{d^2_{max}}||x - t_i||^2)$$

Ora vediamo come calcolare i pesi. Dato un esempio (x_i, d_i) l'output della rete sarà:

$$y(x_i) \approx w_1\phi_1(||x_i - t_1||^2) + ... + w_{m1}\phi_{m1}(||x_i - t_{m1}||^2)$$

Il nostro obiettivo è fare in modo che $y(x_i) = d_i$ per ogni esempio. Questa cosa è possibile solo nel caso in cui si abbia un neurone hidden per ogni esempio, in modo che ogni neurone si specializzi per un particolare esempio e che riconosca solo quello: come abbiamo già anticipato, questo non è possibile per motivi di spazio e per motivi di overfitting. Ci spostiamo quindi verso una approssimazione del risultato, con $m1$ centri abbiamo:

$$w_1\phi_1(||x_i - t_1||^2) + ... + w_{m1}\phi_{m1}(||x_i - t_{m1}||^2) \approx d_i$$

Scrivendo questa formula in notazione matriciale, otteniamo:

$$\begin{bmatrix}\phi_1(||x_i - t_1||) & \dots & \phi_{m1}(||x_i - t_{m1}||)\end{bmatrix} \begin{bmatrix} w_1 \\ \dots \\ w_{m1}\end{bmatrix} = d_i$$

Questa formulazione serve soltanto a trovare i giusti pesi per la classificazione dell'esempio x_i, possiamo fare di più, cioè effettuare una singola moltiplicazione matriciale inserendo tutti gli esempi $x_1...x_N$, trovando i pesi in un colpo solo. Partiamo da:

$$\begin{bmatrix}\phi_1(||x_1 - t_1||) & \dots & \phi_{m1}(||x_1 - t_{m1}||) \\ \dots & \dots & \dots \\ \phi_1(||x_N - t_1||) & \dots & \phi_{m1}(||x_N - t_{m1}||)\end{bmatrix} \begin{bmatrix} w_1 \\ \dots \\ w_{m1}\end{bmatrix} = \begin{bmatrix} d_1...d_N \end{bmatrix}^T \tag{4.6}$$

Dobbiamo ricordarci che il nostro obiettivo è trovare la matrice dei pesi. Riferendoci alla formula 4.6, la matrice delle ϕ ora la chiameremo Φ, dunque avremo:

$$\begin{aligned} \Phi w &= d \\ w &= \Phi^{-1} d \end{aligned} \tag{4.7}$$

Quindi dovremmo invertire la matrice Φ, ma questa operazione non può essere eseguita, perché la matrice non è quadrata, quindi non è invertibile. A questo punto ci viene in aiuto la **matrice pseudo-inversa**, definita come:

$$\Phi^+ \equiv (\Phi^T \Phi)^{-1} \Phi^T$$

Questa matrice, funge da matrice inversa di una matrice non invertibile. Riprendendo l'equazione 4.7 otteniamo:

$$\begin{aligned} w &= \Phi^+ d \\ \begin{bmatrix} w_1...w_{m1} \end{bmatrix}^T &= \Phi^+ \begin{bmatrix} d_1...d_N \end{bmatrix}^T \end{aligned} \tag{4.8}$$

Ricapitolando:

1. seleziona i centri in maniera casuale dal training set
2. calcola lo spread applicando la normalizzazione dell'equazione 4.5
3. trova i pesi utilizzando il metodo della matrice pseudo-inversa, utilizzando l'equazione 4.8

4.5.2 Algoritmo di Apprendimento 2 - Calcolo dei Centri

Questo algoritmo viene utilizzato per trovare i centri della rete.

1. **inizializzazione**: $t_k(o)$[1] casuali con $k = 1, ..., m_1$
2. **sampling**: seleziona un x dallo spazio degli input
3. **similarity matching**: trova l'indice k del centro più vicino a x con:

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

   ossia l'indice del centro che minimizza la distanza con l'input x

4. **updating**: aggiusta i centri con:

$$t_k(n+1) = \begin{cases} t_k(n) + \eta \left[x(n) - t_k(n) \right] & \text{se } k = k(x) \\ t_k(n) & \text{altrimenti} \end{cases}$$

5. **iterazione**: incrementa η, ritorna al punto 2 e continua finché non viene fatto alcun aggiustamento

Di fatto, quello che stiamo andando ad applicare, è un semplice algoritmo di clustering per trovare i centri. Questo algoritmo viene chiamato anche **algoritmo ibrido** perché mette insieme più algoritmi di apprendimento per poterne formare uno solo. Infatti, dopo aver trovato i centri, gli spreads vengono trovati attraverso la normalizzazione con l'equazione 4.5 e i pesi vengono calcolati attraverso un algoritmo di apprendimento lineare, per esempio quello di Adaline, dato che la rete attraverso i suoi centri sposta gli esempi all'interno dello spazio in modo che il problema diventi lineramnete separabile.

4.5.3 Algoritmo di Apprendimento 3 - Discesa del Gradiente

Questo algoritmo applica il metodo della discesa del gradiente per trovare i centri, gli spreads ed i pesi, in particolare:

- **centri**:

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

- **spread**:

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

[1] ricordiamo che t_k sono i centri

- **pesi**:

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

con $E = \frac{1}{2}(y(x) - d)^2$, quindi il classico errore dato dalla differenza tra il valore ottenuto ed il valore desiderato.

4.6 Confronto con le Reti FFNN

Confrontando le Reti Radiali con quelle Feed-forward, possiamo dire che le Reti Radiali sono utilizzate per risolvere problemi di regressione e di classificazione non lineare, questa è una analogia con le reti FFNN, anche perché entrambe possono essere viste come approssimatori di funzioni che svolgono gli stessi compiti.

Una prima differenza si può trovare nell'**architettura**, in quanto le reti RBF hanno un singolo livello hidden, mentre le reti FFNN possono avere più livelli hidden.

Altre differenze si trovano nel **modello del neurone**, in quanto nelle RBF i neuroni hidden svolgono funzioni diverse da quelli di output, in particolare il livello hidden è non lineare, mentre quello di output è lineare. Nelle reti FFNN entrambi i livelli sono non lineari e svolgono la stessa funzione.

Un'altra differenza sostanziale si trova nel **metodo di approssimazione**, in particolare, come abbiamo detto nell'introduzione, le RBF sono degli approssimatori locali, in quanto vengono settati diversi centri che si occupano di una particolare zona dello spazio, mentre le FFNN sono degli approssimatori globali, che si limitano a disegnare delle rette nello spazio e può classificare tutto ciò che si trova da una parte o dall'altra delle rette.

Un ultima differenza possiamo trovarla nella **funzione di attivazione**: mentre le reti RBF utilizzano una distanza euclidea come funzione di attivazione, le reti FFNN utilizzano un prodotto scalare tra il vettore di input ed i pesi.

Capitolo 5

Extreme Learning Machines

5.1 Features

Questo non è un vero e proprio tipo di rete, ma più che altro un modo per apprendere i pesi in una rete con dei livelli hidden. I pesi vengono suddivisi in pesi di **primo livello** e pesi di **secondo livello**. Per spiegarla, osserviamo la Figura 5.1. I pesi di primo livello sono quelli che si trovano tra il livello di

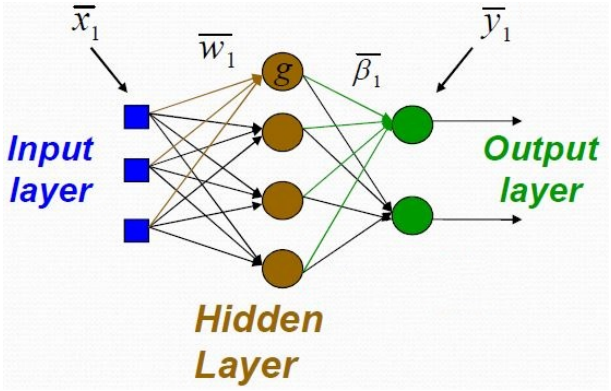


Figura 5.1: Modello della Rete Neurale

input ed il livello hidden. Nell'ELM, questi pesi vengono settati casualmente, secondo una distribuzione data. I pesi di secondo livello sono quelli che si trovano tra il livello hidden ed il livello di output, questi invece sono oggetto di apprendimento, che può avvenire in diversi modi. Bisogna precisare che, nella Figura 5.1, prendendo ad esempio $\overline{w}_1$, esso non è un singolo valore, bensì a sua

volta un vettore, che descrive i tre diversi pesi che vanno dai tre neuroni di input al primo neurone hidden, stessa cosa per le altre variabili. In notazione matriciale, abbiamo:

$$g\left[\begin{bmatrix}\overline{x}_1\\ \overline{x}_2\\ \vdots\\ \overline{x}_N\end{bmatrix}\begin{bmatrix}\overline{w}_1 \dots \overline{w}_4\end{bmatrix}\right] \cdot \begin{bmatrix}\overline{\beta}_1 \ \overline{\beta}_2\end{bmatrix} = \begin{bmatrix}\overline{y}_1\\ \overline{y}_2\\ \vdots\\ \overline{y}_N\end{bmatrix} \tag{5.1}$$

con:

$$\overline{y}_i = \sum_{h=1}^{4} \beta_h g(\overline{x}_i), \quad \text{con } i = 1, \dots, N$$

L'equazione 5.1 può essere riscritta in una forma più generale, ricordando che $\tilde{N}$ è il numero di neuroni hidden e m è il numero di neuroni di output:

$$\begin{bmatrix} g(\overline{x}_1 \cdot \overline{w}_1) & \dots & g(\overline{x}_1 \cdot \overline{w}_{\tilde{N}}) \\ \vdots & & \vdots \\ g(\overline{x}_N \cdot \overline{w}_1) & \dots & g(\overline{x}_N \cdot \overline{w}_{\tilde{N}}) \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \dots & \beta_{1m} \\ \vdots & & \vdots \\ \beta_{\tilde{N}1} & \dots & \beta_{\tilde{N}m} \end{bmatrix} = \begin{bmatrix} y_{11} & \dots & y_{1m} \\ \vdots & & \vdots \\ y_{N1} & \dots & y_{Nm} \end{bmatrix} \tag{5.2}$$

Da questo, otteniamo:

$$\begin{aligned} H \cdot \beta &= y \\ \beta &= H^{-1} \cdot y \end{aligned} \tag{5.3}$$

Questa formulazione la otteniamo perchè essendo che i pesi $\overline{w}$ sono settati casualmente, sono già dati. Il nostro compito è trovare i pesi β. Per fare ciò, guardando la Formula 5.3, dobbiamo passare per l'inversa della matrice H: anche questa volta abbiamo il problema che questa matrice non è invertibile, quindi dobbiamo calcolarne la pseudoinversa.

5.2 Teoremi

Ci serviremo di due teoremi molto importanti.

Teorema 2.1. *Data una rete standard, con N neuroni hidden ed una funzione di attivazione $g : \mathcal{R} \to \mathcal{R}$, con N esempi (x_i, t_i), dove $x_i \in \mathcal{R}^n$ e $t_i \in \mathcal{R}^m$, per ogni w_i e b_i selezionati casualmente, la matrice dei neuroni hidden H è invertibile e si ha che $||H\beta - T|| = 0$*

In poche parole, il Teorema 2.1 ci sta dicendo che nel momento in cui si hanno N neuroni hidden ed N esempi, quindi un neurone hidden per ogni esempio, si

può avere una matrice dei pesi H che moltiplicata per la matrice dei pesi β, si ha errore nullo (dato da $||H\beta - T|| = 0$). Ma questo abbiamo già detto che è infattibile. Per questo ci viene in aiuto un secondo teorema:

Teorema 2.2. *Dato un valore positivo $\epsilon > 0$ ed una funzione di attivazione $g : \mathcal{R} \to \mathcal{R}$, esiste una funzione $\tilde{N} \leq N$ tale che per un N arbitrario di esempi (x_i, t_i) dove $x_i \in \mathcal{R}^n$ e $t_i \in \mathcal{R}^m$, per ogni w_i e b_i selezionati casualmente, si ha che $||H_{N\times\tilde{N}}\beta_{\tilde{N}\times m} - T_{N\times m}|| < \epsilon$*

Quindi, il Teorema 2.2, ci sta dicendo che nel momento in cui non vogliamo istituire un numero di neuroni hidden pari al numero di esempi, possiamo avere un numero $\tilde{N}$ di neuroni hidden minore ed avremo comunque che l'errore non sarà più grande di un certo ϵ. Ovviamente, il numero di neuroni hidden è in funzione dell'errore che non si vuole superare: più si sceglie un errore basso, più siamo costretti ad aumentare il numero di neuroni hidden. Se volessimo errore nullo, allora dovremmo avere un numero di neuroni hidden pari al numero degli esempi, ricadendo nel Teorema 2.1.

5.3 Algoritmo di Apprendimento

Dato un training set $\mathcal{N} = \{(x_i, t_i) | x_i \in \mathcal{R}^n, t_i \in \mathcal{R}^m, i = 1, \ldots, N\}$, una funzione di attivazione $g(x)$ e un numero di neuroni hidden $\tilde{N}$:

1. assegna casualmente i pesi w_i ed il bias b_i con $i = 1, \ldots, \tilde{N}$
2. calcola la matrice H
3. calcola la matrice dei pesi β con $\beta = H^+T$, dove $T = \begin{bmatrix} t_1, \ldots, t_n \end{bmatrix}^T$

Uno dei problemi da risolvere in questo algoritmo è calcolare la matrice H. Uno dei modi per calcolarla è utilizzare la matrice psudoinversa, avendo

$$H^+ = (H^T H)^{-1} H^T \tag{5.4}$$

dove H è una matrice i cui elementi $h_{i,j}$ sono, per esempio:

$$h_{i,j} = g(\overline{x}_i, \overline{w}_j) = \frac{1}{1 + exp(\frac{-\overline{x}_i \cdot \overline{w}_j}{\sigma})}$$

se viene scelta una funzione di attivazione sigmoide.

Singolarità della matrice

Per come è formata la matrice H, essa potrebbe diventare singolare. Per risolvere questa problematica, viene aggiunto un termine di regolarizzazione, che è un

termine scalare che viene moltiplicato per la matrice identità. Di conseguenza si ottiene:

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

Dato che la matrice identità dispone di 1 sulla diagonale principale e di 0 sul resto degli elementi, di fatto andiamo a sommare sulla diagonale principale di $H^T H$ la quantità λ. Questo procedimento deriva dal fatto che il calcolo del determinante è fortemente dipendente dal valore della diagonale principale, di conseguenza, aggiungere il termine λ ne impedisce l'avvicinamento a zero.

Questo tipo di regolarizzazione può essere introdotto anche esplicitando l'errore che si vuole minimizzare. Partendo da:

$$\begin{aligned} E &= E_D + \lambda E_w \\ &= \frac{1}{2} \sum_{i=1}^{N} \sum_{h=1}^{m} ((t_i^h - \sum_{l=1}^{\tilde{N}} \beta_{hl} g(\overline{x}_i \cdot \overline{w}_h))^2 + \frac{\lambda}{2} \sum_{l=1}^{M} |\beta_{hl}^2|) \end{aligned}$$

Il termine di regolarizzazione introdotto è λE_w che si traduce con la frazione con al numeratore il λ. Si può passare attraverso delle proprietà matematiche tali per cui minimizzando la norma del vettore β si ottiene una condizione che impedisce ai valori presenti all'interno del vettore di essere tanto distanti tra di loro.

Utilizzo di Singular Value Decomposition

Un altro approccio si basa sull'utilizzo della tecnica della *singular value decomposition* di H, che passa per:

$$H = USV^T$$

Dove U, S e V sono altre matrici con delle caratteristiche che servono per questo calcolo. In particolare, la prima e l'ultima matrice sono matrici rettangolari, mentre la matrice al centro è una matrice quadrata costituita da zeri, tranne sulla diagonale principale. Questo metodo è preferibile al precedente perchè elimina il passaggio di inversione della matrice, inoltre non serve introdurre un termine di regolarizzazione. Utilizzando questa strategia, la matrice H^+ è definita come:

$$H^+ = V\Sigma^+ U^T$$

Dove, anche qui, V, Σ e U sono altre particolari matrici, in particolare la matrice centrale viene calcolata in base alla matrice S. Anche in questo caso, possiamo avere a che fare con dei termini di regolarizzazione.

Capitolo 6

Reti Neurali Convoluzionali

6.1 Introduzione alle Reti Profonde

Con questo capitolo introduciamo qualche concetto riguardante il Deep Learning, in cui si collocano le così dette **Reti Neurali Profonde**. Con questo termine, indichiamo delle reti che hanno più di un livello di neuroni hidden, organizzati in livelli gerarchici. L'organizzazione in livelli gerarchici, permette alle reti di riutilizzare e condividere le informazioni acquisite. Ne consegue, però, che più si va in profondità, più la complessità dell'addestramento aumenta. Con questo tipo di strutture, però, diventa sempre più difficile rappresentare l'informazione all'interno della rete.

Una cosa interessante, però, è che con l'avvenimento di questo tipo di reti, non ci si deve più preoccupare della selezione delle features, infatti con queste reti abbiamo un apprendimento di tipo non supervisionato, basta dunque sottoporre alla rete i vari esempi che si vogliono imparare e la rete si preoccuperà di costruirsi una rappresentazione interna per poter rappresentare gli esempi al meglio. Questo porta ad un utilizzo più generale delle reti, che ora sono in grado anche di riconoscere eventuali caratteristiche latenti tra gli esempi presentati, che quindi possono essere usate in domini diversi.

Ma la profondità è solo uno dei fattori che influisce sulla complessità. Un'altra ragione la possiamo trovare nel numero di connessioni e di conseguenza nel numero di pesi tra neuroni.

Una domanda che ci si può porre è: ma quanto profonde? Tipicamente, le reti profonde presentano un numero di livelli hidden che va tra i 7 e i 50. Ovviamente, si possono avere delle reti più profonde, ma si ha un costo computazionale altissimo a fronte di un miglioramento delle performance molto basso.

La prima struttura di rete neurale profonda è quella della rete neurale convoluzionale, che viene spesso utilizzata per il processamento delle immagini.

6.2 Architettura della Rete

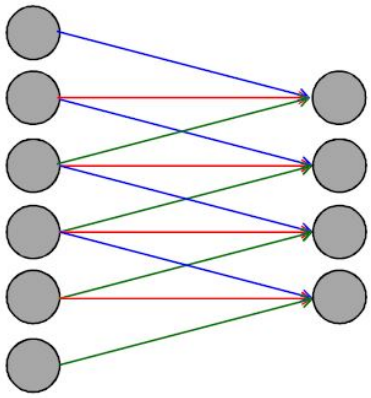


Figura 6.1: Rete Neurale Convoluzionale

Possiamo descrivere l'architettura di questa rete attraverso delle differenze con le reti MLP:

- **processamento locale**: i neuroni di un certo livello, sono connessi ad altri neuroni del livello precedente solo *localmente*, infatti, in Figura 6.1 possiamo vedere che, per esempio, il primo neurone di output è connesso soltanto ai primi tre neuroni del livello prima, questo porta ad una riduzione del numero di connessioni tra neuroni
- **pesi condivisi**: il valore dei pesi è condiviso per alcuni neuroni, infatti, nella Figura 6.1, i primi tre neuroni del livello hidden condividono i pesi con i primi tre neuroni del livello successivo (i pesi blu). In questo modo, neuroni differenti allo stesso livello, processano nello stesso modo parti differenti dell'input. Grazie a questo, si ha una riduzione del numero di connessioni tra neuroni

L'obiettivo di una rete di questo tipo è fare in modo che diverse parti delle reti si occupino di compiti diversi. Proprio per questo le reti CNN sono state create per il processamento di immagini. Osservando la Figura 6.2, possiamo vedere come una rete processa una immagine. In particolare bisogna porre l'attenzione sul fatto che esiste una matrice di neuroni di input che corrispondono ai diversi pixel dell'immagine che si sta prendendo in input. Alcune zone della rete si occupano di riconoscere delle caratteristiche solo su alcune parti dell'immagine,

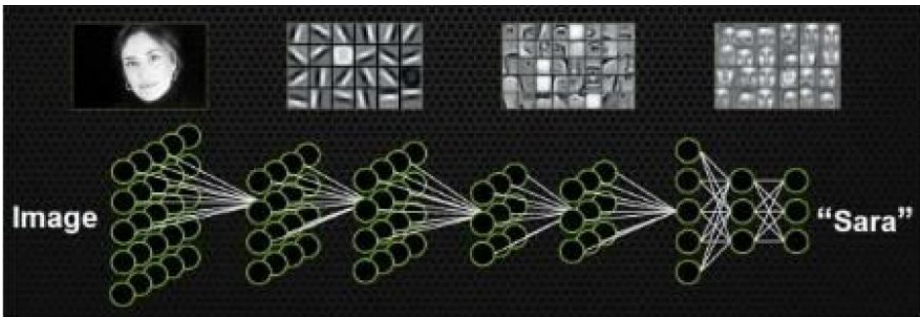


Figura 6.2: Processamento di un immagine di una Rete Neurale Convoluzionale

come per esempio un mucchio di pixel nell'angolo in alto a sinistra della foto. Tutte le diverse informazioni vengono propagate lungo la rete, fino ad arrivare ad un livello di output che fornisce il risultato.

6.3 Livello Convolutivo

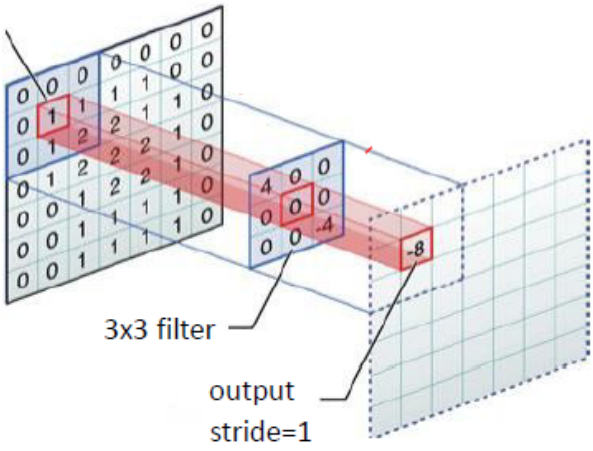


Figura 6.3: Livello Convolutivo

In Figura 6.3, possiamo vedere come funziona il livello convolutivo. Di fatto, esso lavora come un filtro, che prende i valori dal livello prima e li processa. Nella Figura 6.3, il filtro centrale effettua proprio questa operazione: possiamo vedere che la matrice 3×3 al centro viene utilizzata per effettuare una somma, cominciando dal valore in alto a sinistra, che potrebbe essere visto come il valore di un pixel di un immagine, ci si sposta in tutta la matrice, effettuando

moltiplicazioni e somme:

$$(4 * 0) + (0 * 0) + \cdots + (-4 * 2) = 8$$

Un altro esempio, possiamo vederlo in Figura 6.4, in cui ad ogni pixel vengono

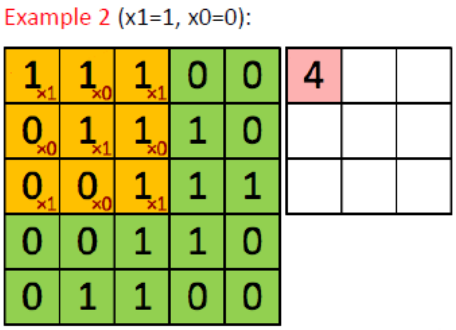


Figura 6.4: Altro esempio di Livello Convolutivo

applicati i diversi pesi di riferimento ($x0 = 0$ e $x1 = 1$), ed il risultato è la somma pesata.

6.3.1 Il Filtro e lo Stride

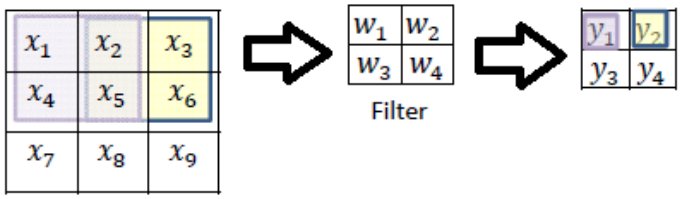


Figura 6.5: Utilizzo del Filtro e dello Stride

In Figura 6.5 possiamo vedere l'introduzione di un concetto, ossia quello di **stride**. Si tratta, infatti di un valore che determina lo spostamento di una finestra che viene spostata lungo l'input per determinare il risultato. Al centro è presente il **filtro**, ossia una matrice che determina i valori da moltiplicare. Per fare un esempio concreto, nella matrice finale, abbiamo che:

$$y_1 = w_1x_1 + w_2x_2 + w_3x_4 + w_4x_5$$

Una volta determinato il valore di y_1, per determinare y_2 spostiamo la finestra di una quantità quanto lo stride, nel nostro caso $stride = 1$, ottenendo:

$$y_2 = w_1x_2 + w_2x_3 + w_3x_5 + w_4x_6$$

0	0	0	0	0	0
0	x_1	x_2	x_3	x_4	0
0	x_5	x_6	x_7	x_8	0
0	x_9	x_{10}	x_{11}	x_{12}	0
0	x_{13}	x_{14}	x_{15}	x_{16}	0
0	0	0	0	0	0

Figura 6.6: Introduzione del Padding

e così via, spostando la finestra. Avendo una finestra, è possibile che questa, durante lo spostamento, possa uscire fuori dalla griglia dell'input. Per risolvere questo problema, viene introdotto del padding di grandezza giusta affinché la finestra non esca dalla griglia, come si può vedere in Figura 6.6.

6.4 Convoluzione 3D

Fino ad ora, abbiamo assunto che sia possibile rappresentare l'input in due dimensioni. Ma spesso non è così. Come si può vedere dalla Figura 6.7, il livello

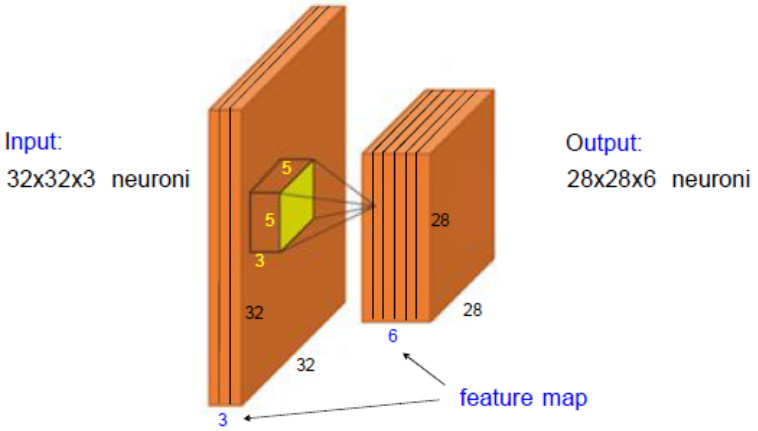


Figura 6.7: Livello Convolutivo 3D

di input è in tre dimensioni, come il livello di output. Per fare questa cosa, vengono predisposti diversi livelli di neuroni in ogni sezione. In particolare, ogni livello di neurone viene chiamato **feature map**. Nell'esempio abbiamo:

- 3 feature map (di dimensioni 32×32) nel livello di input
- 6 feature map (di dimensioni 28×28) nel livello di output

I pesi sono condivisi all'interno di una feature map ed i neuroni della stessa feature map processano parti differenti del livello di input allo stesso modo.

A questo punto, viene predisposto un nuovo livello di neuroni che si occupano di un task differente, detto **Pooling**. Questo task serve per comprimere una parte dell'informazione, generando delle features map di dimensioni minori. Si prenda ad esempio l'immagine di un panorama. Se gran parte dell'informazione è rappresentata dal cielo, è inutile processare quella parte di informazione con neuroni diversi, tanto l'informazione è la stessa. L'aggregazione può avvenire in modi differenti, utilizzando il valore massimo, oppure la media. In Figura 6.8

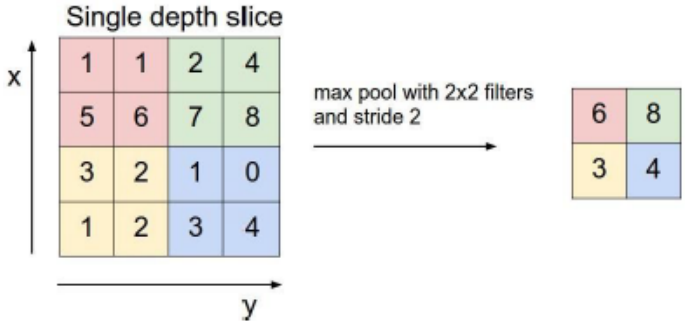


Figura 6.8: Livello Pooling

possiamo vedere un esempio di aggregazione, in cui viene mantenuto sempre il valore massimo.

Anche la funzione di attivazione cambia: vengono utilizzate delle funzioni chiamate **Relu**. Il problema della funzione sigmoide è nel gradiente, che tende ad annullarsi nei valori prossimi a 0 e ad esplodere con valori lontani da 0. Un esempio di funzione Relu[1], può essere vista in Figura 6.9. In particolare, questa è una funzione la cui derivata è 0 per valori negativi e 1 per valori positivi. Questo porta ad avere una *attivazione sparsa*, ossia che non tutti i neuroni vengono attivati, in base alle necessità, portando alcune componenti del gradiente a 0.

6.4.1 SoftMax e Cross-Entropy

Quando le CNN vengono utilizzate come classificatori, spesso viene insierito come ultimissimo layer una funzione **soft max**. Essa consiste in s neuroni

[1]REctified Linear Unit

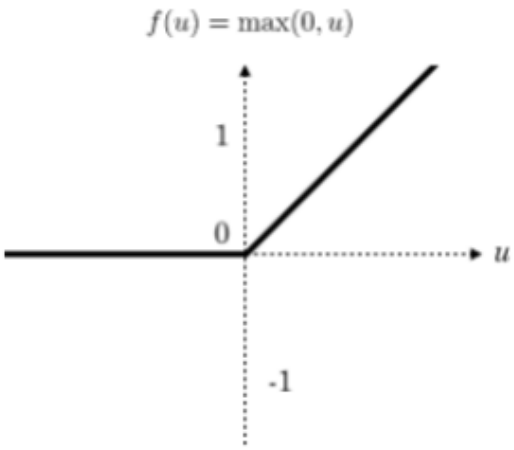


Figura 6.9: Una funzione Relu

completamente connessi al livello precedente. La funzione di attivazione per il neurone k è:

$$z_k = f(v_k) = \frac{e^{v_k}}{\sum_{c=1...s} e^{v_c}}$$

Il valore di z_k può essere visto come una probabilità che varia tra 0 e 1.

Inoltre, viene utilizzata la funzione di **Cross-Entropy** come funzione di loss (al posto del mean squared error utilizzato nelle MLP), che la rete neurale deve minimizzare. Questa funzione si basa sul fatto di avere due distribuzioni discrete, che simboleggiano ciò che vogliamo ottenere p e ciò che abbiamo ottenuto[2] q. Questa funzione misura quanto le due distribuzioni differiscono:

$$H(p, q) = -\sum_{v} p(v) \cdot log(q(v))$$

In Figura 6.10 è riportato un esempio di Rete Neurale Convoluzionale completa, con l'organizzazione a livelli appena trattata, in particolare, i livelli 6 e 7 sono completamente connessi, mentre il livello 8 è il livello in cui viene utilizzato SoftMax.

6.5 Training e Transfer Learning

Il training di una Rete Neurale Convoluzionale, può richiedere molto tempo. Inoltre, per poter essere addestrata correttamente, la rete ha bisogno di una

[2]per esempio dalla funzione SoftMax

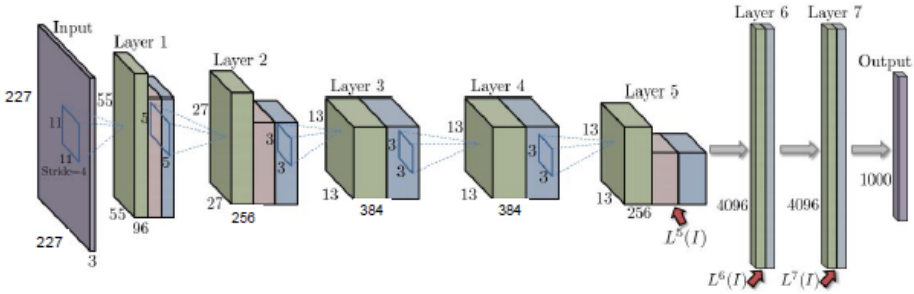


Figura 6.10: Rete Neurale Convoluzionale completa

quantità di dati e parametri non indifferente, ricadendo in una certa difficoltà di implementazione e debuging. In conclusione, il training di alcune CNN su particolari dataset molto complessi, può richiedere anche giorni, anche se vengono utilizzate delle tecniche di ottimizzazione utilizzando le GPU. Per poter venire venire incontro a queste esigenze, possono essere applicate delle tecniche che in qualche modo possono facilitare in alcuni casi l'addestramento di una rete:

- **fine-tuning**: si comincia da una rete già addestrata su un problema simile e
  1. si rimpiazza il livello di output con un livello softmax
  2. come valori iniziali dei pesi, vengono utilizzati quelli della rete già addestrata, tranne per i pesi sulle connessioni tra il penultimo livello e l'ultimo, che vengono inizializzati casualmente
  3. il training avviene sul nuovo dataset con la nuova rete
- **re-using features**: applicazione in cui vengono utilizzate le features prodotte da una certa rete neurale e si utilizzano quelle come features di un classificatore esterno

Capitolo 7

Self-Organizing Maps

7.1 Funzionamento della rete

Questo tipo di rete, implementa una forma di apprendimento **non supervisionato**. In questa configurazione, durante il training, non viene mostrata alla rete l'etichetta dell'esempio che gli stiamo proponendo, costringendola a trovare autonomamente delle somiglianze tra gli esempi e a classificarle di conseguenza. In Figura 7.1, possiamo vedere un esempio di Self-Organizing Map, che è forma-

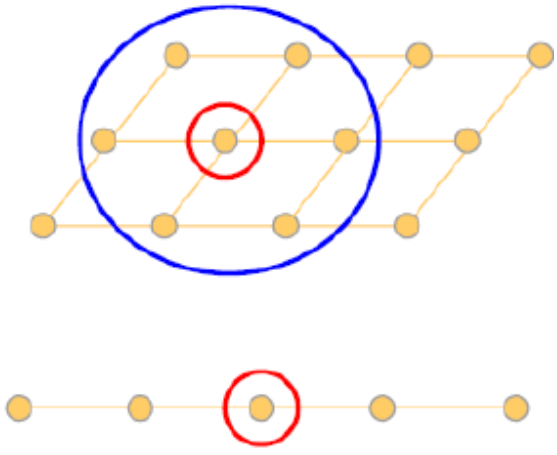


Figura 7.1: Esempio di SOM

ta da una struttura che può essere monodimensionale (sotto) o bidimensionale (sopra). Di fatto, la disposizione dei neuroni, permette ad essi di specializzarsi

in particolari aree dello spazio degli input e riconoscere gli input simili tra di loro. In Figura 7.2, possiamo vedere come i neuroni formano una griglia, ogni

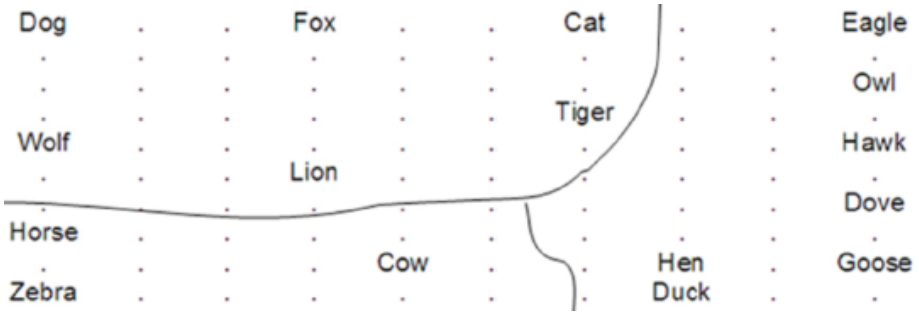


Figura 7.2: Disposizione dei Neuroni

neurone si specializza con una determinata classe e i neuroni vicini lo "aiutano" a riconoscere quella classe stessa.

7.2 Struttura della Rete

Queste reti, sono molto amate dagli psicologi, in quanto effettuano un lavoro molto simile a quello fatto dal cervello umano. Se si prende, infatti, il modello di apprendimento di un bambino, esso in base ai diversi stimoli, classifica le varie istanze sulla base di similarità tra di loro. Solo alla fine le raggruppa sotto una unica classe. A proposito di questo, si può introdurre il concetto di **prototipo**: sempre pensando all'apprendimento di un bambino, nel momento in cui deve impare che cosa sia un *cane*, viene generato all'interno della sua mente un prototipo di cane, ossia un animale con quattro zampe, peloso, con la coda, che abbaia.

L'input di una SOM è un vettore di dimensione n. Ogni elemento i del vettore, che può equivalere al valore di una particolare feature, oppure al valore di un particolare pixel, è connesso ad ogni neurone tramite un peso w_{ji}. L'attivazione del neurone j in risposta all'input x è dato da:

$$w_j^T x = w_{j1}x(1) + w_{j2}x(2) + \cdots + w_{jn}x(n) \tag{7.1}$$

Il neurone che presenta il valore più alto derivato dalla Formula 7.1 è detto **Best Matching Unit**. Nel momento in cui si presentano alla rete due input x_1 e x_2, essi vengono categorizzati allo stesso modo se hanno lo stesso BMU (oppure due BMU vicini). Alternativamente, vengono categorizzati in due modi differenti.

7.3 Fase di Learning

7.3.1 Calcolare il Best Matching Unit

Questo tipo di learning viene detto **competitivo**, in quanto durante questa fase tutti i neuroni competono per attivarsi in risposta ad un determinato input. Dato un un input:

$$x = \begin{pmatrix} x_1 \\ \dots \\ x_n \end{pmatrix}$$

si calcola il BMU con la Formula 7.1. Un modo alternativo di stabilire il BMU è considerare la distanza euclidea dall'input e prendere il neurone i che ha distanza minore. La distanza euclidea è definita nel modo seguente:

$$d = i : \forall j ||x - w_i|| \leq ||x - w_j|| \tag{7.2}$$

In parole, si sta prendendo in considerazione quel neurone i tale per cui, preso qualsiasi altro neurone j, la distanza tra l'input presentato ed il vettore pesi w_i (associato al neurone i) è minore della distanza tra l'input presentato ed il vettore pesi w_j (associato al neurone j).

Sotto la condizione che i vettori pesi abbiano la stessa norma, le condizioni 7.1 e 7.2 sono identiche.

7.3.2 Aggiornamento dei pesi

Una volta individuato il neurone i come BMU, si aggiornano i pesi nel seguente modo:

$$w_i(n+1) = w_i(n) + \eta(n)(x - w_i(n)) \tag{7.3}$$

Nella Formula 7.3 stiamo semplicemente dicendo che i pesi all'iterazione $n+1$ dipendono dai pesi all'iterazione n più un aggiornamento dato dalla differenza tra l'input e i pesi, tutto pesato dal learning rate, che è soggetto anche esso a decrescita durante le varie iterazioni, secondo una legge esponenziale:

$$\eta(n) = \eta_0 \cdot exp(-\frac{n}{\tau}) \tag{7.4}$$

La Formula 7.3 non è del tutto corretta, in quanto manca ancora un fattore, definito tramite il concetto di **neighborhood**. Questa funzione determina in che modo i neuroni vicini al BMU sono coinvolti nell'apprendimento. Questo valore, dati due neuroni i e j, è definito come:

$$h_{ji}(n) = exp(-\frac{d_{ji}^2}{2\sigma(n)^2}) \tag{7.5}$$

dove d_{ji} è la distanza tra i due neuroni.

Seguendo la formula, possiamo concludere che più un neurone j è vicino al neurone i, più il valore di h_{ji} sarà maggiore. Come si puo notare, la distanza è definita secondo una distribuzione gaussiana guidata dal valore σ, anche esso soggetto a decrescita come il learning rate:

$$\sigma(n) = \sigma_0 \cdot exp(-\frac{n}{\tau}) \tag{7.6}$$

Da precisare che, sia nella Formula 7.4 sia nella Formula 7.6, il valore di τ è fissato. La Formula 7.3 viene quindi corretta nel modo seguente:

$$w_j(n+1) = w_j(n) + \eta(n)h_{ji}(n)(x - w_j(n)) \tag{7.7}$$

aggiungendo la quantità $h_{ji}(n)$, che funge, quindi, da peso aggiuntivo, oltre al learning rate. Se si nota, c'è stato un cambio dei pedici: da i sono diventati j. Questo rende la formula molto più generale, in quanto, una volta individuato il BMU che corrisponde al neurone i, allora si aggiornano tutti i pesi di tutti gli altri neuroni j secondo la Formula 7.7, ed è proprio qui che entra in gioco il valore di neighborhood. Quando si arriverà a calcolare i pesi del neurone $j = i$, allora il valore di neighborhood sarà uguale a 1.

Grazie alla funzione di neighborhood, i neuroni vicini tra di loro assumono valori simili per i pesi e rispondono in modo simile ad input simili.

7.3.3 Regola di apprendimento

I passi da compiere sono i seguenti:

1. **inizializzazione**: inizializza i pesi in maniera casuale
2. **sampling**: seleziona un particolare input x
3. **similarity matching**: trova il BMU j tramite la Formula 7.1 o 7.2
4. **aggiornamento**: i pesi vengono aggiornati seguendo la Formula 7.7
5. **continua**: finché non si raggiunge una certa condizione di convergenza

Capitolo 8

Reti di Hopfield

8.1 Il Pattern Completion

Le Reti di Hopfield vengono utilizzate per recuperare la versione originale di un oggetto a partire da una sua versione corrotta: questo è il task di **pattern completion**. In Figura 8.1 possiamo vedere un esempio di questo task, in cui,

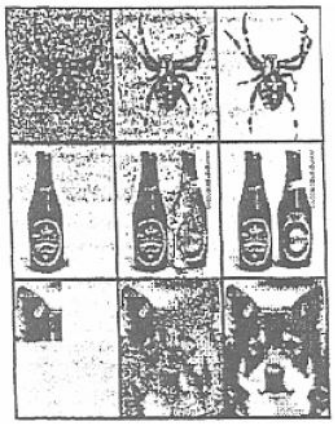


Figura 8.1: Esempio di Pattern Completion

partendo da sinistra, si possono ricostruire le varie versioni di una immagine, fino ad arrivare alla sua versione originale.

8.2 McCulloch-Pitts

Un esempio di rete di Hopfield è quella della rete di McCulloch-Pitts, di cui possiamo vederne un esempio in Figura 8.2. All'interno di questa rete, ogni

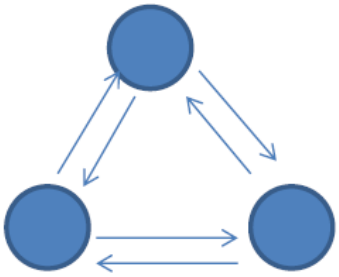


Figura 8.2: Esempio di Rete di McCulloch-Pitts

neurone può avere un solo stato: -1 o 1. In generale, date N unità, ogni neurone è connesso con ogni altro, tranne se stesso, attraverso dei pesi simmetrici, quindi $w_{ij} = w_{ji}$. L'attivazione di un neurone j all'iterazione n è data da:

$$y_j(n) = \phi(v_j(n)) \tag{8.1}$$

Dalla formula 8.1, possiamo notare diversi elementi, tra cui v_j che è l'input al neurone j definito come:

$$v_j = \sum_{i=0}^{N} w_{ji} y_i(n) \tag{8.2}$$

Quindi, nella Formula 8.2 possiamo notare che l'input al neurone j è dato dalla somma dei pesi che connettono il neurone j a tutti gli altri neuroni i della rete, moltiplicato per l'attivazione del neurone i stesso. Un altro elemento che possiamo notare è la funzione di attivazione $\phi(v_j)$ che determina l'uscita di un determinato neurone j a fronte di un certo input v_j all'iterazione n:

$$\phi(v_j)(n) = \begin{cases} 1 & \text{se } v_j(n) > 0 \\ -1 & \text{se } v_j(n) < 0 \\ \phi(v_j)(n-1) & \text{se } v_j(n) = 0 \end{cases}$$

I tre casi sono abbastanza semplici e autoesplicativi: c'è da porre soltanto l'attenzione sull'ultimo caso, che vuol dire semplicemente che se siamo nel caso $v_j(n) = 0$, allora l'uscita nel neurone j rimane invariata.

Esempio di esecuzione

Di seguito, riportiamo un semplice esempio di esecuzione di questa rete. Ricordiamo che il compito è partire da uno stato corrotto e ritornare allo stato

originale, di conseguenza, questa non è la fase di learning, in quanto in questo momento si presuppone che i pesi siano già stati appresi correttamente. Supponiamo di avere una rete addestrata con dei pesi in Figura 8.3.

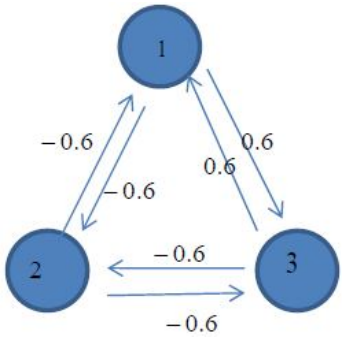


Figura 8.3: Esempio di Rete di McCulloch-Pitts

Partendo dallo stato

`[-1,-1,1]`

vediamo come avviene l'esecuzione.

Possiamo partire da qualsiasi neurone e vederne il relativo input della rete. Partendo dal neurone 1, abbiamo che:

$$v_1 = -0.6 \cdot \underbrace{(-1)}_{\text{stato neurone 2}} + 0.6 \cdot \underbrace{(1)}_{\text{stato neurone 3}} = 1.2$$
$$y_1 = \phi(v_1) = 1$$

Quindi lo stato cambia in `[1,-1,1]`. Adesso scegliamo il neurone 2 e allo stesso modo otteniamo:

$$v_2 = -1.2$$
$$y_2 = \phi(v_2) = -1$$

Quindi lo stato per il neurone 2 dovrebbe diventare -1, ma lo era già, quindi lo stato rimane `[1,-1,1]` Adesso scegliamo il neurone 3, ottenendo:

$$v_3 = 1.2$$
$$y_3 = \phi(v_3) = 1$$

Quindi lo stato per il neurone 3 dovrebbe diventare 1, ma lo era già, quindi lo stato rimane `[1,-1,1]`.

A questo punto, diciamo che lo stato che abbiamo raggiunto è detto **stato stabile**, poichè soddisfa la condizione $\forall i, y_i(n+1) = y_i(n)$, ossia che, riprovando ad effettuare una seconda volta tutti i calcoli effettuati in precedenza, non abbiamo più cambiamenti degli stati.

Proprietà Questo è uno dei tanti stati possibili di una rete: un altro stato potrebbe essere `[-1,1,-1]` e questo mette in risalto una proprietà importante, ossia che l'inverso di uno stato stabile è uno stato stabile.

8.3 Fase di Storage

Come detto all'inizio, le Reti di Hopfield sono utilizzate per memorizzare dell'informazione, di conseguenza, date le **memorie fondamentali**, ossia l'insieme dei valori che vogliamo memorizzare, dobbiamo organizzare i pesi in modo che lo stato che ci interessa sia uno stato stabile. Per trovare i pesi corretti, dobbiamo basarci sul **Principio di Hebb**, per il quale bisogna:

- *rafforzare* le connessioni tra due neuroni con lo stesso valore di attivazione
- *indebolire* le connessioni tra due neuroni con valore di attivazione opposto

Questo principio trova le sue fondamenta nelle neurobiologia, secondo la quale, all'interno di un cervello, le sinapsi che collegano due neuroni che si attivano nello stesso momento sono più forti rispetto a quelle che collegano due neuroni che hanno attivazione opposta.

Prendiamo ad esempio la rete in Figura 8.3 senza considerare i pesi. Vogliamo che lo stato stabile sia `[1,1,-1]`. Di conseguenza, vogliamo che:

$$w_{12} > 0$$
$$w_{23} < 0$$
$$w_{13} < 0$$

Più in generale, vogliamo che le unità con attivazione concorde abbiano i pesi che li collegano positivi, in caso di attivazione discorde, i pesi saranno negativi. Per trovare la configurazione giusta dei pesi, possiamo servirci di alcune formule molto semplici. Partendo dal caso base, quindi con una sola memoria fondamentale f, possiamo utilizzare la seguente formula:

$$w_{ji} = f(i) \cdot f(j) \tag{8.3}$$

ossia, per valorizzare il peso che connette il neurone i al neurone j bisogna moltiplicare i valori di attivazione dei due neuroni nella memoria fondamentale

f. Pensando al caso generale, invece, date M memorie fondamentali $f_1 \dots f_n$, abbiamo:

$$w_{ji} = \frac{1}{M} \sum_{k=1}^{M} f_k(i) \cdot f_k(j) \text{ se } j \neq i \tag{8.4}$$

È facile notare che la Formula 8.3 è una specializzazione della Formula 8.4: basta porre $M = 1$.

Esempio guidato

Vediamo ora come applicare questa formula in un esempio concreto, prendendo sempre la rete in Figura 8.3 senza considerare i pesi. Abbiamo la necessità di memorizzare due memorie fondamentali: `[1,-1,1]` e `[-1,1,-1]`: dobbiamo semplicemente applicare la Formula 8.4.

$$w_{12} = w_{21} = \frac{1}{2} \sum_{k=1}^{2} f_k(1) \cdot f_k(2) = \frac{1}{2}(1 \cdot (-1) + (-1) \cdot (1)) = -1$$

$$w_{23} = w_{32} = \frac{2}{3} \sum_{k=1}^{2} f_k(2) \cdot f_k(3) = \frac{1}{2}(-1 \cdot (1) + (1) \cdot (-1)) = -1$$

$$w_{13} = w_{31} = \frac{1}{2} \sum_{k=1}^{2} f_k(1) \cdot f_k(3) = \frac{1}{2}(1 \cdot 1 + (-1) \cdot (-1)) = 1$$

8.4 Teorema di Convergenza

Il Teorema di Convergenza serve per essere sicuri che a partire da una versione corrotta di una memoria, è sempre possibile raggiungere uno stato stabile. Ci sono un po' di cose da mettere in chiaro. Si parte dal fatto che bisogna tenere presente che lo spazio di ricerca dei vari stati è molto ampio, in quanto, date N memorie fondamentali ci sono 2^N possibili stati raggiungibili. Ad ogni stato è associata una particolare funzione chiamata **funzione energia**, che definisce la propensione a cambiare stato: ne consegue che più è grande il valore dell'energia, più siamo lontani da uno stato stabile. Il teorema dimostra che questa funzione decresce ad ogni passo, fino a raggiungere un punto in cui non può più variare. La funzione energia è definita nel modo seguente:

$$E = -\frac{1}{2} \sum_{i} \sum_{j} w_{ij} y_i y_j \tag{8.5}$$

Adesso consideriamo come cambia il valore di E all'attivazione della unità k-*esima*. Consideriamo il valore di E prima del cambio di y_k

$$E = -\frac{1}{2}\sum_i \sum_j w_{ij} y_i y_j \tag{8.6}$$

$$= -\frac{1}{2}(\sum_{i\neq k} \sum_{j\neq k} w_{ij} y_i y_j + 2\sum_{j\neq k} w_{kj} y_k y_j) \tag{8.7}$$

Si parte considerando la sommatoria nella Formula 8.6. Essa viene scomposta in altre due sommatorie, riportate nella Formula 8.7, in particolare, sempre seguendo la Formula 8.7, le prime due sommatorie considerano tutte le unità, tranne l'unità k che stiamo prendendo in considerazione, andando ad effettuare delle somme di prodotti tra i pesi ed i valori di attivazione, mentre la seconda sommatoria considera tutte le connessioni in entrata al neurone k. Adesso vediamo come cambia E nel momento in cui y_k cambia:

$$E' = -\frac{1}{2}(\sum_{i\neq k} \sum_{j\neq k} w_{ij} y_i y_j + 2\sum_{j\neq k} w_{kj} y'_k y_j)$$

che è la stessa Formula 8.7, considerando però il nuovo valore di attivazione y'_k. Effettuiamo una semplice differenza

$$E - E' = -\frac{1}{2}((2\sum_{j\neq k} w_{kj} y_k y_j - 2\sum_{j\neq k} w_{kj} y'_k y_j))$$

$$= -\sum_{j\neq k} w_{kj} y_j (y_k - y'_k) \tag{8.8}$$

Adesso il nostro compito è dimostrare che la quantità nella Formula 8.8 è una quantità sempre positiva. Distinguiamo due casi:

1. y_k passa da $+1$ a -1. Questo vuol dire che $y_k - y'_k > 0$ e $\sum_j w_{kj} y_j < 0$, quindi avremo che:

$$-\sum_{j\neq k} w_{kj} y_j (y_k - y'_k) > 0 \text{ e } E > E'$$

2. y_k passa da -1 a $+1$. Questo vuol dire che $y_k - y'_k < 0$ e $\sum_j w_{kj} y_j > 0$, quindi avremo che:

$$-\sum_{j\neq k} w_{kj} y_j (y_k - y'_k) > 0 \text{ e } E > E'$$

In entrambi i casi possiamo notare che $E > E'$ quindi in entrambi i casi avremo che $E - E' > 0$. Ad ogni iterazione l'energia diminuisce, ma dato che rimane sempre positiva, ad un certo punto si fermerà, di conseguenza non ci saranno più cambi di stato.

8.5 Considerazioni finali

8.5.1 I pro

Le qualità delle Reti di Hopfield sono molteplici. Riguardo il pattern completion, possiamo dire che esse sono in grado di generalizzare, in quanto dato un input simile a ciò che è stato memorizzato, riescono a recuperare l'informazione desiderata. In secondo luogo, possiamo dire che è resistente, in quanto se un neurone comincia a non funzionare, si potrebbe comunque raggiungere un output ragionevole.

Riprendendo il concetto di prototipo, è proprio grazie a questo tipo di reti che possiamo recuperarli. Il prototipo estratto può essere visto in ciò che la rete stessa impara, in quanto ogni neurone si specializza a riconoscere diverse parti di input. Facendo l'analogia con quanto detto all'inizio sull'apprendimento del bambino, potremmo immaginare che i diversi neuroni si specializzano nel riconoscere la coda, altri riconoscono il verso, altri riconoscono il pelo e così via: è solo nel momento in cui si mette tutto insieme che si ha la vera istanza del cane che si è presentata, avendo quindi l'information retrival.

Per utlimo, ma non per importanza, potremmo dire che l'algoritmo di apprendimento si basa sul principio di Hebb, che è biologicamente plausibile. Le altre reti viste fino ad ora utilizzano degli algoritmi di apprendimento che non trovano fondamenta nella biologia in materia di apprendimento di un vero cervello. Per farla breve, il cervello umano, per imparare, non utilizza un algoritmo di backpropagation.

8.5.2 I contro

Sfortunatamente, non tutti gli stati stabili sono delle memorie fondamentali memorizzate durante la fase di storage. in quanto, a partire da uno stato corrotto, è possibile raggiungere uno **stato spurio**, ossia uno stato stabile che non era una memoria fondamentale. Questo deriva da alcune proprietà degli stati stabili, per esempio il fatto che l'opposto di uno stato stabile, oppure una combinazione di stati stabili, è a sua volta uno stato stabile. Per abbassare la probabilità di cadere in uno stato spurio, bisogna dimensionare la rete in maniera corretta. La formula è la seguente:

$$M = \frac{N}{2logN} \tag{8.9}$$

dove M è il numero di memorie fondamentali ed N è il numero di unità della rete. In poche parole, la Formula 8.9 ci sta dicendo che a fronte di un certo numero di unità, si avrà un certo numero di memorie fondamentali memorizzabili con pochi errori.

Capitolo 9

Reti di Boltzmann

9.1 Dalle Reti di Hopfield

Abbiamo visto dalle Reti di Hopfield che durante la fase di information retrival, si può capitare in uno **stato spurio**, ossia uno stato stabile che non fa parte delle memorie fondamentali. Il problema possiamo rivederlo nella definizione

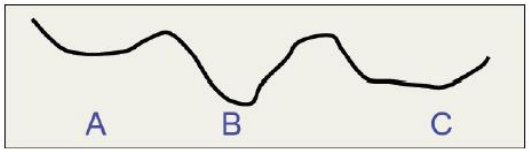


Figura 9.1: Minimi Locali

della funzione **energia**. Ad ogni passo ci si muove verso un minimo, in quanto la funzione decresce sempre. Il problema si trova nel momento in cui ci troviamo in uno stato stabile, che corrisponde ad un minimo della funzione energia: siamo sicuri che sia un minimo locale, ma non sappiamo se è effettivamente un minimo globale. Le Reti di Boltzmann sono state progettate per poter raggiungere degli stati che sono sempre memorie fondamentali, quindi non raggiungere mai stati spuri.

Le differenze con le Reti di Hopfield da mettere in luce sono le seguenti:

1. prevedere una fase di **unlearning**, cioè una fase in cui gli stati spuri vengono dimenticati
2. modificare l'architettura della Rete di Hopfield, prevedendo nuovi livelli

3. le unità diventano **stocastiche**, quindi non più deterministiche

9.2 Architettura della Rete

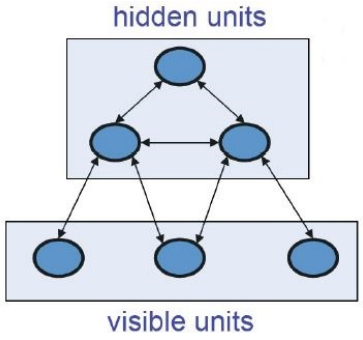


Figura 9.2: Architettura delle Rete di Boltzmann

In Figura 9.2 è presente un esempio di architettura della Rete di Boltzmann da cui si può vedere una prima differenza a livello architetturale con le Reti di Hopfield. Sono presenti due livelli: un livello **visible** che si occupa di ricevere l'input e fornire l'output della rete, ed un livello **hidden** che si occupa di interpretare[1] l'input ricevuto e generare un output da presentare.

9.3 Funzionamento della Rete

Dopo aver presentato un input alla rete, che corrisponde alla versione corrotta di una memoria, si passa alla fase in cui si devono far attivare i neuroni del livello hidden. A questo punto possiamo trovare una seconda differenza rispetto alle Reti di Hopfield: le unità sono stocastiche, alle quali vengono associate delle probabilità. In particolare, viene calcolata la probabilità di una certa unità s_i di attivarsi, quindi di avere il suo stato uguale a 1, come segue:

$$p(s_i = 1) = \frac{1}{1 + e^{-\Delta E_i / T}} \tag{9.1}$$

Nella Formula 9.1 possiamo notare l'introduzione di due quantità: la prima è la T che è definita come **temperatura** ed è un fattore esterno che fa tendere la rete a cambiare. La seconda quantità è ΔE_i che si definisce come segue:

$$\Delta E_i = E(s_i = 0) - E(s_i = 1) \tag{9.2}$$

[1] Il concetto di "interpretare l'input" è piuttosto vago, ma diventerà più chiaro in seguito

Quindi, la Formula 9.2 definisce la differenza tra due energie, la prima con l'unità $s_i = 0$ e la seconda con l'unità $s_i = 1$. Da ricordare che la formulazione dell'energia è la seguente:

$$E = -\sum_{k} s_k b_k - \sum_{k,j} s_k s_j w_{kj} \tag{9.3}$$

Bisogna far notare che la Formula 9.3 dell'energia è diversa dalla Formula 8.5. Nella 9.3, infatti, si tiene anche conto di un certo bias b_k che moltiplica l'attivazione di ogni unità hidden k. La seconda sommatoria, invece, prende tutte le possibili coppie del livello hidden e moltiplica i loro stati di attivazione per il peso che connette la coppia stessa, in questo modo, se almeno una delle due unità della coppia ha valore di attivazione uguale a 0, allora il prodotto sarà uguale a 0.

Esempio di esecuzione

Supponiamo di avere la Rete di Boltzmann in Figura 9.3, dove il livello visible è costituito dai due neuroni in basso ed il livello hidden è costituito dall'unico neurone in alto. Il nostro obiettivo è calcolare $p(h_1 = 1)$ dove h_1 è il neurone

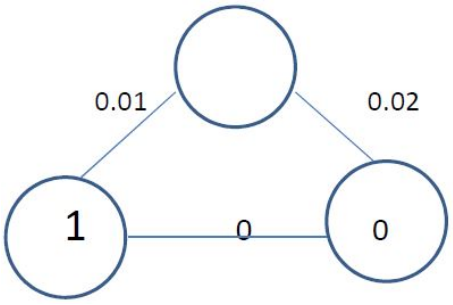


Figura 9.3: Esempio di esecuzione

hidden. Per farlo, dobbiamo dapprima ricorrere alla Formula 9.3, considerando i bias nulli, quindi la prima sommatoria sempre uguale a 0, tenendo quindi in considerazione solo la seconda:

$$\begin{aligned} E(s_i = 0) &= -[1 \cdot 0 \cdot 0 + 1 \cdot 0 \cdot 0.01 + 0 \cdot 0 \cdot 0.02] \\ &= -[0 + 0 + 0] \\ &= 0 \\ E(s_i = 1) &= -[1 \cdot 0 \cdot 0 + 1 \cdot 1 \cdot 0.01 + 0 \cdot 0 \cdot 0.02] \\ &= -[0 + 0.01 + 0] \\ &= -0.01 \end{aligned}$$

A questo punto applichiamo la Formula 9.2:

$$\begin{aligned}\Delta E_i &= E(s_i = 0) - E(s_i = 1) \\ &= 0 - (-0.01) \\ &= 0.01\end{aligned}$$

Ed ora, applichiamo la Formula 9.1, ipotizzando $T = 1$:

$$\begin{aligned}p(s_i = 1) &= \frac{1}{1 + e^{-\Delta E_i / T}} \\ &= \frac{1}{1 + e^{-0.01}} \\ &= \frac{1}{1 + 0.99} \\ &= \frac{1}{1.99} \\ &= 0.5025\end{aligned}$$

Per semplicità, assumiamo che dato che $p(s_i) > 0.5$, allora lo stato s_i deve essere cambiato al valore 1.

A questo punto, bisogna scansionare il livello visible, calcolando la $p(s_i = 1)$ per ogni neurone del livello, potrebbe essere che alcuni neuroni potrebbero cambiare stato. Questo cambiamento equivale al recupero della memoria fondamentale.

9.4 Fase di Learning

L'obiettivo in questo caso è cercare di massimizzare la probabilità che la rete vada verso uno stato che faceva parte del training set, in altre parole, si vuole forzare la rete a non andare verso stati spuri.

9.4.1 Restricted Boltzman Machine

Bisogna prima fare una digressione su questa variante delle Reti di Boltzmann, ossia la versione **restricted**, di cui possiamo vederne un esempio in Figura 9.4. Da questa immagine, possiamo capire la caratteristica fondamentale delle RBM, ossia che i neuroni dello stesso livello, non sono connessi tra di loro, infatti i neuroni del livello visible non hanno connessioni tra di loro, allo stesso modo, in senso orizzontale, i neuroni hidden non sono connessi tra di loro. Consideriamo la Figura 9.5. All'istante $t = 0$, in cui i pesi sono inizializzati random, viene presentata la memoria fondamentale che si vuole immagazzinare. A questo punto vengono attivati i neuroni hidden e di conseguenza viene cambiato il livello

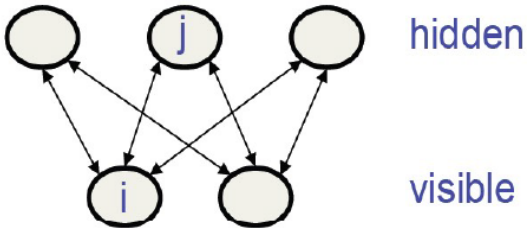


Figura 9.4: Esempio di Restricted Boltzman Machine

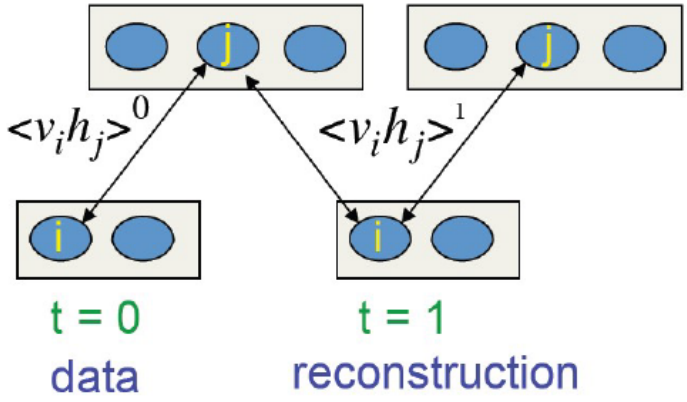


Figura 9.5: Algoritmo di Apprendimento delle RBM

visible, arrivando all'istante $t = 1$. A questo punto si calcola un particolare gap, definito nel modo seguente:

$$\Delta w_{ij} = \epsilon(< v_i h_j >^0 - < v_i h_j >^1) \tag{9.4}$$

Quindi, nel momento in cui si vuole aggiornare il peso tra il neurone i ed il neurone j bisogna calcolare la differenza tra il prodotto dello stato di attivazione del neurone visible i ed il neurone hidden j all'istante $t = 0$ e all'istante $t = 1$, moltiplicato per un certo ϵ definito a priori. Queste due iterazioni, bastano per definire i giusti pesi da attribuire a tutte le connessioni.

9.4.2 La rete in azione

Immaginiamo di dover costruire una rete che si preoccupa di ricostruire delle immagini. In Figura 9.6 possiamo vedere l'inizializzazione dei pesi di ogni unità nascosta in modo random. Alla fine dell'applicazione dell'algoritmo, arriviamo ad avere una situazione come quella in Figura 9.7. In bianco, vengono riportati i pesi con valore più alto, in nero quelli con valore più basso. La Figura 9.7 deve essere letta nel seguente modo: ad ogni quadratino corrisponde una unità

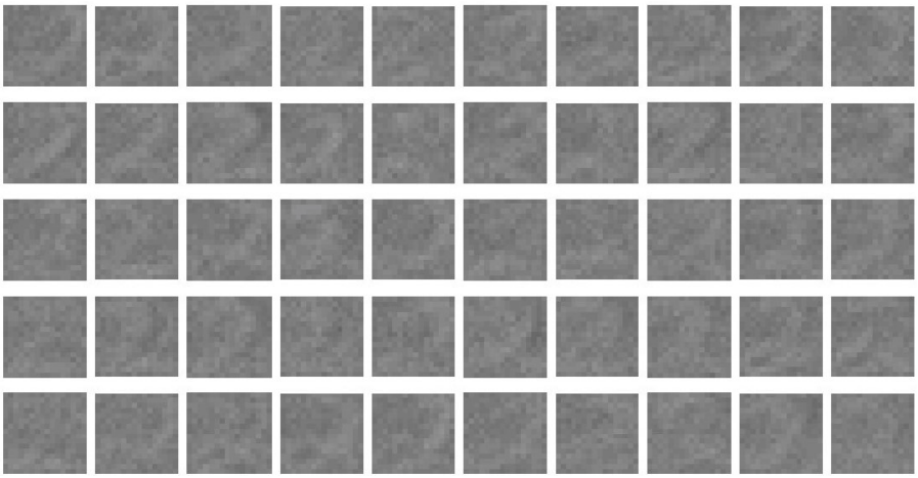


Figura 9.6: Assegnazione di pesi: punto di partenza

hidden a cui viene sottoposta l'immagine. Ogni unità hidden si è specializzata a riconoscere una parte dell'immagine (tramite dei pesi alti). Di conseguenza, se nell'input presentato è presente una determinata parte, questa viene riconosciuta dall'unità. Mettendo insieme tutte le attivazioni delle unità, si ricostruisce l'immagine vera e propria, quindi viene riportata allo stato originale.

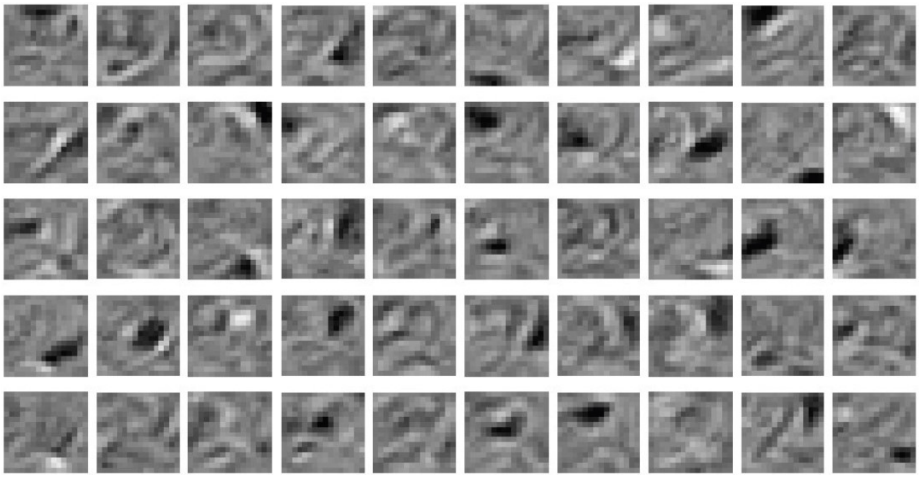


Figura 9.7: Assegnazione di pesi: punto di arrivo