Appunti di Reti Neurali & Deep Learning

Davide Giosa & Andrea Racca[1][2]

2020/2021[3]

[1]Basati sulle lezioni della Prof.ssa Rossella Cancelliere e del Prof Roberto Esposito
[2]Con la collaborazione di Zhongli Filippo Hu
[3]Basati sulle lezione dell'a.a 2019/2020 e aggiornati con le lezioni dell'a.a 2020/2021

Indice

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 è il punto fondamentale delle Reti Neurali, in quanto è proprio attraverso questi pesi che possiamo restituire un risultato piuttosto che un altro. Inoltre,

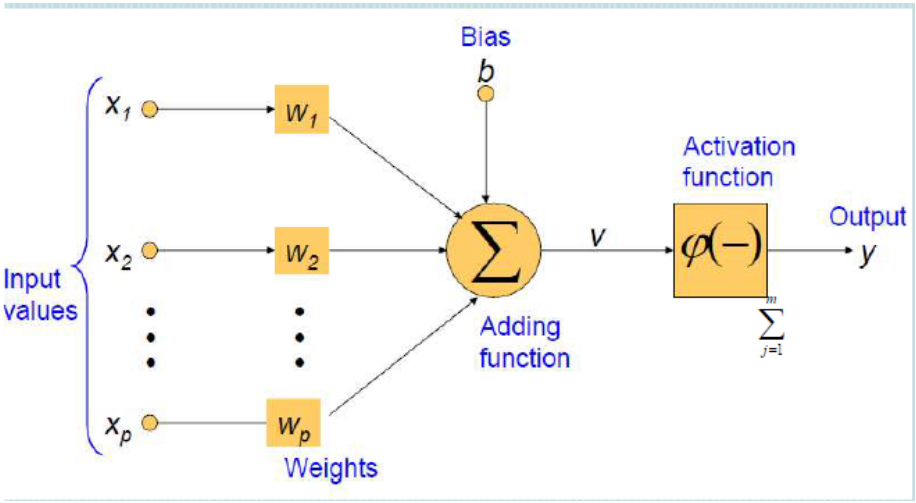


Figura 1.1: Schema di base di una Rete Neurale

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.

La rete neurale è un modello computazionale composto da neuroni artificiali, quindi una NN salva informazioni per poterle utilizzare in futuro. I benefici di una NN sono:

- **Non linearità**: fornisce una risposta elaborata da una funzione non lineare.
- **Input-output mapping**: hanno un ingresso ed un'uscita, ingresso è un insieme di neuroni su cui vi è codificata la sentenza che vogliamo sia imparata, l'uscita fornisce la risposta in base all'esempio.
- **Fault tolerance**: il task al quale viene addestrato è distribuito così vi è una buona tolleranza al danneggiamento. Vi sono tecniche di pruning che testano la rete per testare fino a quando si può prunare la rete senza riscontrare problemi.
- **Incremental training**: è possibile continuare l'addestramento quando vi sono nuovi dati disponibili.

Per riassumere possiamo dire che una NN è specificata da:

- **Un modello neuronale**
- **Un'architettura**: un set di neuroni con collegamenti, ogni link ha un peso.
- **Un algoritmo di addestramento**: usato per addestrare la NN, i pesi sono modificati in modo da modellare il learning task correttamente.

Bias: quantità che consente di "allontanare" da zero la somma pesata u_j: questo per far sì che con tutti i pesi 0 o i componenti in ingresso (x1, ...) la risposta del neurone non sia nulla

Esistono 3 diverse classi di architetture neurali:

- **Single layer feed forward**: grossi limiti quindi sorpassato.
- **Multi layer feed forward**: introduzione di un livello nascosto in quanto non riceve input e non produce output da e verso il mondo esterno.
- **Recurrent**: vi sono anche i layer nascosti e alcuni output vengono ripresi in input. Vi è un feedback loop.

2 tipi di NN:

1. Supervisionato: si ha un insieme di addestramento formato da input e output desiderato a partire da quell'input: usate per classificazione, riconoscimento, diagnostica, regressione
2. Non supervisionato: insiemi di addestramento non vengono associati alla risposta desiderata (non etichettati): impara differenti realizzazioni di input (es. imparano ad associare dalle caratteristiche degli input le istanze del problema). Usate per clustering

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

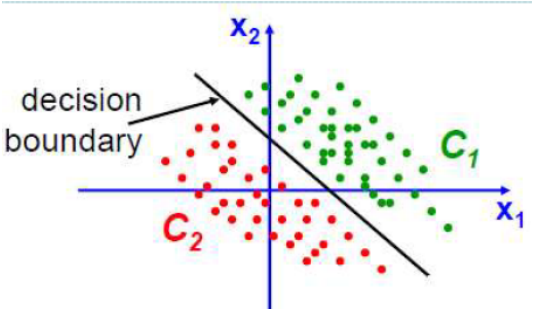


Figura 2.1: Esempio di problema linearmente separabile

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

[1] in italiano, **percettrone**

2.2 Struttura del Perceptron

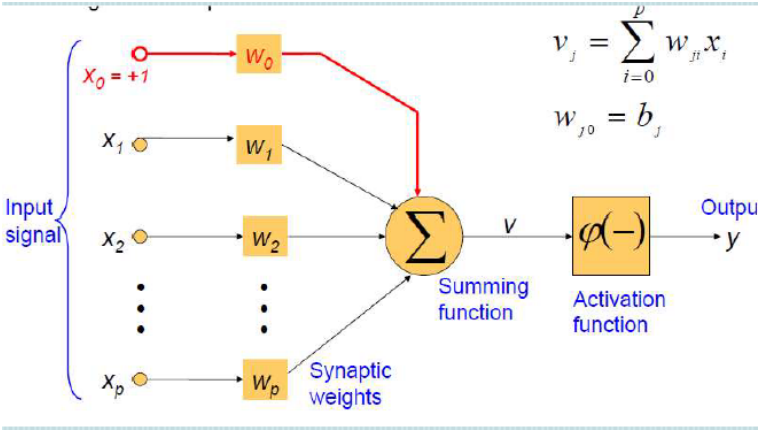


Figura 2.2: Struttura del Perceptron

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

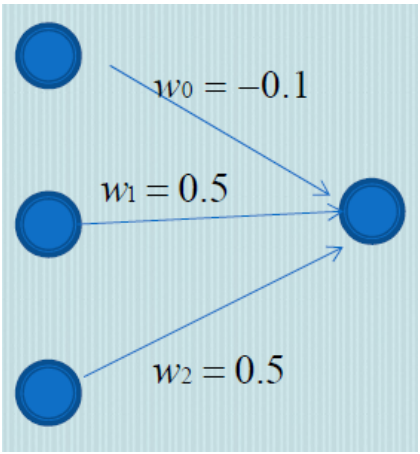


Figura 2.3: Problema dell'OR

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

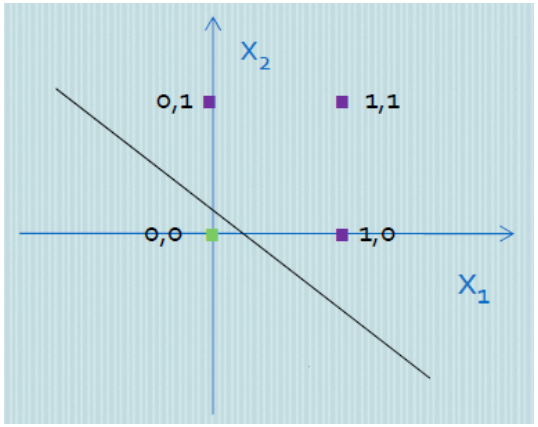


Figura 2.4: Soluzione grafica del problema dell'OR

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

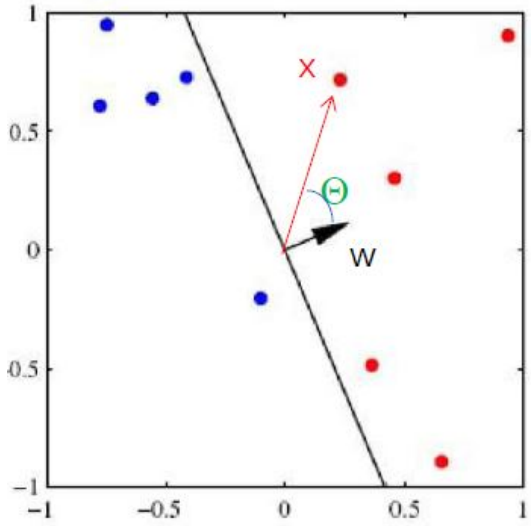


Figura 2.5: Aggiornamento dei pesi

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 ${w^*}^T x(0) \dots {w^*}^T x(k)$. Per la disequazione di Cauchy-Schwarz, abbiamo che:

$$||w^*||^2 ||w(k+1)||^2 \geq [{w^*}^T w(k+1)]^2$$

$$ossia$$

$$||w(k+1)||^2 \geq k^2 \alpha^2 / ||w^*||^2 \tag{2.5}$$

Quindi, con la Formula 2.7, abbiamo trovato il **lower bound** del numero delle iterazioni: abbiamo concluso la prima parte della dimostrazione.

Per la seconda parte, 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:

$$||w(k+1)||^2 \leq ||w(k)||^2 + ||x(k)||^2 \tag{2.6}$$

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

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

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:

$$||w(k+1)||^2 \leq k\beta \tag{2.7}$$

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. In particolare, svolgendo ancora dei calcoli, mettendo insieme la Formula 2.7 e la Formula 2.5, possiamo dire che:

$$k \leq \beta ||w^*||^2 / \alpha^2$$

che rappresenta l'effettivo limite oltre il quale il numero di iterazioni k non può crescere.

2.5 Limiti del Perceptron

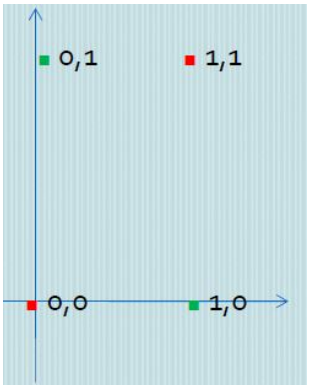


Figura 2.6: Problema dello XOR

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 (adaptive linear element)

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

A partire da un punto arbitrario nello spazio dei pesi, l'errore totale dipende dai parametri, quindi per riuscire a diminuirlo dobbiamo riuscire a modificare i pesi W e usiamo il gradiente che è il concetto di derivata su uno spazio multi-dimensionale. La derivata dice se una certa quantità diminuisce o aumenta in base ad un parametro. In questo caso abbiamo molti parametri W1, W2, ..., Wn. L'insieme delle derivate costituisce il vettore gradiente. Per aggiornare il vettore in modo opportuno definiamo che il peso all'iterazione Wk+1 è uguale a quello che era all'iterazione k meno η^* gradiente. η **è il learning rate.**

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.

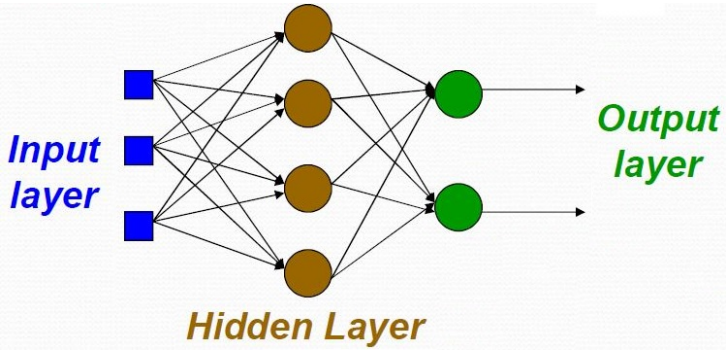


Figura 3.1: Struttura di una FFNN

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

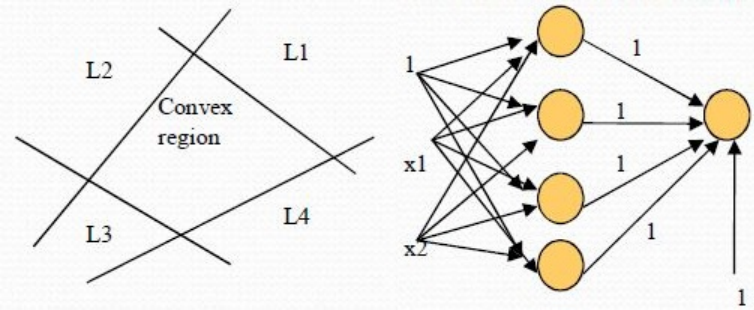


Figura 3.2: Creazione di regioni convesse

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.

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

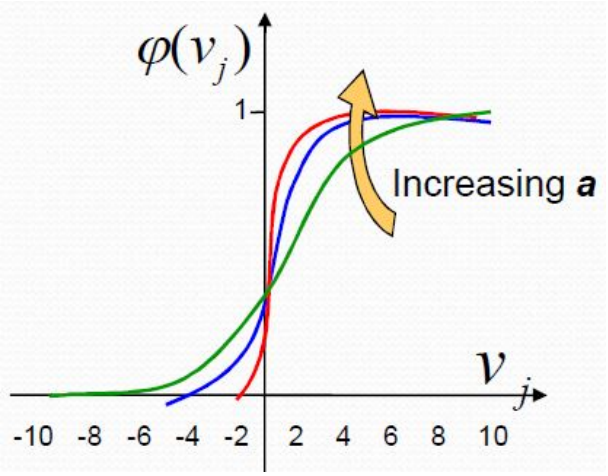


Figura 3.3: Funzione Sigmoide

- $v_j = \sum w_{ji} y_i$ definita come l'input del neurone j,
- y_i uscita del 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:

- **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

Metodi di apprendimento

L'apprendimento può avvenire in due modi differenti. Il primo è detto **by pattern** ed utilizza la formula trattata fino ad ora, cioè la 3.17. In particolare, i

pesi vengono inizializzati random e l'aggiustamento dei pesi avviene ogni volta prendendo un esempio diverso ed effettuando tutti i vari calcoli, finché non si raggiunge un certo criterio di stop.

Alternativamente, si può usare l'apprendimento detto **by epochs**. In questo caso, i pesi vengono aggiustati solo dopo che tutti gli esempi (o un sottoinsieme di essi) sono stati presi in considerazione. La formula di aggiustamento viene leggermente modificata, diventando

$$w_{ji} = w_{ji} + \sum_{\text{ogni esempio x}} \Delta w_{ji}^{x}$$

In parole, aggiusto i pesi solo alla fine dell'epoca, con un unico valore, dato dalla somma di tutti i Δw_{ji}^{x}. In questo caso, il criterio di stop potrebbe essere il raggiungimento del numero di epoche, che deve essere fissato prima di eseguire l'algoritmo di apprendimento.

Criteri di Stop

In generale, possono essere definiti alcuni criteri di stop:

- **errore quadratico medio totale**: si considera che la backpropagation ha raggiunto la convergenza nel momento in cui l'errore quadratico medio sia abbastanza piccolo[1]
- **criterio di generalizzazione**: dopo ogni epoca, si può testare la generalizzazione della rete: anche in questo caso si rientra in una fase di validation test, in cui una parte del training set non viene usato per l'aggiornamento dei pesi ma per testarne la generalizzazione. Se quest'ultima è accettabile, allora ci si può fermare con il training

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

[1] si può prendere un range [0.1, 0.01]

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.

Secondo il **Teorema di Apporssimazione Universale**, è comunque possibile approssimare una funzione utilizzando un solo livello di neuroni hidden: ovviamente la computazione risulterà parecchio inefficiente, dato che questo è solo un risultato teorico, infatti è sempre preferibile aggiungere dei livelli hidden in più.

Per maggiori info vedere la spiegazione del teorema di approssimazione universale
Per le FFNN esistono diverse applicazioni concrete come la classificazione di immagini, il riconoscimento facciale, il riconoscimento vocale, classificazione di oggetti, analisi di segnali. In questo periodo di emergenza sanitaria le FFNN possono aiutare con il riconoscimento delle immagini (es raggi ai polmoni) e distinguere polmoniti sars-cov2 o una polmonite di tipo batterico.

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

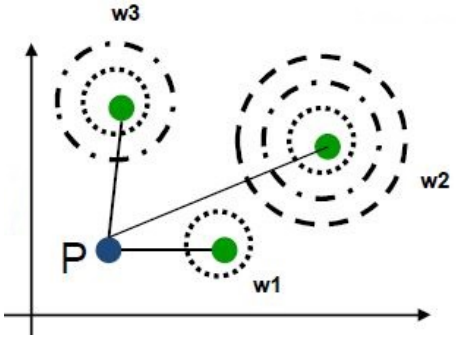


Figura 4.1: Vettori w

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

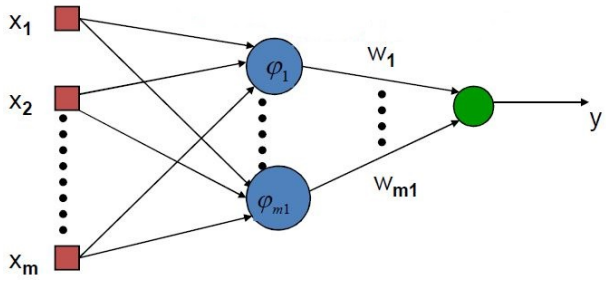


Figura 4.2: Struttura di una Rete Radiale

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:

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

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

L'ingresso di una funzione radiale è una distanza euclidea tra il vettore X che rappresenta il pattern in ingresso e il vettore t_1 che rappresenta il centro della funzione radiale. Ogni funzione radiale ha il suo centro t_i.
Le funzioni radiali $\phi_1, \phi_2, ..., \phi_{m1}$ valutano tutte una distanza funzione per ogni neurone nascosto del suo centro queste funzioni calcolate sulla distanza vengono lineatmente sovrapposte per formare l'uscita y e i coefficenti sono $w_1, w_2, ..., w_{m1}$.

4.2 Modello del Neurone Hidden

Vediamo ora la particolare struttura di un neurone hidden.

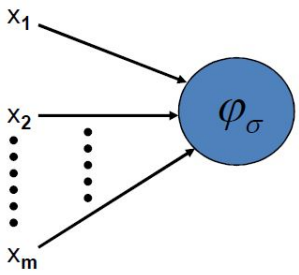


Figura 4.3: Neurone Hidden

In Figura 4.3, possiamo 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.

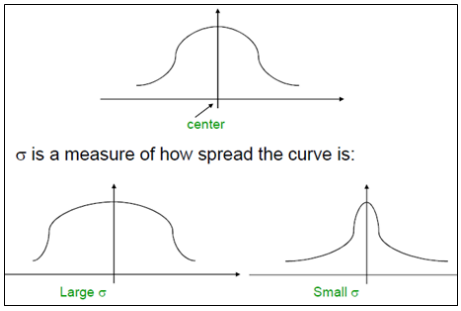


Figura 4.4: Esempio di larghezza della campana

σ determina la larghezza, il campo locale individuato dal neurone sarà molto più stretto se lo spread è piccolo, quindi il neurone risponderà con un segnale solo se il pattern x è molto vicino al centro. quando la funzione radiale è più aperta, possiamo avere una risposta anche per punti non vicini al centro. La simmetria rimane comunque radiale, il segnale si smorza nella stessa misura man mano che ci si allontana in una qualsiasi direzione dal centro.

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:

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

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.

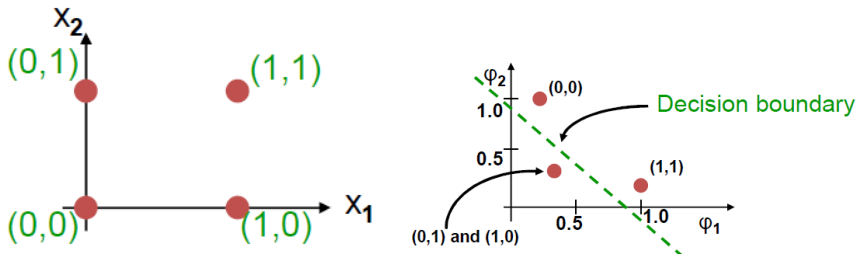
Ci sono diversi tipi di funzioni Φ, quella più usata è la gaussiana, che, per completezza, viene riportata di seguito:

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

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 4.5a.



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

Figura 4.5: 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 4.5b, 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 4.6 in cui

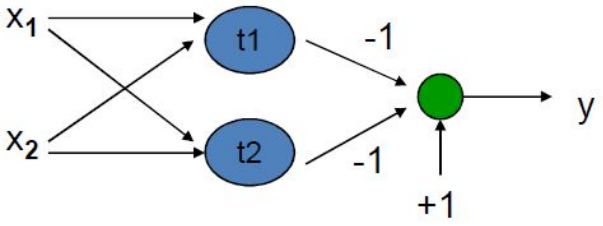


Figura 4.6: Piccola Rete Neurale Radiale

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:

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

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:

$$\sigma = \frac{distanza\ massima\ tra\ ogni\ coppia\ di\ centri}{\sqrt{numero\ di\ centri}} = \frac{d_{max}}{\sqrt{m_1}} \tag{4.5}$$

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:

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

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

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

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:

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

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

[1] ricordiamo che t_k sono i centri

3. **similarity matching**: trova l'indice k del centro più vicino a x con (trova il neurone con il centro più vicino a x):

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

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**:

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

- **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. RBF livello hidden non lineare ouput lineare, FFNN tutti i livelli solitamente non lineari.

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.

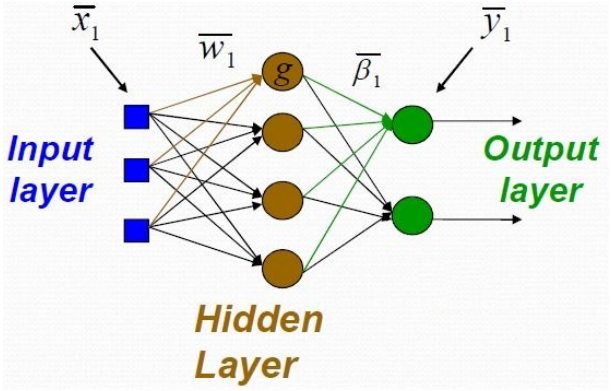


Figura 5.1: Modello della Rete Neurale

I pesi di primo livello sono quelli che si trovano tra il livello di 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 5.4, 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 quindi andremo a calcolare:

$$\begin{aligned} H \cdot \beta &\approx T \\ \beta &\approx H^{+} \cdot T \end{aligned} \tag{5.4}$$

[1]

5.2 Teoremi in ELM

Ci serviremo di due teoremi molto importanti.

[1] T: vettore target

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 $T = \left[t_1, \ldots, t_n\right]^T$

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

$$H^+ = (H^T H)^{-1} H^T \tag{5.5}$$

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

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

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:

- U è una matrice $N \times N$
- V è una matrice $M \times M$
- S è una matrice $N \times M$, costituita da zeri, tranne sulla diagonale principale

Questo metodo è preferibile al precedente perché elimina il passaggio di pseudoinversione 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, 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.

Nel teorema di approssimazione universale, abbiamo visto come un MLP con soli 2 livelli hidden fosse in grado di approssimare, con un errore arbitrario, una qualunque funzione limitata e costante. Dunque a cosa servono le reti profonde e perché vengono utilizzate?
Il teorema di approssimazione universale, nonostante sia importante a livello teorico, è meno utile a livello pratico in quanto non fornisce informazioni su come calcolare la soluzione. Tale soluzione, infatti, può essere molto difficile da

calcolare, con una complessità computazionale elevatissima. Le reti profonde permettono invece di calcolare una buona soluzione molto più efficientemente.

Inoltre, una rete profonda organizza i propri livelli in maniera gerarchica, la quale permette di selezionare le feature dei dati più importanti e di scartare i dettagli che non sono necessari, favorendo così il riuso e la condivisione delle informazioni. Uno dei più grossi vantaggi di una rete profonda, infatti, è che le caratteristiche importanti dei dati che permettono di risolvere un problema sono trovate in maniera automatica: questo va in opposizione a quello che succedeva con le reti tradizionali, in cui gli esperti effettuavano del preprocessing sui dati per selezionare le feature più importanti. Questo aspetto è noto come **representation learning** e denota quindi l'**unsupervised feature learning**. In tal senso, si dice che una rete neurale è costituita da un modello end-to-end, cioè che apprende una rappresentazione direttamente da raw data. Un apprendimento di questo tipo risulta molto corposo, che utilizza tantissimi dati e che quindi necessita di un hardware adeguato. Dunque, i punti importanti nel deep learning sono:

- Trovare il giusto stack di layer per costruire le architetture profonde.
- Trovare un modo per allenarle.

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

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 forte

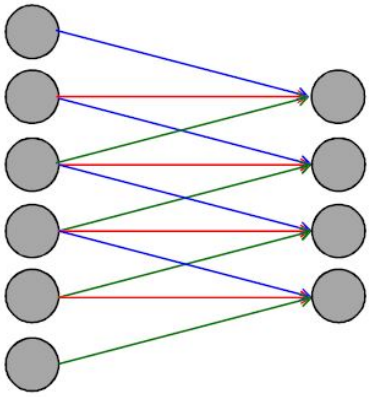


Figura 6.1: Rete Neurale Convoluzionale

riduzione del numero di connessioni tra neuroni, oltre al fatto che, in questo modo, i neuroni sono costretti a guardare solo particolari zone dell'immagine.

- **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 forte riduzione del numero di pesi

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

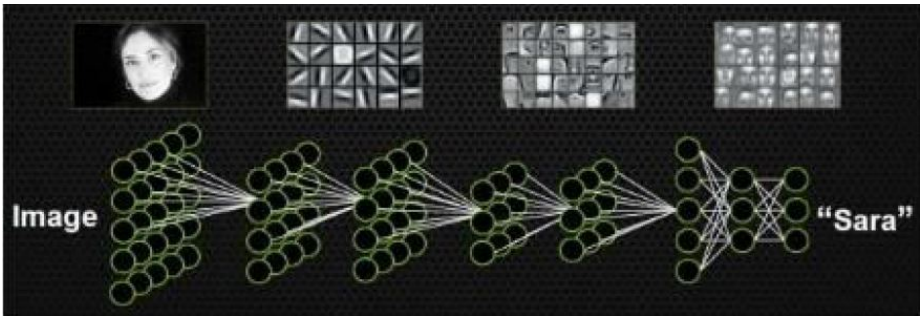


Figura 6.2: Processamento di un immagine di una Rete Neurale Convoluzionale

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, 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: è proprio qui che vengono posti dei livelli "stile multilayer perceptron", che si preoccupano di effettuare la vera e propria classificazione. Le reti convoluzionali hanno effettuato il lavoro di separare le features, la rete completamente connessa è in grado quindi di produrre l'output corretto.

6.3 Livello Convoluzionale

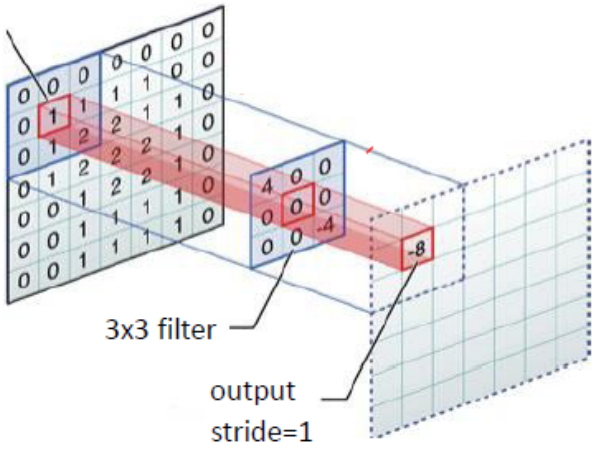


Figura 6.3: Livello Convolutivo

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

I pesi quando cambiano per effetto dell'addestramento, cambiano per tutti allo stesso modo, dopo un certo numero di epoche la maschera sarà diversa allo stesso

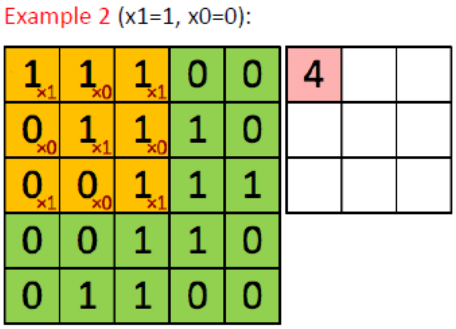


Figura 6.4: Altro esempio di Livello Convolutivo

modo per tutti i neuroni di quel livello. Comi si fa a calcolare caratteristiche diverse con filtri diversi? Si fa mettendo più livelli convoluzionali, ogni livello ha neuroni che gestiscono maschere diverso ovvero valori numerici diversi.

Questa parte viene molto richiesta negli scritti.

6.3.1 Il Filtro e lo Stride

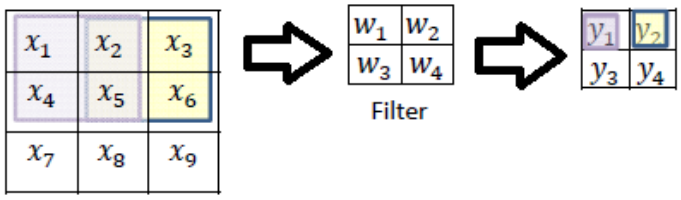


Figura 6.5: Utilizzo del Filtro e dello Stride

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 la 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:

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

e così via, spostando la finestra. Avendo una finestra, è possibile che questa,

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

durante lo spostamento, possa uscire fuori dalla griglia dell'input. Per risolvere questo problema, viene introdotto del padding (un bordo) di grandezza giusta affinché la finestra non esca dalla griglia, come si può vedere in Figura 6.6.

Come si può vedere dagli esempi e da come è già stato detto le reti convoluzionali sono adatte per l'utilizzo sulle immagini.

6.4 Convoluzione 3D

6.4.1 Feature Map

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

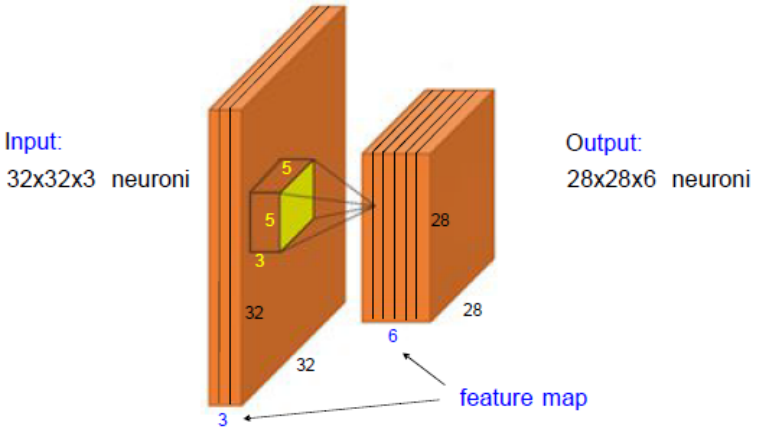


Figura 6.7: Livello Convolutivo 3D

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

L'idea è quella di utilizzare una particolare features map per apprendere una particolare features dell'input: il concetto è che questi livelli agiscano proprio come dei filtri che vanno ad applicarsi su una parte dell'immagine pesando le componenti con dei vettori di pesi locali e condivisi. Ogni livello, va ad addestrarsi su una caratteristica di particolare interesse all'interno dell'immagine. In questo, ci vengono incontro anche le connessioni locali, che fanno vedere a certi neuroni solo alcune parti, in modo da facilitare la specializzazione solo in una particolare zona. Più features map insieme, compongono un **tensore**, di cui possiamo vedere un esempio in Figura 6.8.

Quindi possiamo notare che si passa da una matrice 32x32 ad un blocco di 3 livelli, quindi si passa a 32x32x3 quindi il livello convoluzionale seguente non prende più una matrice piatta (bidimensionale) ma deve scavare in profondità al

parallelepipedo prendendo un blocchetto di dimensione 5x5x3 ovvero 75 valore di neuroni, quindi vi è un filtro con 75 pesi per confluire a creare un valore nel successivo livello convoluzionale.

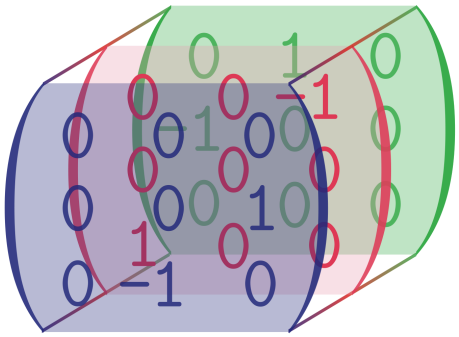


Figura 6.8: Esempio di Tensore

6.4.2 Pooling

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. Questa è una caratteristica abbastanza ricorrente: si parte con delle features map di una certa dimensione che vanno via via diminuendo. 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 (**max-pooling**), oppure la media (**averege-pooling**).

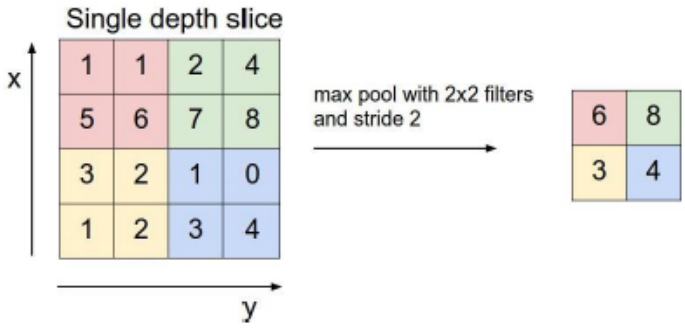


Figura 6.9: Livello Pooling

In Figura 6.10 possiamo vedere un esempio di aggregazione, in cui viene mantenuto sempre il valore massimo.

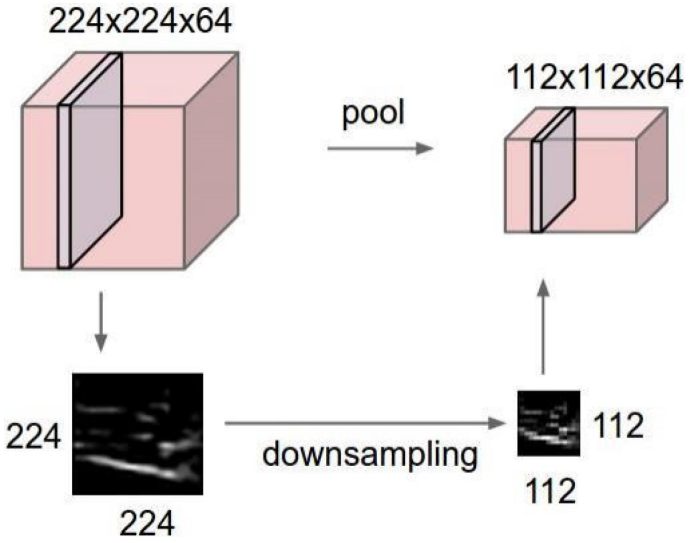


Figura 6.10: Esempio di Pooling

6.4.3 Funzione di attivazione

Il problema del gradiente

Abbiamo visto nelle precedenti sezioni, che come funzione di attivazione viene utilizzata una funzione sigmoide. Questa funzione tende ad avere problemi relativi alla propagazione del gradiente, cioè tendono a causarne l'esplosione o l'annullamento. Questo perché in un'ampia parte del loro dominio hanno un andamento "quasi piatto", cioè con un tasso di incremento molto basso (derivata prima tendente a 0), in particolare nelle zone asintotiche in cui la sigmoide tende al valore 0 ed al valore 1. Infatti, in tutti gli anni in cui le reti profonde ancora non esistevano, si tendeva a lavorare nella zona centrale di ogni sigmoide, precauzione non più sufficiente nel momento in cui il numero di livelli hidden andava sopra un certo numero.

Funzione Relu

Un esempio di funzione Relu[1], può essere vista in Figura 6.11. Come si può notare, essa non è derivabile: per ovvaire a questo, vengono definite due derivate ad-hoc, in particolare, derivata 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

[1] REctified Linear Unit

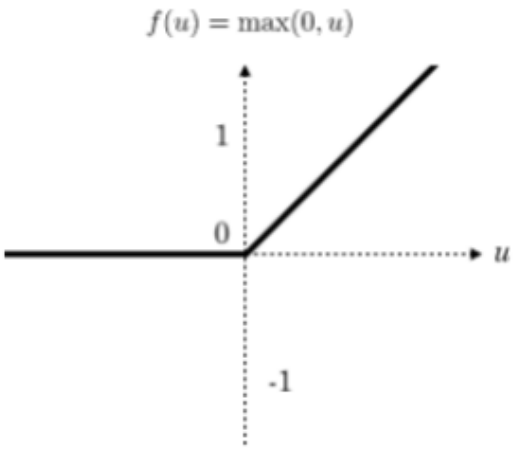


Figura 6.11: Una funzione Relu

a 0, quindi a non contribuirne al calcolo nel momento in cui ricevono in ingresso un valore negativo. Questo comportamento sembra funzionare ed introduce un concetto molto importante, ossia quello del **pruning automatico**, che si traduce in una maggiore robustezza della rete stessa. Visto che non si ha indicazione su quanti neuroni (o quanti livelli) allocare in un'architettura di rete neurale, normalmente si tende ad "abbondare" mettendo alcuni componenti che risultano all'atto pratico essere inutili. In questo modo la rete ha la possibilità di auto-prunarsi da sola, cioè mandando a 0 un certo numero di componenti a seconda delle sue esigenze durante l'addestramento.

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

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

Il valore di z_k può essere visto come una probabilità che varia tra 0 e 1, assegnata ai diversi neuroni di uscita, ciascuno dei quali descrive una classe.

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 discre-

te, 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 6.12 è riportato un esempio di Rete Neurale Convoluzionale completa,

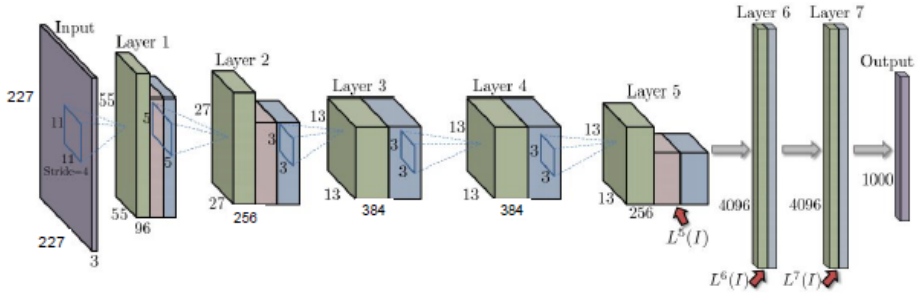


Figura 6.12: Rete Neurale Convoluzionale completa

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.

6.6 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 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

[2]per esempio dalla funzione SoftMax

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

SOM: Self Organizing Maps

Tecnologia di rete neurale, sviluppata a fine anni '60 che si è ispirata al cervello umano. Gli stimoli vengono ricevuti dalle cortecce e vengono organizzati in modo topologico.

7.1 Kohonen Maps

Non implementano fedelmente la rappresentazione del cervello umano ma sono un buon strumento computazionale.

7.1.1 Obiettivi

Si parla di modelli non supervisionati, quindi il task di apprendimento non è la classificazione ne la regressione. è un task di descrizione dei dati, le mappe sono monodimensionali o bidimensionali. Una volta addestrata, gli stimoli arrivati dal dataset vengono mappati in modo topologico.

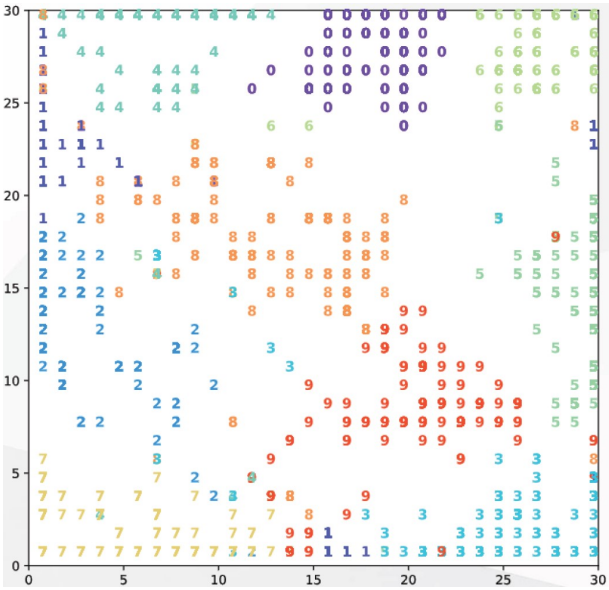


Figura 7.1: Mapping dataset

In Figura 7.1 si può vedere come viene rappresentata una mappa. Per ogni esempio del dataset si sottopone alla mappa l'esempio che eccita un neurone e viene etichettato con il numero corrispondente al suo eccitamento. La mappa riesce ad organizzare i dati in aree omogenee (i 9 sono raggruppati e sono vicini agli 8 e così via).

7.1.2 Principi

Gli algoritmi SOM sono basati su 3 principi:

- **Competizione**: ogni volta che arriva uno stimolo tutti i neuroni competono per diventare la best matching unit (BMU) ovvero il neurone che meglio si adatta a quello stimolo.
- **Cooperazione**: una volta individuato il BMU, tutti i neuroni che sono nel suo intorno diventano un po' più simili (il neurone vincente determina la posizione spaziale dei neuroni vicini topologici che verranno eccitati).
- **Adattamento sinaptico**: la BMU e tutto il suo intorno vengono aggiornati (i pesi) per renderli un po' più vicini all'esempio appena processato.

7.2 Processo competitivo

$$X = [x_1, x_2, ..., x_m]^T \in \mathbb{R}^m \tag{7.1}$$

Per il resto del corso con X si indicherà il vettore colonna con ^{T} si indica la trasposta, quindi lo abbiamo scritto come vettore riga ma si tratta di vettore colonna. Consideriamo una SOM con l neuroni con ogni unità j nella mappa, a questa unità è associato un vettore della stessa dimensione del vettore di input. I neuroni sono disposti all'interno della mappa secondo 2 topologie come si vede in Figura 7.2.

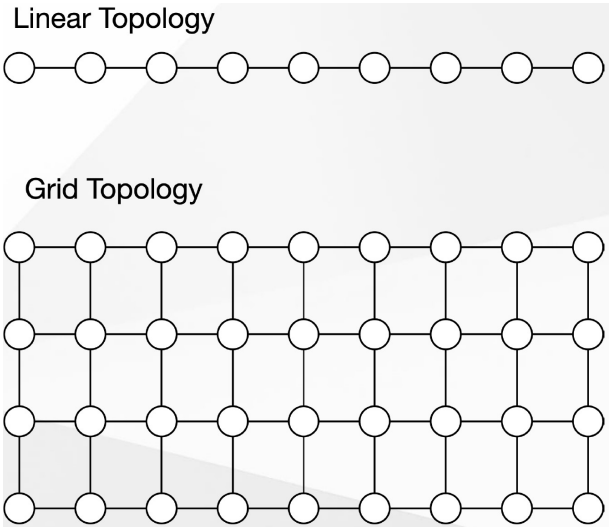


Figura 7.2: Topologie mappe

Le topologie più comuni sono quelle monodimensionali o bidimensionali, si possono aumentare le dimensioni ma non è molto comune come pratica. Nella **topologia monodimensionale** possiamo pensare i neuroni come rappresentati in un vettore, e i rapporti di vicinanza sono molto semplici essendo lineari. La **topologia bidimensionale** ogni neurone ha più vicini, ovvero sia i neuroni prima che dopo sia i neuroni sopra e sotto. I rapporti di vicinanza sono molto importanti in quanto la BMU andrà ad attivare tutti i neuroni vicini.

(La topologia si occupa di studiare le relazioni di vicinanza tra gli oggetti).

L'input viene confrontato con tutti i vettori dei pesi di tutti i neuroni. I neuroni si attivano con una forza pari al loro prodotto scalare $w_j^T X$, quindi più il prodotto scalare è alto e più il neurone è attivato. Dopo guardiamo tutti i valori e scegliamo la BMU.

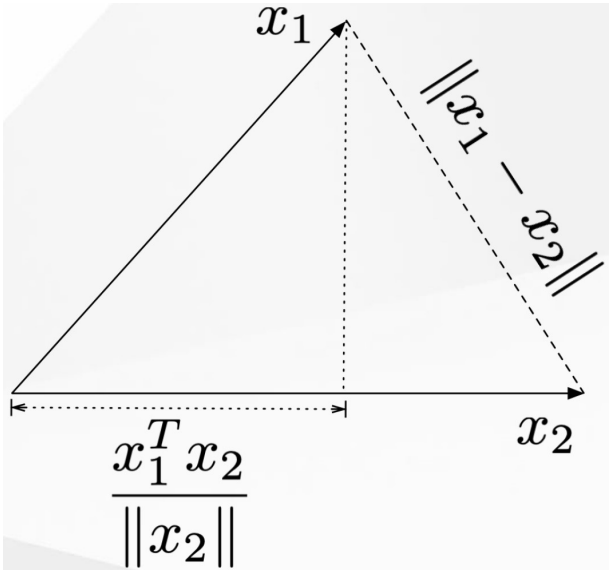


Figura 7.3: Relazione tra prodotto scalare e norma euclidea

Figura 7.3 possiamo vedere che la distanza euclidea $||x_1 - x_2||$ è il segmento che vediamo disegnato tra x_1 e x_2 mentre il prodotto scalare tra x_1 e x_2 è la proiezione di x_1 su x_2 (altezza triangolo). Da questa figura si evince che se l'angolo in basso a sinistra decresce la norma 2 diminuisce anch'essa mentre il prodotto scalare cresce, quindi vi è una relazione inversa tra uno e l'altra.

Se assumiamo che tutti i w_j hanno la stessa lunghezza allora è vero che w con il più grande dot product allora possiamo dire che è uguale a selezionare il w che minimizza la distanza euclidea: $i(x) = arg\ min_j||x - w_j||$. Questo è vero perché la norma a 2 è uguale ad un costante C per tutti i vettori W_j. Quindi possiamo formalizzare con:

$$\forall j : ||w_j|| = C,\ per\ una\ qualche\ costante\ C : \tag{7.2}$$

$$||x - w|| = (x - w)^T(x - w) = ||x||^2 + C^2 - 2w^T x \tag{7.3}$$

Quindi, massimizzare $w_j^T x$ è la stessa cosa di minimizzare $||x - w_j||$.

Per norma due si intende la distanza euclidea

Quando si cerca la BMU si può usare sia il prodotto scalare che la norma 2.

7.3 Processo cooperativo

Nel momento in cui si trova la BMU non si attiva un unico neurone ma anche tutti quelli vicini. Per implementare questa cooperazione bisogna trovare una funzione (per calcolare la distanza topologica) $h_{j,i(x)}$ con $i(x)$ indichiamo l'indice della posizione del neurone. Questa funzione raggiunge il suo massimo in corrispondenza della BMU e poi decresce rapidamente man mano che ci allontaniamo. Introduciamo il concetto di distanza laterale: $d_{j,i}$ tra i neuroni i e j, ad esempio possiamo scrivere $d_{j,i} = ||r_i - r_j||$ dove r_i denota la posizione del neurone nello spazio di output. Un modo tipico di come viene definito $h_{j,i(x)} = exp(-\frac{d^2_{j,i(x)}}{2\sigma^2})$.

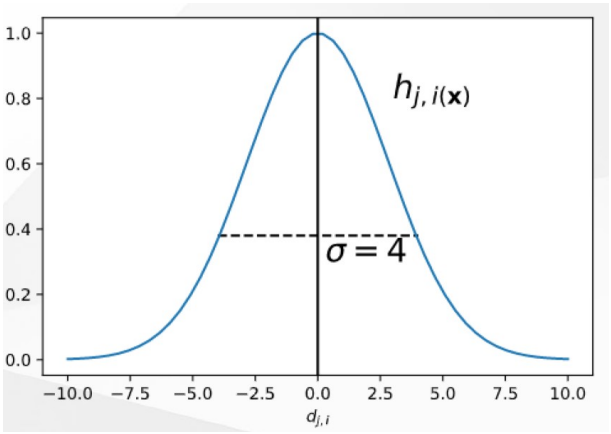


Figura 7.4: Rappresentazione di funzione topologica

In Figura 7.4 possiamo vedere la rappresentazione della funzione della vicinanza topologica, che ha una forma a campana e un parametro σ che indica con che velocità l'eccitazione dei neuroni si attenua, più è stretta (basso σ) più i vicini verranno ignorati man mano che ci allontaniamo dal BMU. Durante la fase di apprendimento è utile modificare σ in quanto si comincia con un raggio ampio e man mano si va a diminuire. Questo viene fatto per avere una maggiore convergenza, un modo per definire il cambiamento è: $\sigma(n) = \sigma_0 e^{-n/\tau_1}$, e di conseguenza $h_{j,i}$ definita come:

$$h_{j,i(x)}(n) = \exp(-\frac{d^2_{j,i(x)}}{2\sigma^2(n)}) \tag{7.4}$$

7.4 Processo adattativo

La regola di update (aggiornamento dei pesi) per il neurone j è:

$$\boldsymbol{w}_j(n+1) = \boldsymbol{w}_j(n) + \eta(n) h_{j,i(x)}(n)(\boldsymbol{x} - \boldsymbol{w}_j(n)) \tag{7.5}$$

dove $\eta(n)$ è il learning rate al tempo n.

Notare che

- Se $\eta(n) = 1$ *e* $h_{j,i(x)}(n) = 1$ *allora* $w_j(n+1) = x$
- In tutti gli altri casi l'aggiornamento fa si che il neurone aumenti la sua somiglianza con l'esempio, ma la forza dell'aggiornamento diminuisce con n e con la distanza dal neurone $i(x)$

Learning rate adattativo
La funzione $\eta(n)$ è definita (molto simile a $\sigma(n)$): $\eta(n) = \eta_0 e^{-n/\tau_2}$. Dipende da 2 parametri η_0 learning rate iniziale e un parametro τ_2 che determina con che velocità questo parametro viene decrementato.

Due fasi del processo adattativo: le SOM vengono inizializzate utilizzando dei pesi molto vicini a zero (si usa una gaussiana) poi man mano che vengono estratti gli esempi si aggiornano i pesi, ogni volta che viene sottoposto un esempio n viene incrementato e la rete comincia a prendere forma.

1. **Fase di auto organizzazione o di ordinamento**: fase in cui la struttura topologica viene inizializzata, inizialmente la struttura topologica è sbagliata, in questa fase la struttura topologica viene un minimo ordinata e strutturata. Come valori indicativi possiamo mettere $\eta_0 = 0.1$ e $\tau_2 = 1000$ ovvero il learning rate è pari a 0.1 (di solito si fa partire da 0.1 e si fa scendere al massimo fino a 0.01) e il numero di iterazioni deve essere almeno pari a 1000.
   Una buona partenza per τ_1 è: $\tau_1 = \frac{1000}{\log \sigma_0}$
2. **Fase di convergenza**: la struttura viene raffinata e vengono migliorati alcuni difetti topologici. In questa fase migliora di molto lo spazio di input. Per accuratezza statistica il learning rate deve essere piccolo ma mantenuto comunque superiore a 0.01. Vi è comunque il rischio che la SOM si blocchi in uno stato metastabile, ovvero in uno stato stabile e non si muove più ma non è nello stato ottimale (manca pochissimo allo stato ottimale)
   La funzione di neighborhood $h_{j,i(x)}$ dovrebbe contenere solo i vicini più vicini dei neuroni vincenti e ridurre il numero dei vicini ad uno o zero. In poche parole se la BMU si attiva vogliamo solo aggiornare la BMU stessa e non il suo intorno.

In questa fase la topologia della SOM è già abbastanza ben definita quindi non ci interessa che la funzione di neighborhood copra molti neuroni.

Le SOM funzionano bene quando gli esempi che utilizziamo vivono in uno spazio a moltissime dimensioni ma descrivono una varietà (manifold) che ha un numero di dimensioni molto più basso.
(Bidimensionale: ogni neurone ha solo un vicino sinistro e un vicino destro)
Manifold: termine che indica la dimensionalità dei dati.

7.5 La qualità delle SOMs

L'output di una SOMs varia in base a diversi fattori (impostazioni iniziali dei parametri come il numero di neuroni, la funzione di vicinanza, il learning rate...).
Per valutare la qualità possiamo tener conto di due fattori:

- **Learning quality**: la qualità del vettore di quantizzazione. Ovvero quanto i neuroni nella SOM riescono ad approssimare bene i dati in input.
- **Projection quality**: qualità di quanto la topologia viene preservata.

Le tre misure che vengono utilizzare per misurare la qualità sono:

- Quantization Error
- Topographic Error
- Distortion Measure

7.5.1 Errore di quantizzazione

Misura quanto bene i dati vengono mappati sulla SOM, lo stimolo che viene dato alla SOM viene mappato sulla BMU, questo errore è la distanza tra lo stimolo dato e la BMU. Viene rappresentato con $QE = \frac{\sum_{l=1}^{L} ||x_1 - w_{i(X_l)}||}{L}$.
L'errore di quantizzazione misura solo la qualità della quantizzazione dello spazio di input (no spazio topologico).

7.5.2 Errore topografico

La topologia è ben preservata se è vero che punti vicini nello spazio di input questi sono anche vicini nello spazio di output o viceversa. L'errore topografico

misura la percentuale di punti in cui la prima e la seconda BMU non sono adiacenti nello spazio di output. Formalizzando l'errore topografico è: $TE = \frac{1}{L}\sum_{l=1}^{L} u(X_l)$ dove $u(X)$ è 1 se il primo e il secondo BMU non sono adiacenti nello spazio di output, 0 altrimenti.

7.5.3 Misura di distorsione

Cerca di catturare entrambe le proprietà viste fin'ora. Si può dimostrare formalmente che non esiste una funzione di loss fissa che la SOM sta ottimizzando durante l'apprendimento. Si può invece mostrare che modificando la regola di apprendimento e assumiamo che la funzione h abbia ampiezza fissa allora si può mostrare che la quantità utilizzata dalla SOM è definita da:$E_d = \sum_{l=1}^{L}\sum_{j=1}^{J} h_{j,i(x_l)}||X_l - W_j||$.
Errorre di distorsione: può essere decomposta in 3 parti:

- componente che misura la qualità della quantizzazione vettoriale
- componente che misura la qualità della ricostruzione topologica
- un componente ricompone le due componenti

7.6 Applicazioni

Le SOMs hanno un'applicazione molto varia.

Self-Organizing Maps (ANNO 2019/2020)

7.7 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 7.5, possiamo vedere un esempio di Self-Organizing Map, che è forma-

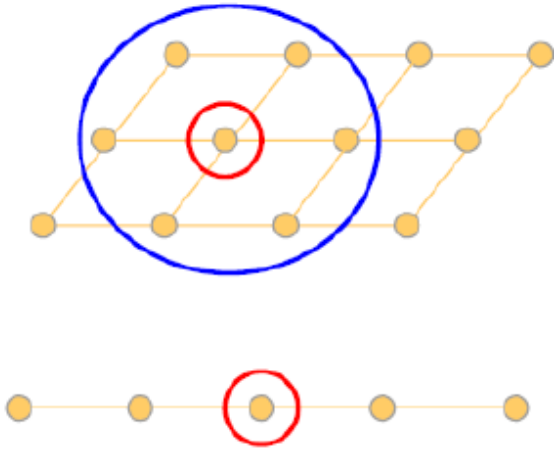


Figura 7.5: Esempio di SOM

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 7.6, possiamo vedere come i neuroni formano una griglia, ogni neurone si specializza con una determinata classe e i neuroni vicini lo "aiutano" a riconoscere quella classe stessa.

7.8 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

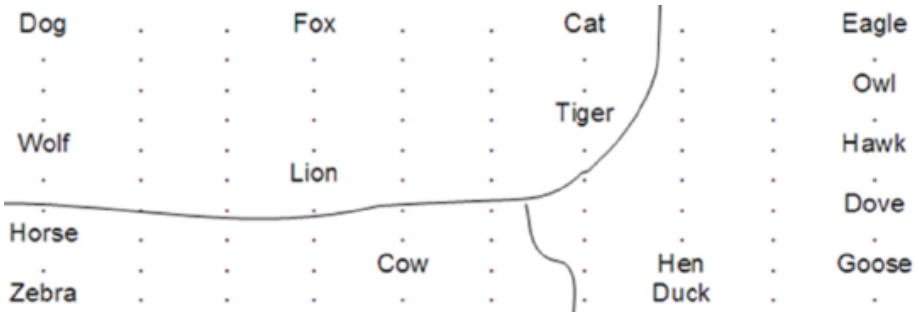


Figura 7.6: Disposizione dei Neuroni

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:

$$w_j^T x = w_{j1}x(1) + w_{j2}x(2) + \cdots + w_{jn}x(n) \tag{7.6}$$

Il neurone che presenta il valore più alto derivato dalla Formula 7.6 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.

7.9 Fase di Learning

7.9.1 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 7.6. 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:

$$d = i : \forall j ||x - w_i|| \leq ||x - w_j|| \tag{7.7}$$

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 7.6 e 7.7 sono identiche.

7.9.2 Aggiornamento dei pesi

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

$$w_i(n+1) = w_i(n) + \eta(n)(x - w_i(n)) \tag{7.8}$$

Nella Formula 7.8 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:

$$\eta(n) = \eta_0 \cdot exp(-\frac{n}{\tau}) \tag{7.9}$$

La Formula 7.8 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:

$$h_{ji}(n) = exp(-\frac{d_{ji}^2}{2\sigma(n)^2}) \tag{7.10}$$

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:

$$\sigma(n) = \sigma_0 \cdot exp(-\frac{n}{\tau}) \tag{7.11}$$

Da precisare che, sia nella Formula 7.9 sia nella Formula 7.11, il valore di τ è fissato. La Formula 7.8 viene quindi corretta nel modo seguente:

$$w_j(n+1) = w_j(n) + \eta(n)h_{ji}(n)(x - w_j(n)) \tag{7.12}$$

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 7.12, 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.

7.9.3 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 7.6 o 7.7
4. **aggiornamento**: i pesi vengono aggiornati seguendo la Formula 7.12
5. **continua**: finché non si raggiunge una certa condizione di convergenza

Capitolo 8

Autoencoders

Sono strumenti non supervisionati che hanno diversi utilizzi. Uno degli utilizzi principali è quello per migliorare le reti supervisionate.

- Colorazione delle immagini
- Super resoloution: aumentare risoluzione immagini
- Image inpainting
- ...

8.1 Definizione

Un autoencoder è una rete neurale che è addestrata per cercare di copiare sul proprio out l'input che gli viene mostrato. Possiamo vederlo come diviso in 2 parti:

- Encoder: $h = f(x)$
- Decoder: $r = g(h)$

Da un autoencoder non deve copiare identicamente gli input sugli output ma viene forzato a mantere gli aspetti salienti e copiarli in output ed eliminare gli aspetti inutili. h è una compressione di x, quindi già solo quello avrebbe un sacco di applicazioni se visto come un compressore.

8.2 Tipi di autoencoders

Esistono diversi tipi di autoencoders.

8.2.1 Undercomplete autoencoders

La rappresentazione h che andiamo ad apprendere ha una dimensionalità più piccola dell'input x. Vogliamo forzare l'autoencoder a imparare qualche proprie-

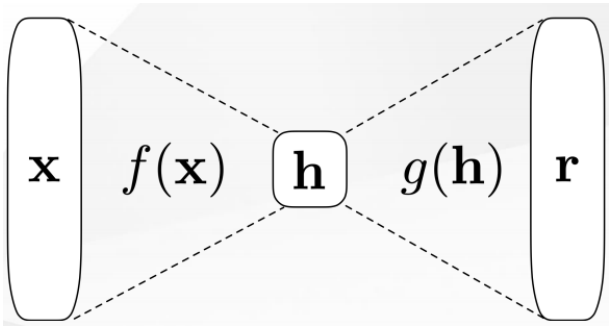


Figura 8.1: Esempio di rete

tà importante di x, non solo copiarlo pari pari. Questo viene fatto riducendo la dimensione di h obbligando l'autoencoder a comprimere alcuni aspetti di x. In un mondo ideale h riesce a cogliere la dimensionalità del manifold tralasciando le dimensioni che non servono.

L'apprendimento si fa addestrando una rete feedforward tramite la funzione: $L(x, g(f(x)))$, quindi tutte le volte che $g(f(x))$ (che sarebbe la r) è diverso da x la loss penalizza l'autoencoder e grazie alla retropropagazione dell'errore dovremmo modificare i pesi in modo corretto.

Autoencoders e PCA (Principal Component Analysis)

Se il decoder è lineare, quindi non ci sono funzioni di attivazione non lineari e la funzione di loss è l'errore quadratico medio (LMS introdotto da Adaline), allora un autoencoder undercomplete sta cercando di risolvere lo stesso problema che sta cercando di risolvere un PCA.
In altre parole, come effetto collaterale per questo processo di apprendimento di copiare l'input nell'output, l'autoencoder prende in considerazione un sottospazio dei dati di apprendimento, cioè trova sottospazio in cui ci sono le componenti principali dei nostri dati.

La **PCA** può essere derivata come un algoritmo che proietta un esempio x su un vettore a più basse dimensioni c cercando di minimizzare l'errore di ricostruzione. Formalizzando:

$$f(x) = \arg\min_c ||x - g(c)||_2 \tag{8.1}$$

In pratica cerchiamo il vettore c per il quale si ha l'errore di ricostruzione più basso.

Se vincoliamo g a sottostare ad alcuni vincoli otteniamo precisamente la PCA. Per ottenere la PCA:

- g è lineare: $g(c) = Dc$ per qualche D.
- le colonne di D sono ortogonali[1] tra di loro(per semplificare il problema, altrimenti avremmo difficoltà a trovare una soluzione in forma chiusa).
- le singole colonne di D devono avere una norma unitaria (si rende unica la soluzione).

Quindi possiamo scrivere che: $f(x) = D^T x$ e $r(x) = DD^T x$ ma sopratutto, possiamo dire che la matrice D è data dagli autovettori che corrispondono ai più grandi autovalori della matrice $X^T X$.

In poche parole:

- Un undercomplete autoencoder minimizza la loss $L(x, g(f(x)))$.
- PCA trova il mapping $f(x) = \arg\min_c ||x - g(c)||_2$, imponendo g come modello lineare.

Deve essere chiaro che quando $L = ||\cdot||_2$ e il livello di ricostruzione è composto da unità lineari, allora abbiamo 2 approcci per risolvere lo stesso problema.

Riassunto sulla PCA

Nel procedimento della PCA si comincia prendendo la matrice X sottraendo la media calcolata per colonna. Successivamente si calcola la matrice di covarianza tramite $X^T X$. Di questa matrice si calcolano autovettori ed autovalori e si genera la funzione di costruzione prendendo gli autovettori che corrispondono agli autovalori più grandi della matrice.

La lunghezza dei vettori corrisponde a quanto sono sparsi i dati nella direzione che si è appena trovata. Prendendo i vettori con gli autovalori maggiori, prendiamo i vettori lungo i quali i dati sono più dispersi e spiegano quindi la maggior parte della varianza dei dati.

[1] Matrice ortogonale: matrice invertibile, la cui inversa coincide con la trasposta

Se dobbiamo rinunciare ad una dimensione conviene senza dubbio rinunciare alla dimensione più piccola, in questo modo riusciamo ad ottenere una approssimazione dei dati migliore perchè perdiamo meno informazioni.

A partire da $A^T A$ (si ottiene una matrice quadrata) un numero scalare λ_0 è un autovalore della matrice che stiamo considerando se esiste un vettore colonna non nullo v tale che la matrice moltiplicata per il vettore colonna v è uguale a $\lambda_0 v$ quindi λ_0 corrisponde al autovalore mentre v all'autovettore.

$$Av = \lambda_0 v \tag{8.2}$$

8.3 Regularized autoencoders

La differenza di dimensione tra il *code* hf e l'input x determina quanto l'autoencoder è obbligato a imparare solo le parti più rilevanti dell'input, quindi determina se l'autoenoder impara qualcosa o no. Se la dimensione di h è molto grande, l'autoencoder non è in grado di imparare niente di rilevante ed è in grado di apprendere soltanto la funzione identità. **(h è il vettore in mezzo che ha un certo numero di componenti)**

Per mantenere un'architettura con h non più piccolo dell'input si inserisce un termine di regolarizzazione, quindi non si limita la capacità del modello. Si usa una funzione di loss per incoraggiare il modello ad imparare proprietà utili.

Varietà di autoencoders regolarizzati:

- Inducono una sparsità nella rappresentazione → **sparse autoencoders**
- Indurre una robustezza rispetto al rumore o agli input mancanti → **denoising autoencoder**
- Inducono che la derivata della funzione di costo sia piccola → **contractive autoencoders**

8.3.1 Sparse autoencoder

In questo caso non implicano che h sia più piccolo di x.
Uno sparse autoencoder aggiunge un termine di penalizzazione $\Omega(h)$ alla funzione di loss

$$L(x, g(f(x))) + \Omega(h) \tag{8.3}$$

vincolando h ad essere un vettore sparso (h quasi tutti i valori uguali a 0).

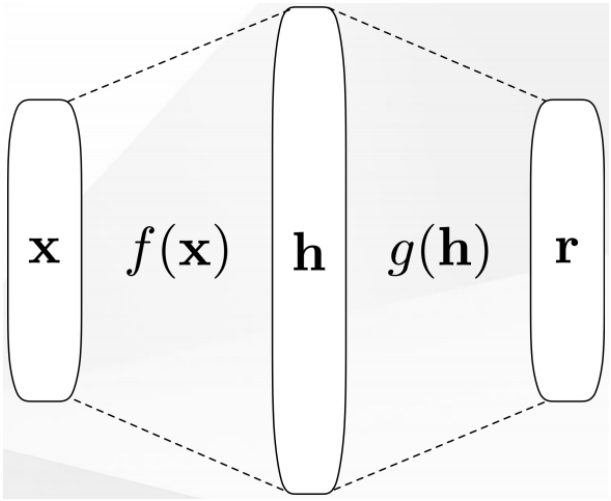


Figura 8.2: Sparse Autoencoders

8.3.2 Digressione su modelli probabilistici

Consideriamo un insieme di m esempi $\mathbb{X} = [x^{(1)}, ..., x^{(m)}]$ estratti da una distribuzione generativa sconosciuta dei dati $p_{data}(\boldsymbol{x})$. $\boldsymbol{\theta}$ denota i parametri che governano la distribuzione di probabilità sui dati p_{data}, ad es. se la distribuzione è gaussiana $\boldsymbol{\theta}$ denota media e varianza. Sia $p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$ la distribuzione di probabilità del modello sui parametri $\boldsymbol{\theta}$: in altre parole, $p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$ mappa ogni esempio $\boldsymbol{x}$ in un numero reale che stima la probabilità vera $p_{data}(\boldsymbol{x})$.
Per stimare $\boldsymbol{\theta}$, utilizziamo la maximum likelihood, così definita:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} p_{model}(\mathbb{X}; \boldsymbol{\theta}) \tag{8.4}$$

$$= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{model}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{8.5}$$

Questa produttoria ha una serie di inconvenienti. La peggiore è che moltiplica tanti valori molto piccoli tra di loro, portando a un possibile errore di underflow. Per risolvere questo inconveniente, di solito si utilizza la log likelihood, cioè si prende il logaritmo della produttoria. Prendere il logaritmo della produttoria non cambia il risultato dell'arg max e, soprattutto, trasforma il prodotto in una somma di logaritmi:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{model}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{8.6}$$

Adesso, riscaliamo la funzione dividendo per m, in modo da ottenere, come criterio di massimizzazione, il valore atteso rispetto alla distribuzione di probabilità

$\hat{p}_{data}$ definita dal training set:

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\tilde{x}\hat{p}_{data}} \log p_{model}(\boldsymbol{x}) \tag{8.7}$$

Un modo semplice per interpretare la maximum likelihood è di vederla come il minimizzare la dissimilarità tra la distribuzione di probabilità $\hat{p}_{data}$ definita dal training set e la distribuzione definita dal modello p_{model}. Il grado di dissimilarità tra le due è misurata dalla KL divergence, data da:

$$D_{KL}(\hat{p}_{data}||p_{model}) = \mathbb{E}_{\tilde{x}\hat{p}_{data}}[\log \hat{p}_{data}(\boldsymbol{x}) - \log p_{model}(\boldsymbol{x})] \tag{8.8}$$

Si noti che il primo termine della sottrazione è dato dal training set, dunque noi dobbiamo preoccuparci di massimizzare solamente:

$$\mathbb{E}_{\tilde{x}\hat{p}_{data}}[\log p_{model}(\boldsymbol{x})] \tag{8.9}$$

Minimizzare la KL divergence corrisponde esattamente a minimizzare la cross-entropy tra due distribuzioni. Si ricorda che, in generale, la cross-entropy della distribuzione p rispetto a q, misura quanto p è dissimile da q.

Consideriamo ora la stima maximum a posteriori (MAP). È come la maximum likelihood, ma in più tiene conto della distribuzione di probabilità a priori $p(\boldsymbol{\theta})$:

$$\boldsymbol{\theta}_{MAP} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}|\boldsymbol{x}) \tag{8.10}$$

$$= \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{x}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{8.11}$$

Utilizzando la log likelihood, otteniamo:

$$\boldsymbol{\theta}_{MAP} = \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{x}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) \tag{8.12}$$

La MAP ha il vantaggio di ottenere le informazioni non solo guardando dal training set, ma anche dalla probabilità a priori.

8.3.3 MAP legata agli AE

Effettuare apprendimento con un termine di regolarizzazione, può essere interpretato proprio come una approssimazione MAP, in cui il termine di regolarizzazione aggiunto corrisponde alla distribuzione di probabilità a priori sui parametri del modello ($p(\boldsymbol{\theta})$).

In ottica di sparse autoencoder, con $p_{model}(\boldsymbol{h})$ ci riferiamo alla distribuzione a priori del modello sulle variabili latenti, che rappresenta la preferenza a priori del modello **prima** di vedere $\boldsymbol{x}$. Essa è diversa dalla solita probabilità a priori $p(\boldsymbol{\theta})$, in quanto quest'ultima si riferisce alle preferenze sui parametri del modello

prima di vedere i dati di training. La differenza onestamente non l'ho capita, una dice che è prima di vedere x e l'altra prima di vedere i dati, dunque cosa cambia?

Questo non ne sono sicuro: In uno sparse autoencoder uno deve effettuare la maximum likelihood: $\arg\max_{\boldsymbol{\theta}} p_{model}(x;\boldsymbol{\theta})$.

Sparse autoencoders come GM (generative models)

La cosa interessante è vedere il modello da un punto di vista probabilistico. Supponiamo di avere un modello generativo con un certo numero di variabili visibili x, e alcune variabili latenti[2] h, abbiamo che:

$$p_{model}(x,h) = p_{model}(h)p_{model}(x|h) \tag{8.13}$$

dove $p_{model}(h)$ è il modello a priori della distribuzione sulle variabili di latenza.

IMPORTANTE: $p_{model}(h)$ **è un precedente diverso da quello di solito assunto nei modelli probabilistici.** Di solito il termine probabilità a priori fa riferimento al bias che ha l'algoritmo di apprendimento rispetto ad un'ipotesi prima di vedere i dati. **Qui usiamo il termine probabilità a priori per indicare la preferenza che ha il modello stesso per la rappresentazione latente** h **prima di vedere i dati di** x.

$$log p_{model}(x) = log \sum_h p_{model}(h,x) \tag{8.14}$$

Possiamo pensare che l'autoencoder come approssimazione di questa sommatoria ma con un h predefinito che abbia il valore più probabile. L'unica h che viene utilizzato è quello determinato dalla prima parte della rete che estrae l'h più probabile per una buona ricostruzione dato x. Quindi, dato il valore h generato abbiamo la massimizzazione:

$$\log p_{model}(h,x) = \log p_{model}(h) + \log p_{model}(x|h) \tag{8.15}$$

Alcune scelte della funzione di regolarizzazione h corrispondono al modello appena introdotto e quindi sono giustificate dal punto di vista probabilistico. Se un set $\Omega(h) = \lambda \sum_i |h_i|$ (induce sparsità del vettore h, norma 1 di h), questa funzione si ottiene se assumiamo che la nostra probabilità a priori su h sia di tipo laplaciano[3]. Formalmente:

$$p_{model}(h_i) = \frac{\lambda}{2} e^{-\lambda|h_i|} \tag{8.16}$$

[2]Variabili latenti: variabili non direttamente osservabili in quanto rappresentano concetti molto generali o complessi

[3]Distribuzione laplaciana: è la distribuzione di massima entropia

quindi:

$$-\log p_{model}(h) = \sum_i (\lambda |h_i| - \log \frac{\lambda}{2}) = \Omega(h) + const \tag{8.17}$$

Da questo punto di vista la penalità che induce la sparsità (norma 1 di h) non è un termine di regolarizzazione ma la conseguenza di come il modello pensa alla variabile latente.

Profondità e potenza rappresentativa
Una rete con un unico livello hidden è un approssimatore universale. Nelle architetture feed forward all'aumentare dei livelli di profondità aumenta la capacità computazionale, quindi si ha una riduzione esponenziale dei costi computazionali e vi è anche una decrescita nel numero di esempi necessari. Autoencoders più profondi hanno una miglior compressione rispetto ad un autoencoder lineare.

Una strategia è quella di costruire una rete profonda utilizzando più livelli shallow (superficiali) e poi si prosegue con l'apprendimento supervisionato.

Encoder e decoder stocastici
L'obiettivo di una rete è quello di cercare modellare $p(y|x)$ con la rete neurale e pensare alla rete come uno strumento per minimizzare $-\log p(y|x)$ (-log-likelihood). Quindi possiamo vedere il decoder come una rete che modella $p_{decoder}(x|h)$. La probabilità che vogliamo modellare si distribuisce in modo *gaussiano* (questa è un'assunzione).

$$p_{decoder}(x|h) = \mathcal{N}(x; \hat{\mathrm{x}}, I) \; con \; \hat{\mathrm{x}} = Wh \tag{8.18}$$

quindi minimizziamo log likelihood negativo che corrisponde alla minimizzazione dell'errore loss quadratico: si fa la **discesa del gradiente su** $-\log p_{decoder}(x|h)$

$$p_{decoder}(x|h) = c_1 \; exp(-(x - Wh)_T I(x - Wh)) = c_1 \; exp(-||x - Wh||_2) \tag{8.19}$$

$$-\log p_{decoder}(x|h) = ||x - Wh||2 + c \tag{8.20}$$

Se x si fosse distribuito secondo la funzione di Bernulli[4], l'output sarebbe settato secondo una sigmoide. Invece se x fosse un valore discreto corrisponderebbe ad una distribuzione softmax. In generale tutti gli autoencoders sono un modo di calcolare la congiunta: $p_{model}(x, h)$

In generale le distribuzioni marginali di decoder ed encoder potrebbero non essere compatibili con un'unica distribuzione congiunta. In questo caso trattando decoder ed encoder come **denoising** si possono rendere compatibili asintoticamente.

[4] $P(X = \; i) = p^i(1-p)^{1-i}$ dove p $\in [0, 1]$

8.3.4 Denoising autoencoders

Per vincolare gli autoencoder ad apprendere informazioni in modo interessante:

$$L(x, g(f(\tilde{x}))) \tag{8.21}$$

Dove $\tilde{x}$ è la copia corrotta di x con del rumore. Per riuscire ad avere buone performance l'autoencoder rimuove il rumore dall'input, si può mostrare che questo tipo di apprendimento forza f e g ad imparare la struttura dei dati, quindi la distribuzione delle x che abbiamo in input. Quindi possiamo fare la

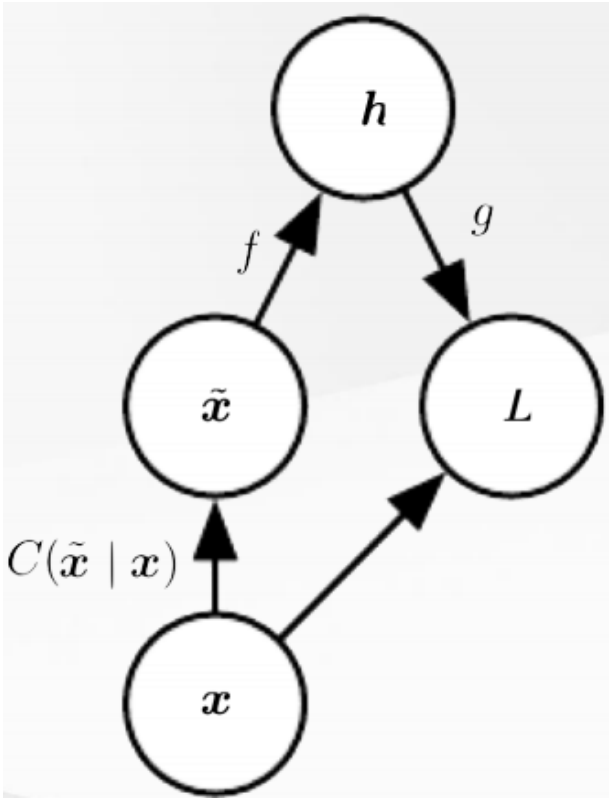


Figura 8.3: Esempio di DAE stocastica

discesa del gradiente sulla log likelihood e se la parte di encoding è deterministica si può mostrare che tutto l'autoencoder si può addestrare end to end (uso di retropropagazione dell'errore) cercando quindi di massimizzare:

$$-\mathbb{E}_{x \sim p_{data(x)}} \mathbb{E}_{\tilde{x} \sim C(\tilde{x}|x)} \log p_{decoder}(x|h = f(\tilde{x})) \tag{8.22}$$

Vector field: in ogni punto dello spazio la funzione restituisce un vettore che indica una direzione.

Un modo interessante di pensare ai DAE è come un'approssimazione di un campo di vettori. Minimizzare ci permette di prendere il vettore corrotto e portarlo sul manifold a cui appartiene.

Laerning Manifold un modo di caratterizzare i manifold è utilizzare il piano tangente, se siamo abbastanza vicino al punto possiamo approssimare la curva utilizzando questi piani. Posso assumere che tutti i punti nell'intorno del punto che sto guardando abbiano tutti la stessa etichetta. Questi piani ci dicono come possiamo variare x e rimanendo comunque sul manifold.

Quindi come abbiamo detto le procedure di apprendimento degli autoencoders coinvolgono 2 distinte forze:

1. Apprendere una rappresentazione h tale che a partire da essa si riesca a capire il valore di input
2. soddisfare qualche vincolo di regolarizzazione

L'autoencoder può ricostruire solo le possibili variazioni dei dati visti in input, quindi tralascia le variazioni che non centrano nulla.

Se la distribuzione che genera i dati si concentra su un manifolds di basse dimensioni. l'autoencoder tende ad apprendere il manifold in cui vivono i dati: apprende una rappresentazione che cattura implicitamente il sistema di coordinate del manifold e facendo questa proiezione si perdono tutte le direzioni che portano lontano dal manifold.

8.3.5 Contractive autoencoders

Aggiunge un termine di regolarizzazione:

$$\Omega(h, x) = \lambda \sum_i ||\nabla_x h_i||^2 \tag{8.23}$$

Questo forza che per piccole variazioni di x non vi siano grosse variazione dell'output. Il termine di regolarizzazione può essere scritto come:

$$\Omega(h) = \lambda ||J||_F^2 = \lambda || \frac{\partial f(x)}{\partial x} ||_F^2 \tag{8.24}$$

dove $\frac{\partial F(x)}{\partial x} = J$ è la matrice Jacobiana[5] e

$$||A||_F = \sqrt{\sum_{i,j} |A_{i,j}|^2} \tag{8.25}$$

[5]Matrice Jacobiana: data una funzione (solitamente vettoriale), è la matrice che contiene le derivate parziali prime della funzione vettoriale a cui fa riferimento

Se prendiamo un vettore x e lo moltiplichiamo per la Jacobiana, se il vettore x è lungo la funzione f in quella direzione cambia velocemente mentre se teniamo il vettore corto cambierà lentamente.

Perchè si chaimano contractive? Perché l'autoencoder impara a mappare un intorno del punto x in uno spazio ridotto rispetto a quello di partenza.

Un altro modo di vedere questo fatto: se abbiamo un λ molto alto oppure se rimuovessimo un termine (non ho capito quale) prenderemmo una funzione costante che andrebbe a mappare tutto lo spazio di input in un unico punto dal momento che si sta forzando la derivata prima ad essere zero.

Nei casi in cui λ non abbia un valore troppo grande stiamo cercando di forzare il mapping a ridurre le dimensioni dello spazio.

La proprietà contrattiva è vera solo localmente, infatti due punti vicini possono essere mappati in due zone dello spazio molto distanti

8.3.6 CAE e DAE

La ricostruzione con rimozione del rumore (denoising) è equivalente ad una penalità di tipo contrattivo (non su f) ma sulla funzione di ricostruzione mappando x a $r = g(f(x))$.
Denoising autoencoders fanno si che la funzione di ricostruzione (g) sia in grado di sopportare piccole variazioni al suo input.
Mentre **contracrtive autoencoders** fanno la stessa cosa ma sulla funzione f, fanno si che tutto il processo sia robusto perché $f(x)$ è robusta a piccole variazione del suo input.

Nel caso di CAE le due forze sono reconstruction error (cioè il fatto che l'output sia il più possibile simile all'input) e la penalità contrattiva, il compromesso produce una soluzione con delle derivate più piccole.

Capitolo 9

Representation Learning

Una buona rappresentazione permette la condivisione delle proprietà statistiche dei dati quando si affrontano dei task diversi.
Si utilizzano le rappresentazioni apprese per fare le classificazioni con scopi diversi.
Utile anche per trasferire la conoscenza appresa per problemi per cui non ci sono esempi ma per cui esiste una rappresentazione dei dati (Semi Supervised Learning = **SSL**, li vediamo in nelle prossime sezioni).

Cambiare la rappresentazione dei dati può rendere un problema molto semplice o molto difficile. Una buona rappresentazione rende i task successivi più facili.
Una rete feedforward con algoritmo di apprendimento supervisionato può essere vista come un representation learning, l'ultimo livello è un semplice classificatore in quanto i livelli precedenti hanno classificato in modo accurato i dati.
Multiple Objectives: non sempre apprendere solo la rappresentazione porta buoni risultati, bisogna imporre delle proprietà sulla rappresentazione stessa. I due obiettivi prinicipali sono:

- Preservare tutte le informazioni riguardante gli input
- Forzare delle proprietà interessanti sulle rappresentazioni

Di solito questi due obiettivi sono in contrasto uno con l'altro.
Queste tecniche sono spesso utili per risolvere problemi in cui il numero di esempi etichettato è ridotto.

9.1 Greedy Layer-Wise unsupervised pretraining

È una tecnica strumentale per far rinascere l'interesse verso le reti neurali profonde, ha messo in evidenza come fosse possibile addestrare reti molto profonde con i vincoli della strumentazione reale e con pochi esempi a disposizione. Siamo in un regime non supervisionato e trasferiamo la conoscenza ad un regime supervisionato.
Questa tecnica per funzionare ha bisogni di qualche procedura shallow (piatta):

- Restricted Boltzman Machines: si possono addestrare con backpropagation.
- Single layer autoencoder.
- altri modelli sempre piatti.

Costruiamo una rete profonda impilando uno sull'altro tanti di questi modelli fino a quando non arrivo all'output. Ognuno di questi modelli è addestrato in modo indipendente in modo da generare un nuovo output, si procede per ogni livello che prende in input l'output del livello precedente e genera un nuovo output.

- **Greedy**: algoritmo che non tenta di arrivare ad un ottimo globale, va avanti a pezzettini.
- **Layer-wise**: procede livello per livello.
- **Unsupervisioned**: i livelli sono addestrati in modo non supervisionato.
- **Pretraining**: si suppone che questo sia il primo passo di quelli necessari per addestrare in modo completo la rete cioè prima di far partire la fase di fine tuning: si va a fare questa operazione solamente sui pesi pre-addestrati

Varianti: possiamo usarla anche come passo di inizializzazione per un modello non supervisionato. È anche possibile utilizzarla per fare greedy layer-wise supervisioned pretraining.
Perché funziona questa tecnica: non sempre funziona, alcune volte rende la rete ancora più complessa. Ci sono 2 concetti fondamentali per capire se funziona.

1. **L'inizializzazione dei parametri**: la scelta dei parametri iniziali può avere un effetto di regolarizzazione importante sul modello. Una volta si pensava che la funzione di loss avesse molti minimi locali, invece si è capito che la funzione di loss è molto frastagliata ma esistono molti minimi locali molto buoni. Questo tipo di inizializzazione dei parametri porta a

considerare delle regioni dello spazio che non sarebbero state considerate con l'inizializzazione casuale, dovuto al fatto che le zone analizzate solitamente sono molto disomogenee (difficili da individuare con inizializzazioni casuali)

2. **Apprendere la struttura dei dati in input** può aiutare nell'apprendere il mapping dall'input all'output. Quello che imparo su un task supervisionato lo posso poi applicare in un altro task non supervisionato.

9.2 Simultaneous supervised and unsupervised learning

Non c'è nessuna garanzia che le features che apprendiamo in modo non supervisionato abbiano proprietà interessanti. Utilizzare un approccio misto aiuta parecchio, in questo modo includiamo dei vincoli.

Quando è utile usare questo approccio:

- Quando la **rappresentazione originale è povera**, ovvero dai dati in input è difficile cogliere una struttura: apprendere una struttura più ricca può aiutare molto.
- Se pensiamo questo passo come un **regolarizzatore**, è molto importante quando abbiamo un numero di esempi piccolo. Oppure quando il numero di esempi non etichettati è grande (etichettati piccolo) e quindi posso apprendere la struttura dati. Quindi agisce come regolarizzatore quando faccio fine tuning[1] dei dati non etichettati.
- Un altro scenario utile è quando la **funzione che devo apprendere è molto complicata**, questo tipo di regolarizzatore non forza la funzione che dobbiamo apprendere sia semplice, quindi aiuta a scoprire le features nei dati che sono maggiormente utili per risolvere il problema.

Mettendo insieme tutte le cose, quando la funzione è complicata e la struttura dei dati determina le etichette che voglio assegnare agli esempi, questo tipo di approccio è molto utile.

Svantaggi è una tecnica molto difficile da calibrare, inoltre gli iper parametri del pretraining devono essere aggiustati e questo può essere un passaggio molto lento.

[1] Fine tuning: addestrare l'intera rete a partire dai pesi appresi con l'algoritmo, in modo che sia in grado di predire y

Utilizzi odierni il pretraining non supervisionato è oggigiorno un po' abbandonato, invece il pretraining supervisionato è molto popolare.

9.3 Transfer Learning and Domain Adaptation

Tecniche che fanno riferimento ad una situazione in cui apprendiamo qualcosa in un certo ambiente e siamo in una distribuzione P_1 e cerchiamo di utilizzare le informazioni apprese per afferire qualcosa in una nuova distribuzione P_2. Questa è una generalizzazione del greedy pretraining dove vi è il passaggio tra un modello non supervisionato (P1) ad uno supervisionato (P2). Si tratta quindi di due task: il transfer learning e il domain adaptation.

9.3.1 Transfer learning (TL)

Nel caso del TL il learner (algoritmo di apprendimento) è difronte a 2 o più obiettivi di apprendimento. Molti dei fattori che descrivono la distribuzione P_1 sono caratteristici (rilevanti, in comune) anche in P_2. In questo caso l'input è lo stesso o ha la stessa forma, e quello che cambia è il modo di etichettare i dati. Nella maggior parte dei casi siamo di fronte ad un modello come in 9.1, con l'input dal basso poi si crea una rappresentazione condivisa con più task, la rappresentazione condivisa è poi specializzata per poter risolvere i task veri e propri.

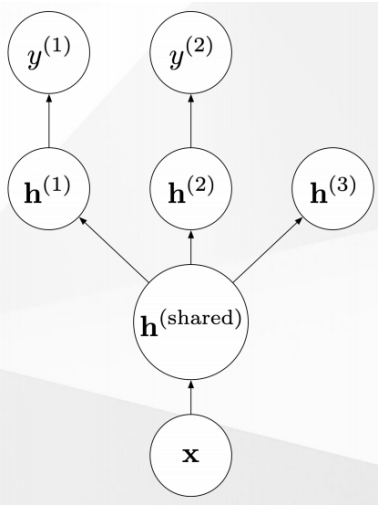


Figura 9.1: Esempio di transfer learning

Quando gli input provengono dalla stessa distribuzione ma sono un po' diversi uno dall'altro, bisogna eliminare dagli input le cose che li differiscono. L'esempio 9.2 è molto utile per il riconoscimento vocale in quanto va ad eliminare le differenze di pronuncia tra le varie persone.

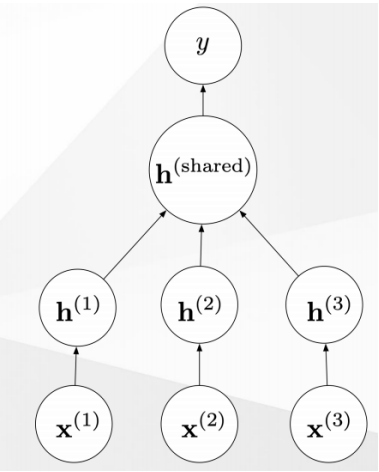


Figura 9.2: Esempio di transfer learning

9.3.2 Domain adaptation

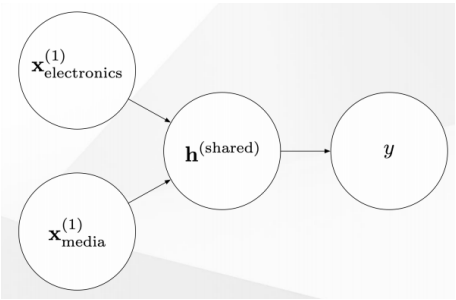


Figura 9.3: Esempio di domain adaptation

In questo caso non abbiamo un cambio di obiettivo tra un task e l'altro (come in TL), in questo caso il task rimane lo stesso e quello che cambia è che le distribuzioni in input sono un po' diverse (come il secondo esempio di TL). In questo caso l'esempio è sentimental analysis (ad esempio riconoscere se la persona che scrive è arrabbiata, felice, innamorata, ecc.).

Concept drift indica quando il concetto che sottende un certo task si sposta, quindi il TL aiuta in quanto non devo apprendere da 0 il modello, ma semplicemente riadattare la rete alla nuova situazione.
In questo caso viene fatto l'esempio del calcolo dell'interesse medio su un mutuo in una certa zona, il modello viene generato su certi dati ma dopo un cambio dell'economia (es pandemia COVID-19) e quindi il tasso di interesse medio cambia drasticamente. Il TL aiuta molto in quanto si parte da una rete già addestrata a cogliere le informazioni più importanti.

One shot learning cercare di imparare da un singolo esempio etichettato, abbiamo moltissimi esempi ma una sola etichetta (penso una per classe?). Funziona bene quando i dati sono distribuiti in uno spazio latente ovvero le classi sono ben separate tra di loro, in questo caso è molto importante imparare una rappresentazione latente che sia adatta al tipo di task. Permette di generalizzare da una sola immagine.

Zero shot learning adattarci ad una nuova situazione senza nessun esempio etichettato, questo non può portare da nessuna parte se non ho nessuna informazione che mi permette di ricostruire l'etichetta, possiamo riformulare il task con un task che include 3 variabili x (*input*) y (*output*) *e* T (*descrizione del task*). Il modello diventa: $p(y|x, T)$. Quindi, non avendo nessun esempio etichettato la variabile T contiene una descrizione che permette di ricostruire le etichette.

9.4 SSL e casuality

Semi Supervised Learning. Ipotesi: la mia rappresentazione ideale è una che fa si che si riescano a determinare le cause sottostanti ai dati che sto osservando (**ipotesi di causalità**): se la mia rappresentazione riesce ad identificare queste cause allora posso usare questa rappresentazione per riuscire a produrre le etichette.

Riuscire a sbrogliare i fattori che causano la distribuzione p_x può essere un buon primo passo per poter predire $p(y|x)$. Questa è una ipotesi che porta a cercare dei modelli per la rappresentazione h che non necessariamente sono quelli usati in passato.

Sfruttare una rappresentazione per $p(x)$ che fallisce non è utile per l'apprendimento di $p(y|x)$. Invece nel caso la $p(x)$ abbia successo può risultare molto utile per la predizione. Per formalizzare: se h rappresenta tutti i fattori che causano x e assumiamo che y sia collegato almeno ad uno di essi allora possiamo

immaginare il processo generativo:

$$p(h, x) = p(x|h)p(h) \tag{9.1}$$

In questa condizione qui la probabilità marginale di x può essere calcolata come:

$$p(x) = E_h p(x|h) = \sum_h p(h)p(x|h) \tag{9.2}$$

$$E_h = \sum_h p(h) \tag{9.3}$$

In pratica abbiamo marginalizzato h per trovare la probabilità di x. Ricostruire $p(x)$ può essere molto utile per scoprire h e predire y. Quando y è collegato ad

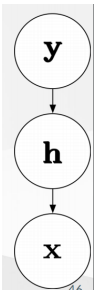


Figura 9.4: SSL

uno degli attributi di h può essere molto utile imparare h.
La regola di Bayes:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} \tag{9.4}$$

Quindi la probabilità $p(x)$ è intimamente collegata alla probabilità $p(y|x)$ e quindi conoscere la probabilità $p(x)$ può essere molto utile per la predizione di y. Apprendere h può essere utile per trovare $p(y|x)$.

Una difficoltà che incontriamo quando cerchiamo di trovare h che colga le cause che sottendono i dati è che le cause che concorrono sono molte ed è difficile cogliere tutte le cause. In questo caso ci sono 2 strategie:

- usare un segnale supervisionato per guidare il processo non supervisionato
- usare una rappresentazione h molto grande quando si usano solo apprendimenti non supervisionati

9.5 What to encode

Un'altra strategia è cambiare cos'è importante per la rappresentazione, di solito viene utilizzato il MSE (Mean Squared Error), di solito viene applicato a delle immagini.
Abbiamo visto l'immagine del braccio robotico che gioca a ping pong e la pallina occupa un numero ridotto di pixel e quindi il MSE tende ad ignorarla. Quindi si cambia la definizione di salienza e il modo di valutare gli errori: il numero di pixel (anche se pochi) ma che seguono un pattern riconoscibile allora sono importanti, si usano le GANs. Questo tipo di rete ha un encoder con un input e cerca di ricostruirlo nel modo più fedele possibile, dall'altra parte c'è un discriminatore che guarda sia istanze dell'input vero che quello generato dall'encoder e cerca di capire qual è quella vera e quella generata, quindi cerca di imparare a distinguere nel miglior modo possibile le due istanze. Il discriminatore dice solo quale delle due immagini è reale.

Anche nell'esempio del volto umano MSE sfoca sia l'orecchio che l'occhio in quanto è una caratteristica molto diversa tra ogni individuo, invece con GANs questo non succede.

Casual factors sono robusti i fattori causali sono robusti a cambiamenti nel mondo, di solito ci sono cambiamenti nelle distribuzioni oppure cambia il task che cerchiamo di modellare.

9.6 Rappresentazioni distribuite

Una rappresentazione distribuita è una rappresentazione in cui ogni oggetto è descritto con tante features e ognuna di queste è coinvolta nella rappresentazione di tanti oggetti (ad esempio forme e colori all'interno di una distribuzione di oggetti: molti di questi elementi sono condivisi da più elementi della distribuzione, ovvero ci saranno tanti oggetti con la stessa forma e/o con lo stesso colore).
Le rappresentazioni distribuite sono molto importanti perchè inducono ad una ricca similarità nello spazio, ovvero oggetti simili dal punto di vista semantico sono anche vicini dal punto di vista della distanza euclidea dai vettori che lo rappresentano. Questo aspetto manca nelle rappresentazioni puramente simbolico. (es. distanza cane, gatto e casa).

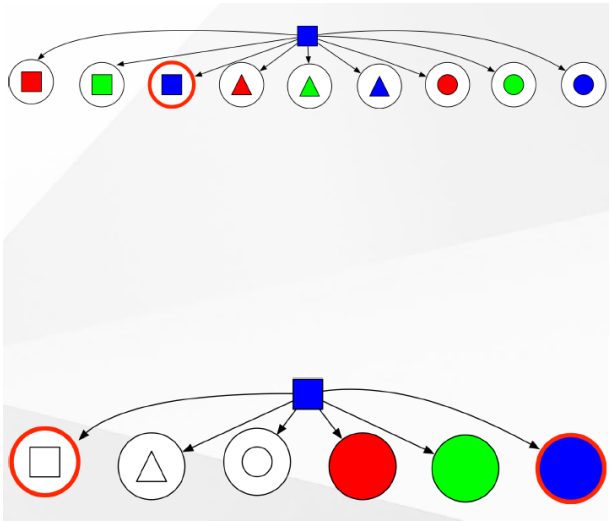


Figura 9.5: Rappresentazione distribuita

9.6.1 Generalizzazione della rappresentazione distribuita

La cosa interessante è che grazie al fatto che possono essere combinate tra di loro riescono a rappresentare moltissimi oggetti (un numero esponenziale). Senza questo tipo di proprietà per approssimare una distribuzione che non ha questo tipo di caratteristica io devo assegnare tanti esempi su ciascuna di queste zone, quindi sarebbe un numero esponenziale di esempi. Posso usare un numero minore di esempi per dare supporto statistico ad una certa zona contribuiscono esempi che condividono parte della rappresentazione, lo stesso esempio partecipa a più zone e quindi con meno esempi riesco a dare più supporto statistico con meno esempi.
Sebbene questi modelli riescano a rappresentare bene molte zone differenti la capacità di rappresentazione è limitata. Non c'è flessibilità infinita in quanto si rischierebbe di fare overfitting.

Possono essere interpretabili Quando apprendo una rappresentazione distribuita, le features possono essere interpretabili. Questo non è sempre vero. La rappresentazione che si ottiene è nell'intorno dell'input.

Guadagno esponenziale Apprendere rappresentazioni distribuite si ha un guadagno esponenziale nella capacità di rappresentare le cose in modo. Si ha un guadagno esponenziale dovuto alla profondità dei sistemi. La capacità espressiva di una architettura profonda contiene una famiglia di funzione rappresentabili

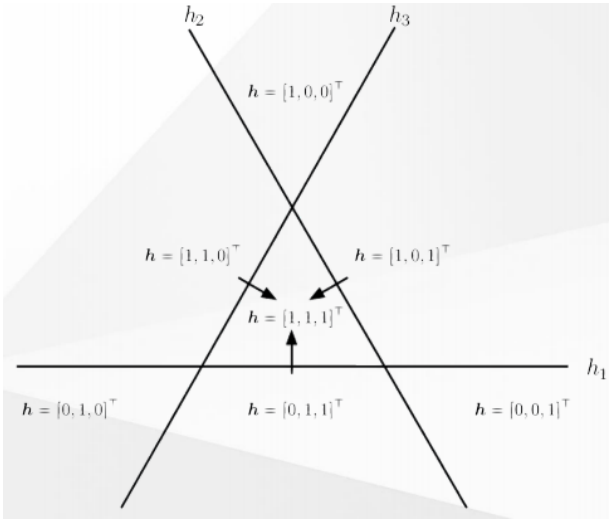


Figura 9.6: Generalizzazione di una rappresentazione distribuita

con architetture a profondità k ma che richiederebbero un numero esponenziale di hidden units.

Capitolo 10

GANs: Generative Adversarial Networks

Modelli generativi: si riferisce ad un qualsiasi sistema che prende in input un insieme di esempi che è estratto in modo indipendente ed eticamente distribuito da una distribuzione p_{data} e impara a rappresentare queste distribuzioni e estrarre nuovi esempi da tali distribuzioni.

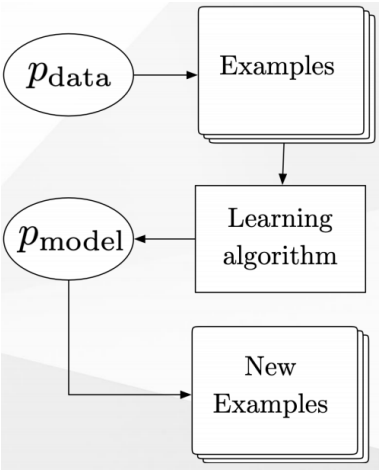


Figura 10.1: Esempio di GANs

I modelli generativi si fermano nel momento in cui hanno appreso p_{model} e sono usati per costruire nuovi esempi. Le GANs vengono utilizzate per diversi motivi: es super resolution e altre cose.

Vantaggi teorici GN:

- Abilità di rappresentare distribuzioni di grandi dimensioni
- Nel campo del Reinforcement Leraning (RL) (es. far imparare ad un agente [robot] di muoversi in un ambiente che non conosce) i GN sono molto utili in quanto permettono di fare assunzioni su un modello creato dai GN. Le query vengono fatte da sistemi RL
- Possono essere usati come base per i sistemi SSL

Vantaggi applicativi GN:

- GN molto bravi a predire output multimodali (es. dell'orecchio di un umano ricostruito in malo modo da MSE, mentre GN lo modella meglio)
- Super resolution: da immagini a bassa risoluzione a immagini con una risoluzione più alta
- GN costruiscono spazi latenti che hanno varie utilità. Ad esempio un'immagine che viene spesso deformata in modo poco legato alla realtà nei modelli classici, un GN immagina una forma simile a quella di partenza

10.1 GANs Vs Altri modelli generativi

La maximum likelihood è un modo di trovare parametri di un modello di apprendimento basato sull'idea di massimizzare la verosimiglianza del dataset sotto la condizione che i parametri siano quelli che stiamo guardando in questo momento.
Dato un θ, $P(D|\theta)$ può essere vista come una misura di "quanto bene il dataset è spiegato dal modello di parametro θ

$$\Theta^* = \arg\max p_{model}(\{x^{(i)}\}_{i=1}^m; \Theta) = \prod_{i=1}^m p_{model}(x^{(i)}; \Theta) \tag{10.1}$$

Però si preferisce lavorare in log space

$$\Theta^* = arg\ max\ log \prod_{i=1}^m p_{model}(x^{(i)}; \Theta) = arg\ max \sum_{i=1}^m log\ p_{model}(x^{(i)}; \Theta) \tag{10.2}$$

Questi modelli sfruttano la divergenza di Kullback-Leibler (KL):

$$KL(p_{\text{data}} \| p_{\text{model}}) = E_{\mathbf{x} \sim P_{\text{data}}} \left[\log \frac{p_{\text{data}}(\mathbf{x})}{p_{\text{model}}(\mathbf{x}; \theta)} \right] \tag{10.3}$$

$$= E_{\mathbf{x}\sim P_{\text{data}}} \left[\log p_{\text{data}}(\mathbf{x}) - \log p_{\text{model}}(\mathbf{x};\theta)\right] \tag{10.4}$$

$$= E_{\mathbf{x}\sim P_{\text{data}}} \left[\log p_{\text{data}}(\mathbf{x})\right] - E\left[\log p_{\text{model}}(\mathbf{x};\theta)\right] \tag{10.5}$$

Minimizzare la divergenza KL equivale a massimizzare la maximum likelihood della distribuzione p_{model} e p_{data}. **Minimizzare la divergenza KL è più facile di massimizzare la maximum likelihood**.

10.2 Explicit density models

Fully visible belief state (fvbn) Trattiamo solo distribuzioni che possiamo modellare in modo esplicito e sono trattabili[1].

$$p_{model}(x) = \prod_{i=1}^{n} p_{model}(x_i|x_1, ..., x_{i-1}) \tag{10.6}$$

L'idea che offrono è di far ricorso alla regola della catena per le probabilità (chain rule): la probabilità dell'esempio x la calcolo con chain rule applicata n volte.
Il problema di questi modelli è che la formula è costosa da calcolare: per generare un sample devo impiegare $O(n)$, il che vuol dire che questi modelli (sebbene siano ottimi nel risultati) sono molto lenti

Nonlinear indipendent components analysis si utilizza una funzione non lineare

$$p_x(x) = p_z(g^{-1}(x)) \left|\det\left(\frac{\partial g^{-1}(x)}{\partial x}\right)\right| \tag{10.7}$$

L'Interpretazione geometrica di $\det(A)$ (determinante di A)[2] rappresenta la misura di quanto sia invertibile A.

$$|p_x(g(z))dx| = |p_z(z)dz| \tag{10.8}$$

E risolvendo da questa formula otteniamo:

$$p_x(x) = p_x(g^{-1}(x)) \left|\frac{dz}{dx}\right| = p_z(g^{-1}(x)) \left|\frac{dg^{-1}(x)}{dx}\right| \tag{10.9}$$

[1] Un problema per il quale esiste una soluzione algoritmica buona (che richiede un tempo polinomiale, o inferiore) è detto trattabile, altrimenti (con un tempo esponenziale o superiore) è non trattabile

[2] Determinante di una matrice: è una misura di quanto A distorca i volumi. Ci dice anche quanto A è invertibile. il determinante è una conseguenza di come si cambiano le probabilità quando si applica una trasformazione alle probabilità stesse (parliamo di densità probabilistiche nel continuo).

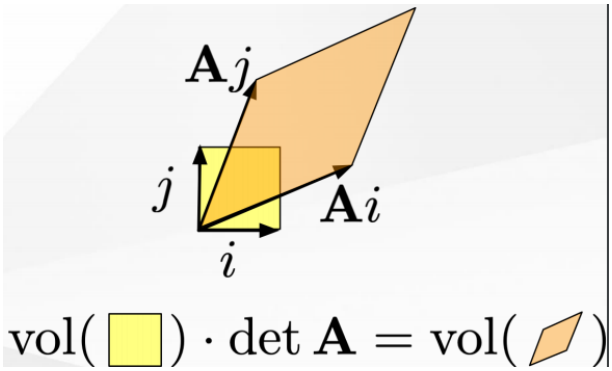


Figura 10.2: Esempio del determinante di una Jacobiana

Gestire la probabilità nello spazio originale x è molto difficile e allora utilizzo la funzione non lineare per mappare x in un nuovo spazio e poi gestisco la probabilità in questo nuovo spazio.
Cerca di risolvere il problema che sta lavorando con una distribuzione complicata, utilizzando una trasformazione che semplifica g. I due modelli che modellano in modo esplicito la distribuzione hanno 2 problemi:

- Riescono a generare i sample molto lentamente (le gan invece no)
- I vincoli imposti su g (deve essere invertibile) sono molto onerosi e non si ha tutta la flessibilità che si ha con le gan.

Ci sono altri approcci che rinunciano alla trattabilità del modello, quindi considerano anche distribuzioni non trattabili.
Per sopperire al fatto che non sono trattabili introducono qualche approssimazione

10.3 Modelli espliciti necessitano approssimazione

I metodi variazionali sono una grossa classe di metodi, l'idea è che $\log p_{model}$ non possa essere ottimizzato, quindi uso una versione del modello trattabile che lo approssima. Se con il modello approssimato ottengo una probabilità di likelihood dell'esempio che che è un **lower bound** della likelihood che darebbe il modello dell'esempio allora posso ottimizzare il sistema massimizzando il lower bound garantendo così si ottenere una certa prestazione rispetto al modello.

Il Variational AE rispetto agli AE visti in precedenza è che h rappresenta un sample ovvero un insieme di numeri estratti da una particolare distribuzione, che è la distribuzione che utilizziamo per approssimare quella che non riusciamo a trattare.
Rispetto alle GAN hanno prestazioni meno convincenti. Con un'approssimazione molto cruda la distanza tra il modello originale e quello generato potrebbe far si che anche arrivando all'ottimo rispetto all'approssimazione siamo molto lontani dalla vera distribuzione dei dati. Rispetto alle FVBN i VAE sono difficili da addestrare.

10.3.1 Variational autoencoders

I metodi variazionali

$$\mathcal{L}(x;\theta) \leq logp_{model}(x;\theta) \tag{10.10}$$

Questa è l'idea alla base dei variational autoencoder. L'idea è di un autoencoder

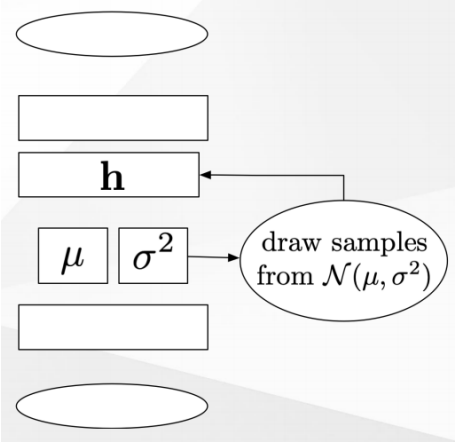


Figura 10.3: Esempio di variational autoencoder

con un input che cerchiamo di ricreare in output. La differenza è che in mezzo c'era solo una rappresentazione h che usa solo la prima parte della rete con qualche vincolo. In questo caso la rappresentazione h è un **sample** ovvero un insieme di numeri che sono estratti da una particolare distribuzione utilizzata per approssimare quello che non si riesce a trattare. Si utilizza una gaussiana multivariata e quindi si cerca di apprendere i valori della gaussiana. Rispetto alle GANs si hanno performance inferiori, dando comunque buoni risultati. Le VE sono difficili da addestrare.

10.3.2 Approssimazione con la catena di Markov

Trovare un operatore di transizione q che inizia ad estrarre dalla distribuzione completa delle approssimazioni e a furia di iterare si arriva ad un'approssimazione molto buona.

$$x' \sim (x'|x) \tag{10.11}$$

Il problema di questo tipo di approccio: è molto lento. Quando stiamo modellando anche la distribuzione, rende il tutto ancora più lento ed inutilizzabile. Un altro modo di trattare con un modello approssimato è quello di generare i dati attraverso una catena di markov.
Generare un sample $p_{model}(x)$ è troppo oneroso, perchè ci sono troppe dimensioni e distribuzioni troppo complicate. Una **catena di Markov** trova un operatore detto di transizione q che inizia ad estrarre dalla distribuzione completa delle approssimazioni e a furia di iterare questo processo arriva ad un'approssimazione buona. Dato che non si riesce a fare un'approssimazione completa di $x_1, ..., x_n$, il caso più semplice delle markov chain è che si possono mettere dei numeri a caso per $x_2, ..., x_n$ e viene estratto solo x_1. Quindi ho una situa in cui x1 viene effettivamente dalla distribuzione che sto cercando di modellare, però x2 ... xn sono presi a caso. Il prossimo passo viene fatto tenendo x_1 appena estratto, x_2 viene estratto, $x_3, ..., x_x$ rimangono con i valori scelti a caso. Questo passaggio viene ripetuto finché non vengono estratti tutti i punti e quindi viene iterato più volte il processo. Alla fine si ottiene un vettore $x_1, ..., x_n$ che sembra effettivamente estratto dalla distribuzione originale.
Il problema di questo approccio è che è molto lento in particolare durante l'apprendimento.
Se utilizziamo le markov chain solo a tempo di inferenza, questo funziona abbastanza bene ma comunque più lento rispetto alle GAN.

10.3.3 Implicit Density Models

Generative stochastic network

IDEA: invece di approssimare una certa distribuzione $x_1, ..., x_n$ troppo complicata, vengono utilizzate le markov chain per imparare come costruire l'operatore di transizione q in modo ottimale. è inutile che impari a rappresentare $p(x_1, ..., x_n)$ perché questo operatore è tutto quel che serve per lavorare con la distribuzione originale. Il problema è lo stesso di prima: la generazione degli esempi è molto lenta

Generative Adversarial network

Viene modellata la distribuzione $p(x_1..., x_n)$ in modo implicito, però si riesce comunque a lavorare e viene fatto usando un metodo molto veloce. Vantaggi:

- Gli esempi possono essere generati in parallelo, e in particolare le componenti degli esempi
- La funzione che genera gli esempi ha pochissime restrizioni
- Non c'è bisogno markov chain
- Non c'è bisogno di un bound variazionale come quello usato negli AE variazionali (che fanno una approssimazione di ciò che non si può trattare)
- Test empirici soggettivi dicono che le gan sono migliori

L'idea alla base delle gan è molto interessante perché è un regime di addestramento molto diverso da quanto c'era prima.

10.4 Come funzionano le GANs

Idea molto semplice ed interessante(?). Sono modellate come un gioco tra due giocatori:

- **il generatore**: prende in input un vettore (di numeri casuali, rumore bianco) e usa il rumore bianco per generare una nuova immagine che viene data in pasto al discriminatore con immagini reali.
- **il discriminatore**: deve dire quali immagini sono state generate e quali sono reali.

Il generatore deve creare immagini senza aver mai visto immagini del mondo reale e deve apprendere solo grazie al feedback del discriminatore. Formalizzando si può dire che: G e D sono due funzioni differenziabili (due NN) per poterne calcolare il gradiente.

- G prende in input un vettore z ed è definita in un vettore di parametri $\theta^{(G)}$, e i valori output un valore $\widetilde{x}$ estratto dallo stesso spazio in cui vive x.
- D prende in input un oggetto x (immagine) ed è definita da un insieme di parametri $\theta^{(D)}$ e il suo output è $y \in \{fake, real\}$

G e D hanno funzioni di loss J definite in termine dei parametri $\theta^{(G)}$ e $\theta^{(D)}$, ovvero:

$$J^{(D)}(\theta^{(G)}, \theta^{(D)}) \; e \; J^{(G)}(\theta^{(G)}, \theta^{(D)}) \tag{10.12}$$

Ottimizzando le loss function le due funzioni non possono interferire sui parametri dell'altra funzione e $J^{(D)}$ può intervenire solo su $\theta^{(D)}$.

Il gioco tra G e D è a somma nulla ogni volta che uno dei due giocatori ha un tipo di vantaggio, questo si tramuta in uno svantaggio di uguale entità per l'altro giocatore (es degli scacchi).

Soluzione gioco a somma nulla = punto di equilibrio di Nash: punto nello spazio $(\theta^{(G)}, \theta^{(D)})$ che è un minimo locale sia $J^{(D)}$ rispetto a $\theta^{(D)}$ che $J^{(G)}$ rispetto a $\theta^{(G)}$. Si ottiene una soluzione quando il primo giocatore non può fare nulla agendo sui suoi parametri per migliorare la sua situazione.

10.4.1 Addestramento GAN

utilizza la discesa del grandiente stocastico simultaneamente sia su $J^{(D)}$ sia su $J^{(G)}$ secondo due fasi:

- Estrazione di due mini-batch[3],uno dai dati reali e uno richiesto al generatore. In questo modo si stanno estraendo dati da entrambi gli insiemi (immagini reali e immagini generate)
- Valutazione di $J^{(D)}$ e $J^{(G)}$ e si aggiorna $\theta^{(D)}$ usando gradienti che arrivano da $J^{(D)}$ e $\theta^{G)}$ usando gradienti che arrivano da $J^{(G)}$

A volte si fanno alcuni passi in più sul discriminatore prima di tornare dal generatore (es. uno step sul generatore ogni 5 sul discriminatore). Questo perchè D è troppo "scarso" e dà troppe poche informazioni a G: si preferisce dare più risorse a D affinchè apprenda nel miglior modo.

Durante l'apprendimento sia G che D si muovono nel buio, anche il discriminatore non sa distinguere il vero dal falso all'inizio e ci saranno errori grossolani.

Funzione di loss di G usata da D per migliorare la distinzione

[3]Batch significa che si usano tutti i dati per calcolare il gradiente durante una iterazione. Mini-batch significa che si prende un sottoinsieme di tutti i dati durante una iterazione

10.4.2 Funzione di costo (o di Loss) del discriminatore

$$J^{(D)}(\theta^{(D)}, \theta^{(G)}) = -\frac{1}{2} E_{x \sim p_{data}} \log D(x) - \frac{1}{2} E_z \log(1 - D(G(x))) \quad (10.13)$$

Se D(x) è vicino a 1 e il log = 0 non si paga tanto quindi il discriminatore sta agendo bene.

- Prima componente: indica che si viene penalizzati quando un esempio reale viene etichettato come non reale
- Seconda componente: prende il valore medio del valore atteso secondo la distribuzione z→(1-D(G(x)). Quando il discriminante pensa che G(x) sia reale allora il valore è vicino a uno e il log vicino a 0 (quindi -∞) e si paga pegno quando il discriminatore dice che l'esempio è reale: ovvero vicino a 1.

Questa funzione non è altro che la **cross-entropy** applicata al problema che si sta cercando di risolvere.

10.5 Cross entropy

Misura quanti bit in media si devono mandare quando il codice che si sta inviando è ottimizzato per una distribuzione q mentre i dati seguono la distribuzione p.

$$H(p, q) = -E_p[\log q] = -\sum p(x) \log q(x) \quad (10.14)$$

In genere misura la lunghezza media del messaggio che è più lunga di quella ottimale. La differenza tra cross entropy e l'entropia ottimale è la divergenza KL. Nel caso ideale la cross entropy si riduce all'entropia $H(p)$. Succede quando p diventa uaguale a q.

La minimizzazione della cross entropy corrisponde all'ottimizzazione del modello per la distribuzione p.

La cross entropy per la classificazione binaria (il caso delle GAN) ha la seguente formula:

$$CrossEntropy(y, \hat{y}) = -E_x[y \log \hat{y} + (1 - y) \log(1 - \hat{y})] \quad (10.15)$$

Per le GAN:

Con $y = 1$ significa che è l'esempio reale e l'y corrispondente è $D(x)$ dove x è l'esempio reale in questione.

Con $y = 1$ il secondo termine scompare e rimane quindi il primo termine: $E_x[\log \hat{y}]$.

Con $y = 0$ si ha a che fare con un esempio fake.
Con $y = 0$ il primo termine scompare e la predizione è data da $\hat{y} = D(G(z))$ dove z fa parte del rumore casuale dato in input a G.

10.6 Funzione di costo (o di Loss) del generatore

Trattandosi di un **gioco a somma nulla** è semplicemente l'inverso della loss del discriminatore. Possiamo definire:

$$V(\theta^{(D)}, \theta^{(G)}) = -J^{(D)}(\theta^{(D)}, \theta^{(G)}) \tag{10.16}$$

Quindi tutto il gioco può essere formalizzato in una funzione **minimax**:

$$\theta^{(G)*} = \arg \min_{\theta^{(G)}} \max_{\theta^{(G)}} V(\theta^{(D)}, \theta^{(G)}) \tag{10.17}$$

Questa formula si occupa di trovare il massimo secondo $\theta^{(G)}$ e il minimo secondo $\theta^{(D)}$: come risultato si trova il **punto di equilibrio di Nash**[4] della componente definita.

Questo setup per il gioco è facilmente analizzabile teoricamente (teoria dei giochi) infatti il gioco corrisponde a minimizzare la divergenza di **Jensen-Shannon**:

$$JSD(P||Q) = \frac{1}{2}KL(P||M) + \frac{1}{2}KL(Q||M) \; dove \; M = \frac{1}{2}(P + Q) \tag{10.18}$$

In questo caso il gradiente della loss è basso quando il discriminatore rifiuta gli esempi del generatore con un'alta confidenza quindi il generatore non potrà più migliorare.
Per risolvere il problema della loss (ovvero che si giunge ad un punto in cui il generatore non può più migliorare) si usa **loss euristica non-saturatig** del generatore.

$$J^{(G)} = -\frac{1}{2}E_z \log D(G(z)) \tag{10.19}$$

Si tratta di una **loss di tipo euristico** la cui idea di fondo è quella di dare al generatore tutta l'informazione possibile per poter riuscire ad apprendere il più rapidamente possibile.
In pratica la formula 10.19 riprende la parte che definiva la cross-entropy nella formula del discriminatore (10.13), ovvero il secondo termine dopo il $-$.

[4]Nash ha dimostrato che i giochi a somma nulla hanno una soluzione ottima chiamata equilibrio di Nash ed è un punto nello spazio ($\theta(G)$, $\theta(D)$) tale che è un minimo locale per $J(D)$ rispetto a $\theta(D)$ e un minimo locale per $J(G)$ rispetto a $\theta(G)$.

Quindi: loss discriminatore rimane invariata, mentre quella del generatore diventa la formula 10.19 che penalizza il generatore quando il discriminatore dirà 0 per un'immagine generata dal generatore.
Quando si è molto bravi a generare immagini che il generatore non sa più distinguere il gradiente avrà valore pari a 0 ma tanto si è già molto bravi.
Quando il discriminatore capisce esattamente quali sono le immagini generate del generatore la loss è alta e si riesce a migliorare rapidamente.

Cambiando la loss del generatore (quella del discriminatore rimane uguale a $J^{(D)}$) quindi non si ha più un gioco a somma nulla.

10.7 Perchè le GAN funzionano?

Buone performance grazie alla minimizzazione della divergenza Jensens-Shannon e non dalla divergenza KL quindi non è simmetrica. La maximum likelihood minimizza $KL(p_{data}|p_{model})$. Quindi anche minimizzare JS è simile a minimizzare $KL(p_{data}|p_{model})$.

Di preciso non si sa perché le GAN funzionino bene (IO BOH BASITO). **Di certo si sa che le GAN FUNZIONANO BENE**.

10.7.1 DCGAN - Deep Convolution GAN

Modello convolutivo associato alle GAN
La caratteristica principale è la presenza di un livello di **batch normalization** sia nel **G** che in **D**: applica la normalizzazione[5] dei dati non solo prima di darli alla NN (come si faceva di solito), ma lo fa **batch a batch** permettendo alla rete di convergere prima.
Viene calcolato il fattore di normalizzazione su ciascun batch calcolando i fattori livello per livello
La rete ad ogni passo si normalizza.

G e **D** vengono normalizzati separatamente.
Ultimo livello di D e primo livello di D non hanno il batch normalization: permette al discriminatore di **imparare la distribuzione dei dati** nei suoi primi livelli per poi applicare la batch normalization successivamente.

DCGAN non hanno livelli di pooling e nemmeno di unpooling[6]: come risultato si ottiene che non si va a ridurre la grandezza, ma ad ingrandirla.

[5] Normalizzazione: consiste nel limitare l'escursione di un insieme di valori entro un certo intervallo predefinito

[6] unpooling serve ad annullare le operazioni eseguite nel pooling

10.7.2 WGAN - Wasserstein GAN

Sfruttano una loss basata sulla **distanza di Wasserstein** per ovviare al problema che si presenta nei casi reali: difficoltà nel far convergere. La distanza prima citata si può dimostrare che abbia proprietà migliori per la convergenza rispetto alle GAN.

Distanza di Wasserstein:

$$W(p,q) = \inf_{\gamma \in \prod(p,q)} \mathbb{E}_{(x,y)\in\gamma} ||x-y|| \tag{10.20}$$

Putroppo l'infimum della formula non è sempre trattabile per tutte le distribuzioni: si ovvia al problema sfruttano una famiglia di funzioni continue rispetto a Lipschitz. Questo accorgimento permette di calolare W andandolo a semplificare la formula

Vantaggi WGAN

- **metrica loss più interpretabile**: quando la f si avvicina all'ottimalità si può mostrare che è una stima buona della earth-mover distance[7] che dipende dalla costante c.
- **addestramento di D**: lo si addestra fino ad una situazione di quasi ottimalità, una volta raggiunta tale situazione non si addestra più il discriminatore, ma si addestra G con una particolare loss semplificata
- l'evidenza empirica mostra che questa **metrica è correlata con la qualità degli esempi generati**: al migliorare delle immagini, la loss decresce (ecco la correlazione). Nelle GAN questo non accade
- **addestramento robusto e più semplice**

10.7.3 Trucchi per un buon addestramento

è utile addestrare una rete con informazioni supervisionate, ovvero con esempi etichettati.

Fare **one side label smoothing**: sostituzione etichetta positiva (1) con un valore leggermente più piccolo: $1-\alpha$. Così si penalizza il discriminatore quando dà valori troppo vicino ad 1. Funziona perchè funge da termine di regolarizzazione rendendo le reti resistenti agli attacchi avversari[8]

[7] Earth-mover distance: se ho 2 distribuzioni e voglio calcolare la DISTANZA, immaginare che queste 2 siano CUMULI DI TERRA che voglio spostare. La distanza È LA MINIMA QTA DI TERRA CHE POSSO SPOSTARE per far si che le 2 distrib. diventino uguali.

[8] Cambiamento dell'immagine con un rumore senza rendere tale cambiamento percettibile all'occhio, ma causa la modifica dell'output della rete passando da una classe x ad una y

Quando i batch sono troppo piccoli[9] c'è il rischio che introducano del rumore dannoso. Come soluzione si può fare il **Virtual Batch Normalization**: applicare la normalizzazione su un batch più grande utilizzato in parallelo quando si deve normalizzare (si usa sia questo che quello piccolo).

[9] Spesso succede per limiti hardware