DIETI. UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II
DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE DELL'INFORMAZIONE

Appunti di Deep Learning and Neural Networks

Mario Turco, Giuseppe Rauso

Appunti

2021

Note Preliminari

Questi appunti sono stati scritti durante l'anno di corso 2020/2021 e non si intendono come sostitutivi delle lezioni e del materiale didattico fornito dal docente.

Contents

CHAPTER 1

Introduzione

Data: 08/03/2021

Roberto Prevete, email: rprevete at unina.it

Materiale di riferimento: lezioni registrate, appunti delle lezioni, materiale online, testo (Neural networks for pattern recognition di Bishop)

Modalità di esame: progetto da svolgere in gruppo di 2-3 elementi(con tesina di 20-30 pagine) + orale

Contenuti principali

1. Feed-Forward Neural Network (Shallow Neural Networks)
    - Funzione di errore
    - Back Propagation
    - Regole di aggiornamento
    - Strategie di Learning
    - Iper-Parametri
2. Deep Learning
    - Reti convoluzionali
3. Auto-Encoders
    - Variational Auto-Encoders
4. Reti Ricorrenti
    - CTRNN
    - LSTM
    - Elmann-Jordann
5. XAI (eXplanable Artificial Intelligence)

PART I

The First Part

CHAPTER 2

Reti Neurali

2.1 Reti Neurali Artificiali

Con l'espressione reti neurali artificiali facciamo riferimento ad una quantità immensa di modelli diversi, con caratteristiche anche completamente diverse (Feed-Forward, Reti Ricorrenti, Reti a tempo continuo, reti a tempo discreto, reti continue sullo spazio, ...). Allo stesso tempo però ci sono vari elementi in comune:

- Hanno tutti la stessa unità di calcolo elementare (il **Neurone**)
- Ogni modello ha molteplici neuroni
- I neuroni interagiscono tra di loro

Il tutto è ispirato al funzionamento del sistema nervoso centrale e, l'idea del simulare tale sistema, risale agli anni 40 del 1900. Si è osservato che il cervello è capace di fare cose sorprendenti rispetto alla tecnologia che possediamo, ad esempio: un drone non riesce a volare come una mosca nonostante la mosca sia molto più semplice ed abbia meno potenza di calcolo di un drone.

Alcuni di questi modelli sono nati ad hoc per risolvere dei problemi, mentre altri modelli sono stati creati proprio per riuscire a capire meglio come funziona il cervello umano, vedi Modello Standard della visione di Tommaso Poggio che cercava di capire come funzionavano le prime aree visive del cervello umano ed, allo stesso tempo, era un modello in grado di riconoscere immagini ed effettuare task di classificazione (inizio anni 2000).

In passato, nell'ambito dell'intelligenza artificiale era divisa in varia categorie, tra cui il machine learning (che contiene anche le neural network), la logica, i sistemi simbolici, etc.. Oggi, le tecniche di machine learning sono diventate tantissime ed è diventato un argomento predominante nell'IA ed allo stesso modo le reti neurali sono diventate un argomento predominante del machine learning. Oggi quando si parla di Intelligenza Artificiale ci si riferisce spesso alle reti neurali anche se di fatto, le neural network sono solo una parte del machine learning che a sua volta è una parte dell'IA.

Negli ultimi anni c'è stato un grande successo dell'approccio delle reti neurali a causa delle idee innovative e proposte nuove portate dal Deep Learning, con molto successo in diversi campi (classificazione, regressione, ...). Il deep learning, inoltre, presenta una notevole facilità di utilizzo, cosa che ha contribuito al

suo successo.
Tuttavia, spesso, il problema di alcuni modelli è che, anche se funzionanti, tali modelli non sono facilmente spiegabili, hanno un comportamento poco trasparente.

Tipologie di problemi

Abbiamo 3 tipologie principali di problemi:

- Classificazione, voglio decidere se un input appartiene ad un determinata classe
- Regressione, voglio fare una previsione su una variabile di tipo continuo
- Controllo (o Governo,che è leggermente differente), ad esempio ho un robot in un ambiente sconosciuto voglio che sia in grado di navigarlo e di raggiungere un goal

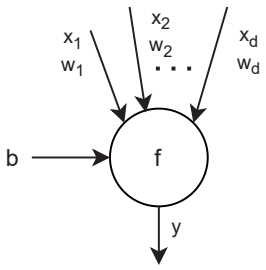
Data: 10/03/2021

2.2 Il neurone

Esistono tantissime architetture differenti di reti neurali artificiali che hanno degli elementi condivisi, in particolare: l'unità di calcolo elementare di tutte le reti è il **neurone** ed in tutte c'è una forte interazione tra tali unità.
Ed esiste una sorta di computazione generale, dove ogni neurone da un contributo alla computazione generale.
I vari modelli di rete si differenziano nel modo in cui questi neuroni sono costituiti e come questi interagiscono tra di loro.

La struttura del neurone

N.B. questa è la struttura che viene storicamente dal neurone di McCulloch e Pitts? degli anni '40 del 900; tuttavia esistono altri modelli in cui la struttura è leggermente diversa. Questa va intesa come una struttura generale.



Struttura del neurone

Abbiamo un'**unità di calcolo** che riceve d **linee di ingresso**(chiamate anche connessioni), su ciascuna linea di ingresso abbiamo una variabile di ingresso $x_1, \ldots, x_d$ ed in particolare con $x_i \in R$ (assumono valori reali).
Ogni neurone ha una **linea di uscita** a cui è associata una variabile di uscita con $y \in R$.
Ad ogni unità di calcolo è associata una **funzione di attivazione** $f : R \to R$.
Ad ogni linea di ingresso x_i è associato un **peso** $w_i \in R$ ed un **bias** $b_i \in R$.

Comportamento del neurone

C'è una computazione che avviene in due step:

1. Calcolo dell'input del neurone facendo una somma pesata dei valori di ingresso:

$$a = \sum_{j=1}^{d} w_j x_j + b$$

2. Calcolo dell'output:

$$y = f(a)$$

Possiamo fare subito due osseravazioni:

- Fissati i valori $x_1, \ldots, x_d$ e la funzione di attivazione f, il risultato di tale computazione dipende unicamente da i pesi ed il bias.
  Cambiando i pesi ed il bias ottengo un valore diverso,*i pesi ed il bias di un neurone determinano il comportamento del neurone stesso.* In genere

si fissa f ed il lavoro sarà nella scelta dei pesi e dei bias adatti.

- Al contrario, fissati pesi, bias e funzione di attivazione, il neurone realizza un mapping funzionale F, ovvero una funzione

$$NET_{\underline{w},b} : x \in R^d \rightarrow y = NET_{\underline{w},b}(x) \in R$$

Quindi NET è una funzione parametrizzata sui pesi e sui bias.

Al variare dei pesi e del bias, ottengo funzioni diverse.
L'obiettivo di un processo di learning è quello di trovare gli appropriati[1] pesi e bias.

Nota sul caso omogeneo I pesi ed il bias hanno un motivo storico, nascono dalla volontà di modellare il neurone biologico dove esiste il concetto della forza della connessione ed il concetto della soglia di attivazione di un neurone.
Però, in effetti, bias e pesi posso vederli come la stessa cosa, utilizzando il caso omogeneo, utilizzando:

$$w_0 = b$$

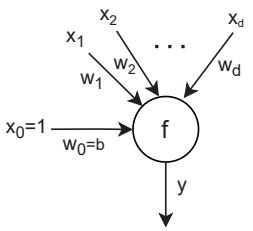
$$x_0 = 1$$

$$w = (b, w_1, \ldots, w_d)$$

$$x = (1, x_1, \ldots, x_d)$$

ed a diventerebbe:

$$a = \sum_{j=0}^{d} w_j x_j$$



Il bias può essere visto, a tutti gli effetti come il peso di una connessione che riceve 1 come variabile di ingresso.
D'ora in avanti parleremo direttamente solo di pesi, senza perdere di generalità, includendo automaticamente il bias nei pesi.

Calcolo matriciale

Il comportamento e l'output del neurone può essere anche visto in termini di **prodotti matrice-matrice**.
Da un punto di vista formale, posso riscrivere il tutto in forma matriciale, ovvero come prodotti matrice-matrice o matrice-vettore.
Posso considerare:

$$\underline{w} = (w_1, w_2, \ldots, w_d) \in R^{1 \times d}$$

$$\underline{x} = (x_1, x_2, \ldots, x_d) \in R^{1 \times d}$$

[1] In seguito specificheremo cosa significa "appropriati" per pesi e bias

$$b \in R^{1\times 1}$$

Posso scrivere il prodotto come prodotto matriciale tra w ed x^T che sarà proprio uguale ad $\sum_{j=0}^{d} w_j x_j$ e possiamo scrivere:

$$a = \underline{w} \cdot \underline{x}^T + b = \big(w_1, w_2, \ldots, w_d\big) \cdot \begin{pmatrix} x_1 \\ x_2, \\ \ldots \\ , x_d \end{pmatrix} + b = \sum_{j=0}^{d} w_j x_j + b$$

La funzione di attivazione

Possiamo scegliere tra varie funzioni di attivazione e questa scelta dipende dal neurone, dal tipo di architettura, etc. . .
Le funzioni più usate sono:

- Identità
- Sigmoide (Sigmoidali)
- Heaviside
- Rettificate (Deep)

Funzione identità

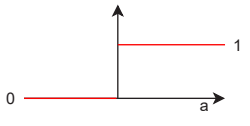
$$y = f(a) \Longrightarrow y = a \tag{2.1}$$

Funzione Sigmoide Quando a tende a $+\infty$ $s(a)$ tende a 1, quando a tende a 0 $s(a)$ tende a 0.5, quando a tende a $-\infty$ $s(a)$ tende a 0.

$$f(a) = s(a) = \frac{1}{1 + e^{-a}} \tag{2.2}$$

Funzione Heaviside Assume un valore pari ad 1 quando $a > 0$ ed assume valore 0 negli altri casi.

$$f(a) = H(a) = \begin{cases} 1 & \text{if a>0} \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$



Data: 15/03/2021

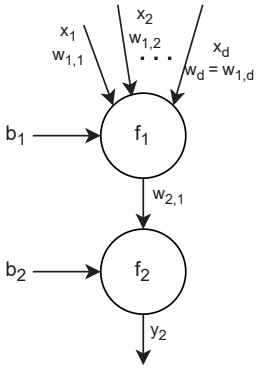
Dal singolo neurone a più neuroni

Anche qui esistono diverse soluzioni, noi vedremo una delle più usate, ovvero le reti *Feed Forward* (ma esistono soluzioni diverse che portano a soluzioni architetturali diverse).

2.3 Reti Feed-Forward

2 Neuroni

Supponiamo di avere un neurone che riceve d input con d pesi, un bias ed una funzione di attivazione. Possiamo supporre che l'output del primo neurone va a diventare l'input del secondo neurone (che supponiamo avere solo questa linea di input).
Il secondo neurone avrà una sua funzione di attivazione ed un suo output.



Dobbiamo formalizzare per poter distinguere la f di un neurone da quello dell'altro, distinguere i bias, i pesi, etc.. I neuroni possiamo identificarli con neurone1 e neurone2, e quindi chiamare le funzioni di attivazione rispettivamente: f_1, f_2. I pesi invece diventeranno:

$$w_1 = w_{1,1}; w_2 = w_{1,2}; w_d = w_{1,d}$$

Dove il primo indice identifica il neurone ed il secondo indice identifica la linea d'input. Identifichiamo l'output del neurone 1 come $w_{2,1}$.
Possiamo dire che c'è un ordine definito dalle connessioni, in questo caso, il neurone1 viene prima del neurone2 poichè c'è un verso nelle connessioni.
In termini di grafi, questo è un grafo aciclico, quindi posso stabile una relazione d'ordine parziale sui vari elementi del grafo.

Il fatto che ci sia un ordine tra i neuroni indica che il flusso di computazione segua tale ordine, nel nostro caso:

1. calcolo a_1 (input del primo neurone)
2. calcolo y_1 (output del primo neurone)
3. calcolo a_2 (input del secondo neurone)
4. calcolo y_2 (output del secondo neurone)

c'è un flusso di computazione in avanti (*Forward Propagation*). Quindi, in questo ordine :

$$a_1 = \sum_j w_{1,j} x_j + b_1$$

$$y_1 = f_1(a_1)$$

$$a_2 = w_{2,1} y_1 + b_2$$

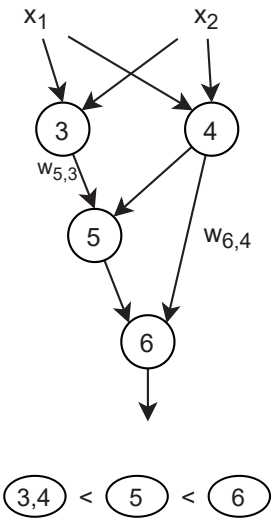
$$y_2 = f_2(a_2)$$

Più neuroni

Possiamo estendere tale ragionamento a più neuroni. Supponiamo si avere d variabili di ingresso $x_1, x_2, \ldots, x_d$ e supponiamo di avere n neuroni che possiamo numerare: $d+1, d+2, \ldots, d+n$, supponiamo che i neuroni siano collegati in modo tale da non avere cicli, esiste quindi un ordine parziale.

I primi due neuroni (3,4) ricevono input solo da x_1 ed x_2 e sono 'i primi' però tra di loro non c'è un ordine (ecco perchè abbiamo un ordinamento parziale). Mentre il neurone 5 viene dopo 3 e 4 ed il 6 dopo il 5.

Se i neuroni sono collegati in modo aciclico, posso sempre definire un ordine parziale e posso quindi stabilire un ordine di computazione ($1 \rightarrow 2 \rightarrow 3$) dove ogni passo successivo attende il termine della computazione del passo precedente.



Il calcolo dell'output di un neurone è, $\forall i \in [d{+}1, d{+}n]$:

1. $a_i = \sum_j w_{i,j} y_j + b_i$
2. $y_i = f_i(a_i)$

Dove $w_{i,j}$ è il peso della connessione che parte dal neurone j ed arriva al neurone i (ovvero al contrario di come si indicherebbe il peso di un arco di un grafo). j scorre su tutti gli indici dei neuroni che inviano connessioni ad i, mentre con y_j indico i valori di uscita del neurone j quando $j > d$ altrimenti $y_j = x_j$
La sequenza di computazione segue l'ordine parziale tramite una propagazione dell'input(Forward Propagation).

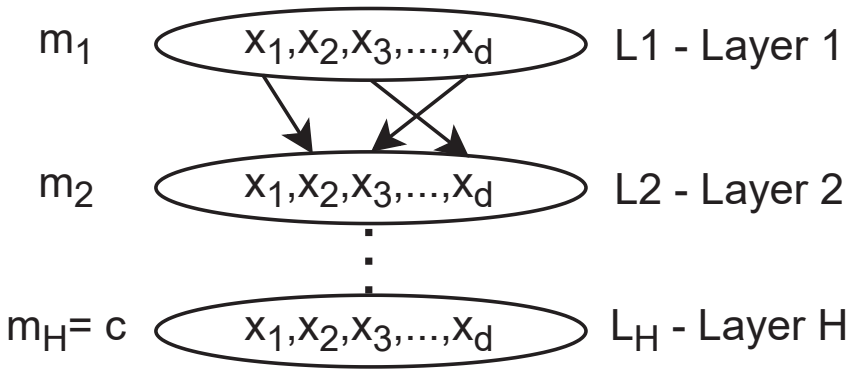
Dato questo tipo di schema posso suddividere le variabili in tre blocchi:

- variabili di ingresso (Input)
- nodi di output che non inviano connessioni a nessuno (nodo 6)
- nodi nascosti che sono i rimanenti neuroni (cioè quelli interni)

Esempio di computazione

La computazione viene svolta nel seguente modo:

1. $y_3 = f_3(w_{3,1}x_1 + w_{3,2}x_2 + b_3), y_4 = f_4(w_{4,1}x_1 + w_{4,2}x_2 + b_4)$
2. $y_5 = f_5(w_{5,3}y_3 + w_{5,4}y_4 + b_5)$
3. $y_6 = f_6(w_{6,5}y_5 + w_{6,4}y_4 + b_6)$



Struttura di una rete multi-strato

Questo tipo di architettura è importante perchè se fisso i pesi, bias e funzioni di attivazione, dati dei valori x_1 ed x_2 io avrò sempre lo stesso valore di output per gli stessi input.
Se considero una rete feed forward con d valori di input e c neuroni di output, allora tale rete realizza un mapping funzionale $F : x \in R^d \to y \in R^c$ dove x sono i possibili valori di ingresso della rete ed y sono i possibili valori di uscita. Esistono anche reti neurali (tipo le reti ricorrenti) in cui non è molto trasparente il fatto che la rete sia un mapping funzionale. In linea teorica, con una rete posso esprimere qualsiasi funzione continua, il problema è capire qual è la funzione giusta (qual è la funzione che ci può essere utile per il nostro problema).

Reti Multi Strato

In particolare noi vederemo una sottoclasse delle reti Feed Forward ovvero le reti Multi Strato (Multi Layer). L'idea è di organizzare i neuroni in gruppi, detti strati, ordinati tra loro. Ognuno dei neuroni riceve connessioni solo da elementi dello strato precedente quindi ogni neurone i appartenente ad uno strato h, può ricevere connessioni solo dai neuroni appartenenti allo strato $h-1$.

In questo modo sto definendo una rete Feed Forward perchè tutti i neuroni ricevono connessioni dallo strato precedente e che ogni neurone computa parallelamente a quelli dello stesso strato. Abbiamo definito quindi un ordine parziale $L_1 < L_2 < \cdots < L_H$
Anche qui abbiamo uno strato di input, uno strato di output e degli strati nascosti.

17/03/2021

Reti Full Connected

In una rete multistrato, se i neuroni ricevono connessioni da tutti i neuroni dello strato precedente, essa è detta *Full Connected*. Spesso, le variabili di ingresso

$x_1, x_2, \ldots, x_d$ vengono chiamati neuroni di input che ricevono l'input e danno in output l'input stesso.
Indichiamo con:

- a_i^h l'input del neurone i-esimo appartenente allo strato h
- z_i^h l'output del neurone i-esimo appartenente allo strato h
- f_h la funzione di attivazione[2] relativa a tutti i neuroni dello strato h
- $w_{i,j}^h$ il peso della connessione che arriva al nodo i-esimo dello strato h dal neurone j-esimo dello strato $h-1$
- e b_i^h il bias del neurone i-esimo appartenente allo strato h

[2]La funzione di attivazione è relativa a tutto il livello, non al singolo neurone di un livello, per questo non si utilizza un indice di neurone ma solo un indice di livello (h).

Possiamo quindi formalizzare il comportamento della rete in questo modo:

$$\forall h \in [1,H], \forall i \in [1,m_h] a_i^h = \sum_{j=1}^{m_{h-1}} w_{i,j}^h z_j^{h-1} + b_i^h$$
$$z_i^h = f_h(a_i^h) \tag{2.4}$$

Il comportamento della rete è dato dalla applicazione dell'equazione 2.4 per $h = 1 \to h = 2 \to \ldots \to h = H$ Quindi parto dall'inizio fino ad arrivare alla fine.

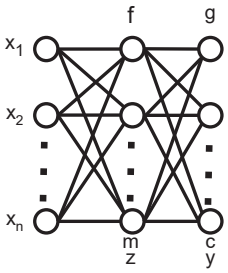
Ricorda: Questa rete realizza un mapping funzionale $F_\theta : x \in R^d \to y \in R^c$ Dove θ sono tutti i parametri (pesi e bias) della rete.

Shallow Networks

Dagli anni '80, partendo da questo tipo di struttura, se n'è scelta una ancora più semplice, le reti multi strano con uno strato interno, anche dette *Shallow Network*.

Abbiamo uno strato di input, uno strato interno ed uno strato di output.
Il calcolo dello strato interno sarà:

$$z_i = f(\sum_{j=1}^{d} w_{i,j}^1 x_j + b_i^1)$$
$$y_i = g(\sum_{j=1}^{m} w_{i,j}^2 z_j + b_i^2) \tag{2.5}$$



Una rete shallow

Questa struttura è un approssimatore universale perchè, in teoria, data una qualsiasi funzione continua in un insieme compatto, posso costruire una shallow network che la approssima quanto voglio.

Theorem 2.3.1 (Teorema di approssimazione universale)**.** *Sia f una funzione continua definita su un insieme compatto* [3]*. È possibile approssimare f con una rete definita come in (2.5) per un numero suffiente di nodi interni m.*

Questo teorema garantisce l'esistenza di una soluzione ma non ci premette di trovarla. Il *learning* (*processo di apprendimento*) è l'insieme di strategie che ci permette di trovare la soluzione o di avvicinarmi ad essa, trovando i parametri (pesi e bias) opportuni.

[3] Chiuso e limitato, ad esempio l'intervallo [5,8]

Note:

- Non tutte le funzioni di attivazione sono adatte
- Devo scegliere funzioni con le seguenti caratteristiche
  - Sullo strato di output: sufficienti le funzioni lineari[4], ad esempio la funzione identità.
  - Sullo strato interno: funzioni derivabili e non polinomiali, ad esempio la *sigmoide*.

Possiamo esprimere in maniera più formale il Teorema di approssimazione universale:

Theorem 2.3.2 (Teorema di approssimazione universale formalizzato)**.** *Data*

$$F : x \in D \subset R^d \rightarrow y = F(x) \in C \subset R^c$$

con D compatto e, data una rete con uno strato interno, come prima definita, con g, f che rispettano le proprietà prima enunciate, allora:

$$\forall \epsilon > 0, \exists m > 0, \exists \theta : ||F(x) - G_\theta(x)|| < \epsilon$$

Dove $G_\theta(x)$ è la funzione realizzata dalla rete con m neuroni interni e parametri (bias e pesi) θ

Formalizzazione in Matrici

Considero:

- $x = (x_1, x_2, \ldots, \mathbf{x}_d) \in R^{1\times d}$ l'input della rete come vettore riga
- $b^{(1)} = (b_1, b_2, \ldots, \mathbf{b}_m) \in R^{1\times m}$ il bias del primo livello
- $W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} & \ldots \\ \vdots & \ddots & \\ w_{m1}^{(1)} & & w_{md}^{(1)} \end{pmatrix} \in R^{m\times d}$ i pesi del primo strato[5]
- $W^{(1)} \cdot x^T = \begin{pmatrix} \sum_{j=1}^{d} w_{1j}^{(1)} x_j \\ \sum_{j=1}^{d} w_{2j}^{(1)} x_j \\ \vdots \\ \sum_{j=1}^{d} w_{mj}^{(1)} x_j \end{pmatrix}$

I passi di calcolo per il primo strato:

1. $a_1 = W^{(1)} \cdot x^T + b^{(1)T}$

[4] Ovvero se $g(a + b) = g(a) + g(b)$ ed $g(\alpha a) = \alpha g(a)$

[5] Sulla prima riga abbiamo i pesi del primo neurone, sulla seconda riga i pesi del secondo neurone, e così via

2. $z_1 = f_1(a_1) = \begin{pmatrix} f(\sum_{j=1}^{d} w_{1j}^{(1)} x_j + b_1^{(1)}) \\ f(\sum_{j=1}^{d} w_{2j}^{(1)} x_j + b_2^{(1)}) \\ \vdots \\ f(\sum_{j=1}^{d} w_{mj}^{1} x_j + b_m^{(1)}) \end{pmatrix}$

In maniera equivalente posso ragionare per il secondo stato:

$$W^{(2)} = \begin{pmatrix} w_{11}^{(2)} & w_{12}^{(2)} & \dots \\ \vdots & \ddots & \\ w_{c1}^{(2)} & & w_{cm}^{(2)} \end{pmatrix} \in R^{c \times m}$$

Calcolo:

- $a_2 = W^{(2)} z_1 + b^{(2)T}$
- $z_2 = f_2(a_2) = ...$

Data 22/03/2021

Per uniformare le varie operazioni, possiamo cambiare leggermente la nomenclatura.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_d \end{pmatrix} \in R^{d \times 1} \quad b^{(1)} = \begin{pmatrix} b_1^{(1)} \\ b_2^{(1)} \\ \dots \\ b_m^{(1)} \end{pmatrix} \in R^{m \times 1} \quad b^{(2)} = \begin{pmatrix} b_1^{(2)} \\ b_2^{(2)}, \\ \dots \\ , b_c^{(2)} \end{pmatrix} \in R^{c \times 1}$$

$$z^{(1)} = z \quad z^{(2)} = y$$

Di conseguenza le operazioni diventano:

1. $a_1 = w^{(1)} \cdot x + b^{(1)}$
2. $z_1 = f(a_1)$
3. $a_2 = w^{(2)} \cdot z + b^{(2)}$
4. $y = g(a_2)$

Implementazione in MATLAB

Matlab è organizzato in file (con estensione .m) ed ogni funzione accessibile si trova in file differenti.

Listing 2.1: newNet.m

```
function net=newNET(d,m,c)
    %INPUT:
    % d numero variabili in ingresso
    % m numero nodi interni
    % c numero nodi di uscita
    %OUTPUT:
    % net struttura che mantiene informazioni sulla rete
    SIGMA=0.1;
    net.W1=SIGMA*randn(m,d); %genera una matrice m * d con valori
        ↪ casuali estratti da
                            %una gaussiana centrata su 0 con deviazione
                                ↪ standard 1
    net.b1=SIGMA*randn(m,1);
    net.W2=SIGMA*randn(c,m);
    net.b2=SIGMA*randn(c,1);
    net.f=@miaSigma; %puntatore alla funzione sigma
    net.g=@miaIdentity;
    net.m=m;
    net.d=d;
    net.c=c;
end
```

Listing 2.2: miaSigma.m

```
function y=miaSigma(x)
    y= 1./ (1+exp(-x));
    %il punto serve ad indicare che la divisione va fatto per tutti i
        ↪ valori di x
    %per esempio se x=[-100 -10 -1 0 1 10 100];
    %oppure x=[-2:0.1:2]; cioe' da -2 a 2 con passi di 0.1
end
```

Listing 2.3: miaIdentity.m

```
function y=miaIdentity(x)
    y=x;
end
```

A questo punto possiamo lanciare il comando

```
>>miaNet=newNet(3,5,2);
>>miaNet
```

E visualizzeremo la struttura creata.
Scriviamo invece un programma che simula la rete

Listing 2.4: simNet.m (Esempio da non seguire)

```
function y=simNet(net,x)
    % INPUT:
    % new e' la struttura che mantiene le informazioni sulla rete
    % x sono i valori di input della rete come un vettore colonna dx1
    %%SENZA SFRUTTARE IL FORMALISMO CON MATRICE
    a1=zeros(1,m);
    for i=1:net.m %i scorre sui nodi interni
        for j=1:net.d %j scorre sulle variabili di ingresso
            a1(i)= a(i) + net.W1(i,j)*x(j); %calcolo la sommatoria
        end
        a1(i) = a1(i)+net.b1(i); %Aggiungo il bias
    end
end
```

Listing 2.5: simNet.m

```
function y=simNet(net,x)
    % INPUT:
    % new e' la struttura che mantiene le informazioni sulla rete
    % x sono i valori di input della rete come un vettore colonna dx1
    %%SFRUTTANDO LA NOTAZIONE MATRICIALE
    a1=net.W1*x;
    a1=a1+net.b1;
    z1=net.f(a1);
    a2=net.W2*z1+net.b2;
    y=net.g(a2);
end
```

Questa struttura è necessaria e sufficiente ad emulare qualsiasi rete ad un solo strato.
Possiamo simularla in questo modo:

```
>>miaNet=newNet(3,5,2);
>>x=randn(3,5);
>>y=simNet(miaNet,x)
```

Esercizio: generalizzare questa rete in modo che possa avere quanti strati voglio

Pesi e Bias

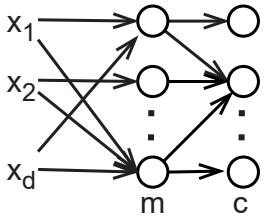
Il problema ora è come determinare i $W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$
In genere $\theta = \{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}\}$ sono detti parametri della rete, mentre m, numero nodi interni, funzioni di attivazione sono detti **iperparametri**.

In prima battuta, il processo di learning coinvolge i θ mentre gli iperparametri sono lasciati fissi e scelti in qualche modo[6].
L'idea è quella di farsi guidare dai dati che:

- Sono etichettati e che rispettano determinate proprietà (Apprendimento supervisionato)
- Non sono etichettati (apprendimento non supervisionato)

[6] In seguito vederemo meglio in che modo

2.4 Calcolo del gradiente



Data: 29/03/2021 Abbiamo la nostra rete full connected, un dataset $DS = \{(x^{(n)}, t^{(n)})\}_{n=1}^{N}$ con $x^{(n)} \in R^d$ $t^{(n)} \in \{0,1\}^C$ per problemi di classificazione o con $t^n \in R^d$ per problemi di regressione. Sia $E(\theta) = \sum_{n=1}^{N} E^{(n)}(\theta)$ la nostra funzione di errore. Il gradiente è:

$$\nabla E = [\frac{\partial E}{\partial \theta_1}, \frac{\partial E}{\partial \theta_2}, \ldots, \frac{\partial E}{\partial \theta_p}]$$

Il calcolo del gradiente si basa su l'algoritmo di Back-Propagation che permette di calcolare le derivate rispetto ai vari parametri in modo molto efficiente (lineare sul numero di parametri, $O(n)$ se n è il numero di parametri).

Nota: L'algoritmo di Back-Propagation non aggiorna i parametri, permette di calcolare il gradiente per poter poi aggiornare i parametri. Mentre sono le regole basate sulla discesa del gradiente che servono ad aggiornare i pesi

Calcolo delle derivate

Definition 2.4.1 (Regola delle funzioni composte)**.** $\frac{\partial E}{\partial w_{i,j}}$ Lo posso scrivere come derivata parziale rispetto a_i per la derivata parziale di a_i rispetto a w_{ij}

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial a_i} \cdot \frac{\partial a_i}{\partial w_{i,j}}$$

Dove $w_{i,j}$ è il peso che va da j ad i.

Esempio derivata delle funzioni composte:

$f(x) = (3x + 2)^2$, chiamo $a = (3x + 2)$, quindi $f(x) = a^2$. Allora $\frac{\partial f(x)}{\partial x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} = 2 \cdot a \cdot 3 = 6 \cdot (3x + 2)$

Osservazione 1 w_{ij} è coinvolto direttamente solo nel calcolo dell'input del nodo i, cioé a_i

Esempio 2: w_{ij} si troverà solo nel calcolo dell'input del nodo i, quindi sarà rilevante solo nel calcolo di a_i.
Allo stesso modo, la mia funzione di errore dipende da tutti i parametri ed in particolare anche da w_{ij} ed in particolare solo del nodo i. Quando vado a calcolare derivata di E rispetto a $w_{i,j}$:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial \sum_{n=1}^{N} E^{(n)}(\theta)}{\partial w_{ij}} = \sum_{n=1}^{N} \frac{\partial E^{(n)}(\theta)}{\partial w_{ij}}$$

Quindi, in particolare, per una qualsiasi n:

$$\frac{\partial E^{(n)}}{\partial w_{ij}} = \frac{\partial E}{\partial a_i^{(n)}} \cdot \frac{\partial a_i^{(n)}}{\partial w_{ij}} \tag{2.6}$$

$$a_i^{(n)} = \sum_h w_{ih} z_h^{n-1}$$

con h che scorre su tutti i nodi che inviano connessione ad i.

Quindi la funzione E che dipende dai valori di uscita e di input di tutti i nodi, si ritrova il calcolo di w_{ij} solo nel nodo i, quindi per la regola delle funzioni composte posso scomporre questa derivata.

Ricordando che $a_i^{(n)} = \sum_h w_{ih} z_h^{n-1}$ dove h corre su tutti i nodi che inviano connessioni ad i, allora:

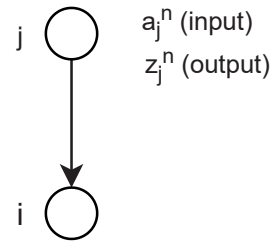
$$\frac{\partial a_i^{(n)}}{\partial w_{ij}} = \frac{\partial \sum_h w_{ih} z_h^{n-1}}{\partial w_{ij}} = \frac{\partial (w_{ih_1} z_{h_1}^{n-1} + w_{ih_2} z_{h_2}^{n-1} + \cdots + w_{ij} z_j^{n-1} + \ldots w_{ih_k} z_{h_k}^{n-1})}{\partial w_{ij}} =$$

$$0 + 0 + \cdots + \frac{\partial w_{ij} z_j^{n-1}}{\partial w_{ij}} + \cdots + 0 = z_j^{n-1} \implies \frac{\partial E^{(n)}}{\partial w_{ij}} = \frac{\partial E^{(n)}}{\partial a_i^n} \cdot z_j^{n-1}$$

Otteniamo quindi:

$$\frac{\partial E^{(n)}}{\partial w_{ij}} = \frac{\partial E^{(n)}}{\partial a_i^n} \cdot z_j^{n-1} \tag{2.7}$$

L'indice i compare una sola volta e tutti gli altri sono costanti(quindi 0 nella derivata) rispetto al nodo i, quindi resta soltanto z_j^{n-1}.



$$\delta_i = \frac{\partial E}{\partial a_i} \tag{2.8}$$

Ogni nodo ha il suo δ e quindi, sostituendo 2.7 e 2.8 nella 2.6 :

$$\frac{\partial E^{(n)}}{\partial w_{ij}} = \delta_i^n \cdot z_j^{n-1} \tag{2.9}$$

Quindi

1. Do in pasto alla rete la n-esima coppia del DS, calcolo tutti gli input ed output di tutti i nodi: a_i^n, z_i^n (con i che scorre tu tutti i nodi
2. calcolo tutti i δ_i^n
3. calcolo le derivate tramite la (2.9)

Come calcolare i $\delta_i^{(n)}$

$$\delta_i^{(n)} = \frac{\partial E^{(n)}}{\partial a_i^{(n)}} = \frac{\partial E^{(n)}}{\partial z_i^{(n)}} \cdot \frac{\partial z_i^{(n)}}{\partial a_i^{(n)}}$$

Per la regola di derivazione di funzioni composte (tra il primo ed il secondo uguale):

$$z_i^{(n)} = f(a_i^{(n)})$$

$$\delta_i^{(n)} = f'(a_i^{(n)}) \cdot \frac{\partial E^{(n)}}{\partial z_i^{(n)}}$$

$z_i^{(n)}$ può essere l'uscita di un nodo interno o l'uscita di un nodo di output.

Nodo di output

Per chiarezza, rinominiamo i con k per indicare un nodo di output $z_k^{(n)} \equiv y_k^{(n)}$

$$\frac{\partial E^{(n)}}{\partial z_k^n} \equiv \frac{\partial E^{(n)}}{\partial y_k^n}$$

Ma $E^{(n)}$ dipende direttamente da $y_1^{(n)}, y_2^{(n)}, \ldots, y_c^{(n)}$. Quindi questa derivata dipende dalla forma della funzione di errore. Ad esempio:

$$E^{(n)} = \frac{1}{2}\sum_k (y_k^n - t_k^n)^2$$

Allora:

$$\frac{\partial E^{(n)}}{\partial y_k^{(n)}} = \frac{1}{2} \cdot 2(y_k^{(n)} - t_k^{(n)}) = (y_k^{(n)} - t_k^{(n)})$$

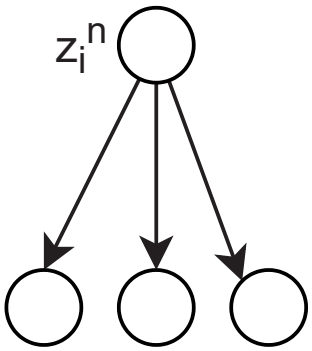
ed

$$\delta_k^n = g'(a_k^n) \cdot \frac{\partial E^{(n)}}{\partial y_k^n}$$

che nel nostro esempio equivale ad $g'(a_k^n) \cdot (y_k^n - t_k^n)$

Definition 2.4.2 (Back-propagation per i nodi di output)**.**

$$\delta_k^n = g'(a_k^n) \cdot (y_k^n - t_k^n)$$



Nodo interno

Riprendiamo la definizione di $\delta_i^{(n)}$:

$$\delta_i^{(n)} = f'(a_i^{(n)}) \cdot \frac{\partial E^{(n)}}{\partial z_i^{(n)}}$$

$z_i^{(n)}$ incide sul calcolo dell'input dei nodi dello strato successivo quindi, per la regola delle funzioni composte:

$$\delta_i^{(n)} = f'(a_i^{(n)}) \cdot \sum_j \frac{\partial E^{(n)}}{\partial a_j^{(n+1)}} \cdot \frac{\partial a_j^{(n+1)}}{\partial z_i^{(n)}} = f'(a_i^{(n)}) \cdot \sum_j \delta_j^{(n)} \frac{\partial(\sum_h w_{jh} z_h^n)}{\partial z_i^{(n)}} =$$

$$= f'(a_i^{(n)}) \cdot \sum_j \delta_j^{(n)} \sum_h \frac{\partial(w_{jh} z_h^n)}{\partial z_i^{(n)}} = f'(a_i^{(n)}) \cdot \sum_j \delta_j^{(n)} w_{ji}$$

j corre sugli indici dei nodi che ricevono connessioni dal nodo i[7]

Definition 2.4.3 (Back-propagation per i nodi interni)**.**

$$\delta_i^{(n)} = f'(a_i^{(n)}) \cdot \sum_j \delta_j^{(n)} w_{ji}$$

[7]Nota: $\sum_h \frac{\partial(w_{jh} z_h^n)}{\partial z_i^{(n)}} = w_{ji}$ perchè stiamo derivando rispetto a z_i quindi soltanto $\frac{\partial(w_{ji} z_i^n)}{\partial z_i^{(n)}}$ avrà un valore diverso da 0, ed in particolare, sarà uguale ad w_{ij}.

ESEMPIO 3 Sia $f(x) = 3x+5x$. Poniamo $a = 3x$ e $b = 5x$. Allora $f(x) = a+b$ e quindi:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial a} \cdot \frac{\partial a}{\partial x} + \frac{\partial f}{\partial b} \cdot \frac{\partial b}{\partial x} = 1 \cdot 3 + 1 \cdot 5 = 8$$

Per calcolare δ_i^n mi servono i delta degli strati successivi che vengono quindi propagati verso dietro (Back-Propagation) e non dipende dalla funzione di errore scelta a differenza del calcolo dei nodi esterni (? VEDI orario 12:35)

Data: 31/03/2021

Regole di aggiornamento di pesi e bias

Considerando la nostra solita rete, con il dataset e funzione di errore e tale funzione di errore dipende da θ ovvero i parametri della rete. Vogliamo calcolare le derivate di E rispetto al generico peso w_{ij}: $\frac{\partial E}{\partial w_{ij}}$ che abbiamo detto essere scrivibile come:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{n=1}^{N} \frac{\partial E^{(n)}}{\partial w_{ij}}$$

Come calcolare la derivata

1. **(Forward Step)** metto $\underline{x}^n$ in input alla rete e calcolo a_i^n, z_i^n, per tutti i nodi della rete

2. **(Back Propagation)** calcolo dei δ

   a) calcolo δ output $\delta_k^n = g'(a_k^n) \cdot \frac{\partial E^{(n)}}{\partial y_k}|_{y_k=y_k^n}$

   b) calcolo delta interni $\delta_i^n = f'(a_i^n) \cdot \sum_j w_{ji}\delta_i^n$ dove k scorre sugli incidici dei nodi che ricevono connessioni da h.

   cioè calcolo i delta dello stato precedente ai nodi di output, poi calcolo i delta dello strati ancora predente (se esiste) e così via

3. $\forall w_{ij}$ calcolo $\frac{\partial E^{(n)}}{\partial w_{i,j}} = \delta_i^n \cdot z_j^n$
   se j è una variabile di ingresso allora $z_j^n \equiv x_j^n$

4. Posso calcolare $\frac{\partial E}{\partial w_{ij}} = \sum_{n=1}^{N} \frac{\partial E^{(n)}}{\partial w_{ij}}$

Precisazione: se w_{ij} è un bias allora $j = 0$ ed $x_0 = 1$, $w_{i0} \equiv b_i$ ed $\frac{\partial E^{(n)}}{\partial b_i} = \delta_i^n$

Precisazione: questo algoritmo può essere applicato se:

1. ho una rete Feed Forward

2. le funzioni di attivazione devono essere derivabili

3. la funzione di errore deve poter essere scomposta come somma degli errori sulle singole coppie

4. la funzione di errore deve essere derivabile rispetto agli output

Con questo tipo di algoritmo si riducono di molto i tempi di calcolo.

Esempio:

Supponiamo di avere

$$f(x) = s(a) \equiv \frac{1}{1+e^{-a}} \quad g(x) \equiv \mathtt{I} \equiv x$$
$$E = \sum_{n=1}^{N} \frac{1}{2} \sum_{k=1}^{C} (y_k^n - t_k^n)^2$$

Nodi output:

$$\delta_k^n = g'(a_k^n) \frac{\partial E^{(n)}}{\partial y_k}|_{y_k \equiv y_k^n} = (y_k^n + y_k^n)$$

Nodi interni:

$$\delta_n^n = f'(a_k^n) \cdot \sum_k w_{kh} \delta_k^n$$
$$con \quad f'(a_k^n) = s'(a_k^n).$$

La derivata della sigmoide:

$$s'(a_k^n) = \frac{e^{-a_h^n}}{(1+e^{-a_h^n})}$$
$$= \frac{1+e^{-a_h^n}-1}{(1+e^{-a_k^n})^2}$$
$$= \frac{1+e^{-a_h^n}}{(1+e^{-a_k^n})^2} - \frac{1}{(1+e^{-a_k^n})^2}$$
$$= \frac{1}{(1+e^{-a_k^n})^2} - \left(\frac{1}{(1+e^{-a_k^n})}\right)^2$$
$$= s(a_k^n) - (s(a_k^n))^2 = s(a_k^n)(1-(a_k^n))$$

Con $s(a_k^n) = z_h^n$:

$$s'(a_k^n) = z_h^n(1-z_h^n)$$

Quindi

$$\delta_h^n = z_h^h(1-z_h^h) \sum_k w_{hk} \delta_k^n$$

Posso avere due tipi diversi di aggiornamento, sfruttando il calcolo delle derivate in due modi diversi, ottenendo:

- Online Learning: calcolo la derivata di ciascuna coppia del DS ed aggiorno direttamente ogni peso senza calcolare la derivata "totale"
- Batch Learning: calcolo le singole derivate degli elementi del DS, trovo la derivata totale ed alla fine aggiorno i pesi

(In realtà esiste anche una via di mezzo chiamata Mini Batch)

Online Learning

$\forall(\underline{x}^n, \underline{t}^n)$:

1. Calcolo derivata $\frac{\partial E^{(n)}}{\partial w_{ij}}$
2. Aggiorno i pesi

Batch Learning

$\forall(\underline{x}^n, \underline{t}^n)$:

1. Calcolo derivata $\frac{\partial E}{\partial w_{ij}} = \sum_n \frac{\partial E^n}{\partial w_{ij}}$
2. Aggiorno i pesi

Computazionalmente, l'online learning è quello più pesante perchè per ogni coppia devo anche aggiornare tutti i pesi, mentre nel batch calcolo direttamente la derivata totale ed aggiorno i pesi una singola volta.

Tuttavia, il nostro apprendimento non avviene facendo una sola volta il calcolo delle derivate e l'aggiornamento dei pesi ma, tale operazione, sarà fatta più volte perchè devo trovare la combinazione di valori che mi da il minimo errore che, processo che più richiedere più iterazioni(chiamate **Epoche**), fino a che non si verifica una certa condizione di uscita (ad esempio sono abbastanza vicino al minimo).

Sul singolo passo, l'Online Learning è più lento rispetto al Batch ma allo stesso tempo l'Online Learning potrebbe portare prima alla verifica della condizione di uscita. Inoltre il batch learning potrebbe occupare molto, troppo, spazio in memoria soprattutto se il dataset è molto grande ($\geq 10^5 - 10^6$coppie). La scelta di quale tecnica usare dipende dall'euristica e dal problema specifico.

Esiste una via di mezzo, il **Mini Batch**, che divide il DS in mini-batch e per ognuno di questi si calcola il gradiente e si aggiornano i parametri. Un'epoca termina quando l'algoritmo ha scorso tutto il dataset (diviso in mini-batch) e si ferma quando si verifica una condizione data, ad esempio per un numero di epoche fissato o quando l'errore sul validation set è accettabile (*early stopping*).

12/04/2021

Sintesi Gradiente

Il processo di aggiornamento della rete avviene in due macro fasi:

1. calcolo delle derivate della funzione di errore rispetto a pesi e bias → *back-propagation*
2. aggiornamento dei parametri tramite le derivate

$$w_{ij} = w_{ij} - \eta \frac{\partial E^{(n)}}{\partial w_{ij}}$$

La back propagation a sue volta avviene in due fasi:

1. Calcolo dei *delta*:
   a) Propagazione in avanti $\underline{x}^n$ quindi calcolo di a_i^n e z_i^nper ogni nodo
   b) Calcolo di δ nello stato output $\delta_k^n = g'(a_k^n) \cdot \frac{\partial E}{\partial y_k^n}$
   c) Calcolo di δ negli stati intermedii $\delta_h^n = f'(a_h^n) \cdot \sum_k w_{kh}\delta_k^h$
2. Calcolo derivate tramite i δ
   a) $\frac{\partial E^{(n)}}{\partial w_{ij}} = z_j^n \cdot \delta_i^n$

2.5 Back Propagation in MatLab

Listing 2.6: forwardStep.m

```
function [y, a2, z, a1] = forwardStep(net, x)
    a1 = net.W1 * x;
    a1 = a1+net.b1;
    z1=net.f(a1);
    a2=net.W2*z1+net.b2;
    y=net.g(a2);
end
```

Listing 2.7: miaBackProp.m

```
function [derivW1, derivW2, derivaBias_hidden, derivBias_out] =
          miaBackProp(net,x, t, derivFunActOutput, deriFunActHidden,
                      derivFunErr)
    % x vettore d x 1
    % t target c x 1
    % y  c x 1
    % a2 c x 1
    % z1 m x 1
    % s1 m x 1

    %% FASE FORWARD-PROPAGATION

    % Calcolo tutti gli input ed output di tutti i nodi:
    [y, a2, z1, a1] = forwardStep(net, x);

    % ci serve la derivata prima della funzione di attivazione e la
        ↪ derivata prima della funzione di errore, quindi li
        ↪ prendiamo come input


    %% FASE BACK-PROPAGATION (calcolo delta)

    % Calcolo dei nodi di uscita:
    delta_out = derivFunActOutput(a2);
    % derivFunActOutput(a2) mi restituisce un vettore c x 1
    % come derivFunErr(y, t)

```

```
26      delta_out = delta_out .* derivFunErr(y, t);
27      % .* prodotto element-wise
28      % delta_out di dimensione c x 1
29
30      % net.W2 e' una matrice (dai nodi interni a quelli di uscita)
31      % ha dimensione c x m
32      delta_hidden = net.W2' * delta_out;
33      % net.W2' m x c, delta_out c x 1
34      % ottenendo una matrice m x 1
35
36      % SE avessi piu' strati:
37      % delta_hidden_2 = net.W1 * delta_hidden;
38
39      delta_hidden = delta_hidden .* deriFunActHidden(a1);
40      % deriFunActHidden restituisce un vettore m x 1
41
42
43      %% CALCOLO DERIVATE
44
45      % Rispetto a W2 (c x m) quindi ho bisogno di una matrice di
            ↪ derivate c x m
46      derivW2 = delta_out * z1'; % (c x 1) * (1 x m) = (c x m)
47      % z1': z1 trasposto -> (m x 1)' = (1 x m)
48
49      % Rispetto a W1 (m x d)
50      derivW1 = delta_hidden * x'; % (m x 1) * (1 x d) = (m x d)
51
52      % ATTENZIONE: solo se l'input non e' un singolo vettore
53      derivBias_out = delta_out;
54      derivBias_hidden = delta_hidden;
55
56  end
```

Listing 2.8: discesaDelGradienteStandard.m

```
1   function net = discesaDelGradienteStandard(net, eta, derivW1, derivW2,
        ↪  derivBias_hidden, derivBias_out)
2
3       net.W1 = net.W1 - eta*derivW1;
4       net.W2 = net.W2 - eta*derivW2;
5       net.b1 = net.b1 - eta*derivBias_hidden;
6       net.b2 = net.b2 - eta*derivBias_out;
7   end
```

Esempi di derivate di funzioni di attivazione

Listing 2.9: derivFunActIdentity.m

```
1   function y=derivFunActIdentity(x)
2       % f(x) = x
3       % f'(x) = 1
4       y = ones( size(x) );
5       % una matrice di 1 della stessa dimensione di x
```

```
6 end
```

Listing 2.10: derivFunActSigmoide.m

```
function y=derivFunActSigmoide(x)
    z = miaSigma(x);
    y = z .* (1-z);
end
```

Listing 2.11: derivFunErrSquares.m

```
function z=derivFunErrSquares(y, t)
    z = y-t;
end
```

Data 14/04/2021

2.6 Parametri nella back propagation

Con la Back-propagation, vogliamo aggiornare i pesi tramite le derivate della funzione di errore rispetto ai parametri della rete utilizzando qualche regola di aggiornamento che a sua volta può avere degli iperparametri ed è a sua volta un processo iterativo.

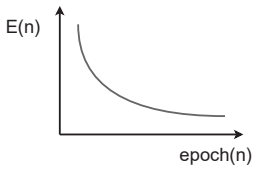
Listing 2.12: miaLearningPhase.m

```
function [min_err, netScelta] = miaLearningPhase(miaNet, N, x, t,x_val
    ↪ ,t_val derivFunActOutput, derivFunActHidden, derivFunErr)
    eta=0.1; %Il learning rate puo essere inserito anche in input alla
        ↪  funzione
    err=zeros(1,N);
    err=val(1,N);
    %inizializzo il valore minimo di errore a quello corrente%
    y_val=simNet(miaNet, x_val);
    min_err=sumOfSquares(y_val,t_val);
    netScelta=miaNet;
    for epoch=1:N %Faccio sempre tutte le operazioni
        %LEARNING ONLINE
        for n=1:size(x,2)
            [derivW1, derivW2, derivBias_hidden, derivBias_out]=
                ↪ miaBackProp(miaNet, n, x(:,n), t(:,n),
                ↪ derivFunActOutput, derivFunActHidden, derivFunErr)
             %Questa regola di aggiornamento si puo scegliere (magari
                 ↪ mettendola come input)
            miaNet=discesaDelGradienteStandard(miaNet, eta, derviW1,
                ↪ derivW2,     derivBias_hidden, derivBias_out)
        end
        %%BATCH LEARNING
%        [derivW1, derivW2, derivBias_hidden, derivBias_out]=
    ↪ miaBackProp(miaNet, n, x, t, derivFunActOutput,
    ↪ derivFunActHidden, derivFunErr)
%            %Questa regola di aggiornamento si puo scegliere (magari
    ↪  mettendola come input)
%           miaNet=discesaDelGradienteStandard(miaNet, eta, derviW1,
    ↪ derivW2,     derivBias_hidden, derivBias_out)
```

```
20          y=simNet(miaNet,x);
21          y_val=simNet(miaNet, x_val);
22          err(epoch)=sumOfSquares(y,t);
23          err_val(epoch)=sumOfSquares(y_val,t_val);
24          if err_val(epoch) < min_error
25              min_err=err_val(epoch);
26              netScelta=miaNet;
27          end
28      end
29      %restituisco la rete con minor errore di validazione
30  end
```

Sui dati su cui faccio training, se è stato fatto tutto correttamente, l'errore diminuisce sempre.

Figure 2.1: Andamento tipico di E(n) rispetto alle epoche

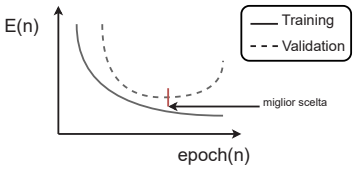


Generalizzazione Tuttavia, questo non è il nostro unico obbiettivo, vogliamo invece che la rete risponda bene anche su dati nuovi su cui non è stato fatto training. Per quantificare la capacità della rete di generalizzare, divido il dataset in due parti: training set e validation set.

Generalizzazione Il training set è utilizzato per aggiornare i parametri della rete, il validation set è usato per verificare la capacità di generalizzare. Ad ogni epoca avrò una rete diversa con un errore diverso, scelgo la rete con minimo errore sul validation set (e non è detto che sia quella corrispondente all'ultima epoca).

Scelta dei dati È importante che il validation set ed il training set siano rappresentativi della popolazione che sto studiando. Una scelta possibile è quella di estrarre i due set in maniera randomica dal dataset. Il training set ha una numerosità/cardinalità maggiore rispetto al validation set (ad esempio 70% 30%) perchè abbiamo bisogno di più dati possibili per aggiornare i parametri.

Figure 2.2: Un possibile andamento di E(n) sul validation set



Valutazione della rete Mi serve una terza porzione di dati, chiamata test set. Una volta aver ottenuto la rete migliore nella fase di training, scegliendola sul validation set, la valuto sul test set utilizzando la stessa funzione di errore e/o altre metriche. Anche in questo caso, il test set deve essere opportunamente scelto. (vedi

pagina 294 del Bishop)

Funzioni di errore

Per i problemi di regressione:

$$E = \sum_{i=1}^{n} E^{(i)}$$

$$E^{(n)} = \frac{1}{2} \sum_{k} (y_k^n - t_k^n)^2$$

Per problemi di classificazione invece, possiamo usare la Cross Entropy. Nel caso di due classi:

$$E^{(n)} = t_k^n \log y_k^n + (1 - t_k^n) \log(1 - y_k^n)$$

Nel caso di più classi:

$$E^{(n)} = \sum_{k=1}^{c} t_k^n \log y_k^n$$

Entrambe le formule vogliono massimizzare la verosimiglianza della risposta della nostra rete e si differenziano perchè, nella prima t è considerata una variabile continua, nella seconda viene considerata discreta.
Ovviamente esistono altre funzioni di errore, queste sono due delle più usate.

Data 19/04/2021

2.7 Riassunto: funzioni di errore e di attivazione

Per problemi di regressione abbiamo:

- una variabile continua (target)
- funzione di errore (Somma di quadrati):

$$E = \sum_{n=1}^{w} E^{(n)}$$

$$E^{(n)} = \frac{1}{2} \sum_{k} (y_k^n - t_k^n)^2$$

Per problemi di classificazione:

- una variabile discreta (target)
- funzione di errore, Cross Entropy:

$$E = \sum_{n=1}^{w} E^{(n)}$$

$$E^{(n)} = \sum_{k=1}^{c} t_k^n \log_e y_k^n$$

Per problemi a due classi con un solo neurone in uscita:

$$E^{(n)} = -(t_k^n \log_e y_k^n + (1 - t_k^n) \log_e (1 - y_k^n))$$

Con $0 < y_k^n < 1$ (Deve essere interpretato come probabilità). Per esempio la funzione di attivazione del neurone di output potrà essere la sigmoide.

Derivata funzione Cross Entropy

La derivata della funzione di cross entropy è la seguente:

$$\frac{\partial E_c^{(n)}}{\partial y_n} = \frac{\partial(-t_h^n \log y_h^n)}{\partial y_h} = -\frac{t_h^n}{y_h^n} = \sum_n \frac{\partial E_c^{(n)}}{\partial y_n}$$

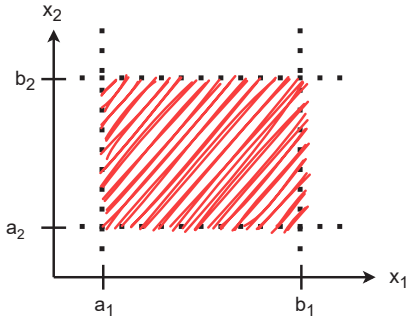
2.8 Maledizione della dimensionalità

Si può dimostrate l'esistenza della **Maledizione della dimensionalità**, ovvero la dimensione di un dataset cresce in maniera esponenziale rispetto a d (dimensione input).

Se ho $x = (x_1, \dots x_d)$ e le componenti sono indipendenti tra di loro (o non posso fare delle ipotesi di dipendenza a priori).

Esempio

Se x_1 è nell'intervallo (a_1, b_1) ed x_2 è nell'intervallo (a_2, b_2), allora i possibili valori che assumeranno sono contenuti nell'area evidenziata.
Però se non posso fare ipotesi di dipendenza, allora il dataset deve ricoprire uniformemente tutto il dominio dei possibili valori affinché sia statisticamente significativo. Allora se considero una numerosità di $k = 10$ per ciascuna variabile un "buon campione" sarebbe:



- 10 valori per x_1
- $10 \cdot 10 = 10^2$ per x_1, x_2
- $10 \cdot 10 \cdot 10 = 10^3$ per x_1, x_2, x_3
- ...
- $10 \cdot \ldots \cdot 10 = 10^d$ per $x_1, \ldots, x_d$

Questo è uno dei motivi che porta ad estrarre caratteristiche da un input

$$(x_1, \ldots, x_d) \xrightarrow[\text{delle caratteristiche}]{\text{Estrazione}} (\tilde{x_1}, \ldots, \tilde{x_r}) \quad t.c. \quad r << d$$

devo assicurarmi che queste caratteristiche (*features*) portino con sé tutte le informazioni necessarie per risolvere il problema e che siano anche il meno ridondanti possibile.

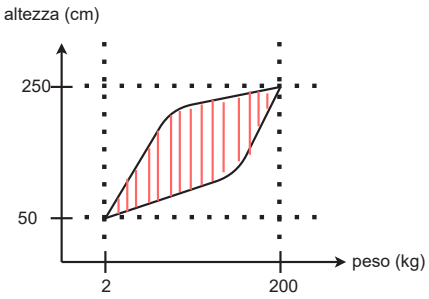
Ridurre l'effetto della maledizione

Spesso le variabili di input non sono totalmente indipendenti ma legate tra loro in qualche misura.

Esempio

Se considero x_1 = peso ed x_2 = altezza con $x_1 \in [2, 200]$ in Kg ed $x_2 \in [50, 250]$ in cm, il dominio possibile non è rappresentato da tutta l'area del rettangolino che abbiamo a disposizione; ad esempio non avremo mai una persona di 50cm che pesa 200kg.

Possibile distribuzione di altezza-peso



Negli approcci classici delle *Shallow Networks*, la fase di progettazione riguardante l'estrazione delle caratteristiche è fondamentale ed è una parte esterna al processo di learning. [8]
Quindi c'è un processo, non banale, di estrazione delle caratteristiche significative prima di poter iniziare il learning.

Invece nel **Deep Learning**, il processo di estrazione delle caratteristiche opportune lo si vuole inserire nella fase di learning. Questo implica la necessità di grandi moli di dati.
In questo modo non risolvo il problema della maledizione della dimensionalità (che è ancora presente), piuttosto spero che sussista una qualche dipendenza tra le features che mi vada a ridurre la dimensionalità.

Data: 21/04/2021

2.9 Vanishing Gradient Problem

Quando aggiungo diversi strati alla rete, rischio di avere tanti minimi locali, e quindi di "appiattire" la funzione di errore, ovvero il *Vanishing Gradient Problem*: la funzione di errore diventa piatta rispetto ai pesi e quindi non si riesce a trovare la soluzione corretta.

$$\frac{\partial E}{\partial w_{ij}^{l}} \simeq 0$$

IMMAGINE FUNZIONE DI ERRORE
Dato che parto da valori casuali dei pesi, probabilmente il processo di learning inizia in una zona "piatta" e possono esserci molti minimi locali, quindi non si raggiunge il minimo globale. Questo era il problema per cui si era restii ad usare più strati in una rete.

Per risolvere questo problema sono stati utilizzati principalmente due approcci. Il primo consiste nel non partire da pesi random ma scelti in maniera più intelligente che dia più chance di arrivare vicino ad un minimo globale.

[8] In qualche modo anche il learning può essere coinvolto per l'estrazione delle caratteristiche

Questo è stato fatto tramite l'uso di *Auto-Encoders*.

Il secondo approccio, che al momento ha più successo, è quello di capire perchè otteniamo il Vanishing Gradient, che è legato all'uso di funzioni di attivazione tipo sigmoide o tangente iperbolica (*tanh*).
Ricordiamo che, nella back propagation:

$$\delta_j = g'_j(a^l_j) \cdot \sum_j w_{ij}\delta^{l+1}_i$$

con

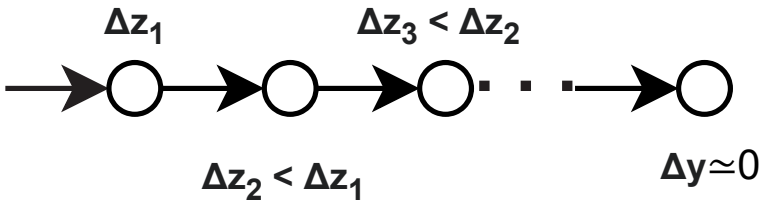
$$a^l_j = \sum_h w_{jh} z^{l-1}_h$$

È altamente probabile che la a^l_j cada in una zona del dominio della sigmoide in cui ho un andamento piatto e quindi con derivata nulla. Quindi $\delta^l_j \simeq 0$, e di conseguenza $\frac{\partial E}{\partial w^l_{jh}} = \delta^l_j \cdot z^{l-1}_h \simeq 0$ con $z^{l-1}_h \in]0, 1[$.

Un altro modo di vedere lo stesso effetto:
Variando il valore di un peso vicino ad uno strato di output

$$z_k = sig(\sum_{kh} z_h)$$

si hanno piccole variazioni dell'output. Variando il valore di un peso lontano dallo strato di output si ha una catena di variazioni sempre più piccole che diventano via via più prossime allo zero.



Andamento del peso

Per risolvere il problema bisogna trovare una nuova funzione di attivazione che soddisfi i seguenti vincoli:

1. Funzioni non polinomiali (e non lineare)
2. Funzione derivabile
3. Una grande parte del dominio deve avere una derivata che è lontana da 0

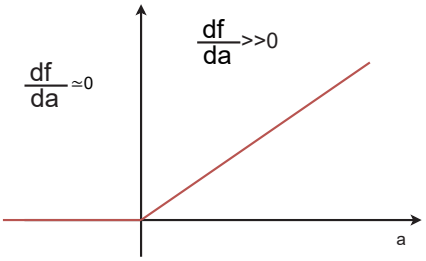
Rectified Linear Unit (ReLU)

$$RELU(a) = \begin{cases} a & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

O, equivalentemente

$$RELU(a) = \max(0, a)$$

Ci sono infiniti punti per cui la derivata è diversa da 0 (è 1). La funzione ReLU



Funzione ReLU

ha permesso la creazione di reti neurali con molti strati ed ha permesso di includere la fase di estrazioni delle caratteristiche nel processo di learning.
Nasce tutta una famiglia di funzioni rettificate

Leaky-ReLU

$$LRELU(a) = \begin{cases} a & \text{if } a > 0 \\ +\beta a & \text{otherwise} \end{cases}$$

e $\beta \in (0, 1)$ può sia essere incluso nel processo di learning che essere considerato un iperparametro.
Se β è appreso, allora ho una funzione di attivazione che è *apprendibile*.

Un altro punto importante è l'organizzazione delle connessioni che possono essere full connected (quindi senza fare nessuna ipotesi) o considerare delle alternative per estrarre delle caratteristiche, ad esempio le reti convoluzionali.

Data: 05/05/2021

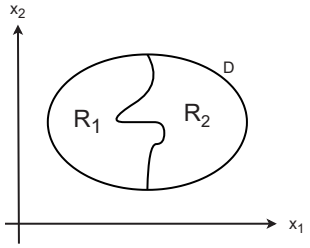
2.10 Classificatori

Dato un problema di classificazione, una rete ed un input $\underline{x}$, si può vedere l'output della rete come la probabilità che $\underline{x}$ sia di una certa classe k .

- Passo 1: $\forall k \quad \mathbb{P}(C_k|\underline{x})$
- Passo 2: regola di decisione $h : \mathbb{P}(C_h|\underline{x}) > \mathbb{P}(C_k|\underline{x}) \quad \forall k \neq h$

Gli elementi che si vogliono classificare appartengono ad un certo $D \subset \mathbb{R}^d$. Dato un problema di classificazione a due classi C_1 e C_2, ogni x in ingresso appartiene a C_1 od a C_2, allora si può dividere il dominio (non necessariamente linearmente separabile) in due regioni, R_1 che corrisponde a C_1 ed R_2 che corrisponde a C_2. L'obiettivo è individuare R_1 e R_2 in modo da poter stabilire $\forall \underline{x} \in D$, se $x \in R_1$ (classe C_1) o se $x \in R_2$ (classe C_2). Si introducono delle funzioni discriminanti ψ_1 e ψ_2 tali che:

$$\psi_i(x) = \begin{cases} > 0 & x \in R_i \\ < 0 & Altrimenti \end{cases}$$



Un esempio di dominio diviso in due classi

x è di classe 1 se $\psi_1(x) > \psi_2(x)$, altrimenti è di classe 2.
In generale se ho c classi, si può dire che x è di classe h se $\psi_h(x) > \psi_k(x) \quad \forall k \neq h$.
Quindi ogni problema di classificazione può essere formalizzato in termini di **funzioni discriminanti**.
Ad esempio:

$$\psi_i(x) = \mathbb{P}(C_i|\underline{x})$$

$\mathbb{P}(C_i|\underline{x})$ come funzione discriminante

L'idea è quella di costruire R_1, R_2 in modo tale da minimizzare la probabilità di fare una classificazione errata:

$$\mathbb{P}(err_{MIS}) = \mathbb{P}(x \in R_1, C_2) + \mathbb{P}(x \in R_1, C_1)$$

O, equivalentemente[9]:

$$\mathbb{P}(err_{MIS}) = \mathbb{P}(x \in R_1, C_2)\mathbb{P}(C_2) + \mathbb{P}(x \in R_2, C_1)\mathbb{P}(C_1)$$

[9] $\mathbb{P}(A, B) = \mathbb{P}(A|B)\mathbb{P}(B)$

In questo caso, x è una variabile continua; si ricorda che:

$$sia\, x \in \mathbb{R}: \quad \mathbb{P}(a \leq x \leq b) = \int_a^b p(x)dx$$

Dove $p(x)$ è la distribuzione di probabilità.

Ad esempio, con $p(x)$ gaussiana:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

$$\mathbb{P}(err_{MIS}) = \int_{R1} p(x|C_2)\mathbb{P}(C_2)dx + \int_{R2} p(x|C_1)\mathbb{P}(C_1)dx$$

Con $\mathbb{P}(C_1)$ e $\mathbb{P}(C_2)$ probabilità a priori della classe.

Per minimizzare l'errore bisogna costruire R_1 ed R_2 nel seguente modo:

$$\begin{cases} x \in R_1 & \text{se } p(x|C_2)\mathbb{P}(C_2) < p(x|C_1)\mathbb{P}(C_1) \\ x \in R_2 & \text{se } p(x|C_2)\mathbb{P}(C_2) > p(x|C_1)\mathbb{P}(C_1) \end{cases}$$

Quindi con

$$\psi_1(x) = p(x|C_1)\mathbb{P}(C_1)$$
$$\psi_2(x) = p(x|C_2)\mathbb{P}(C_2)$$

Si può definire la seguente funzione discriminante

$$\begin{cases} C_1 & \text{se } \psi_1(x) > \psi_2(x) \text{ cioé } x \in R_1 \\ C_2 & \text{altrimenti, cioé } x \in R_2 \end{cases} \tag{2.10}$$

Conoscendo $p(x|C_1)$, $p(x|C_2)$, $\mathbb{P}(C_1)$, $\mathbb{P}(C_2)$, questa scelta di $\psi_1(x)$ $\psi_2(x)$ mi permette di minimizzare l'errore di Mis-classificazione.

I classificatori basati su tali funzioni discriminanti sono detti **generativi** perché, avendo

$$p(x) = (p(x|C_1)\mathbb{P}(C_1) + (p(x|C_2)\mathbb{P}(C_2)$$

si possono generare i dati a partire da $p(x|C_1)$, $p(x|C_2)$, $\mathbb{P}(C_1)$, $\mathbb{P}(C_2)$.

Considerando il **Teorema di Bayes**

$$\mathbb{P}(A,B) = \mathbb{P}(A|B)\mathbb{P}(B) = \mathbb{P}(B|A)\mathbb{P}(A) \implies \mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}$$

Date

$$\mathbb{P}(C_1|x) = \frac{\mathbb{P}(x|C_1)\mathbb{P}(C_1)}{\mathbb{P}(x)}$$
$$\mathbb{P}(C_2|x) = \frac{\mathbb{P}(x|C_2)\mathbb{P}(C_2)}{\mathbb{P}(x)}$$

allora

$$\mathbb{P}(C_2|x) > \mathbb{P}(C_1|x) \implies p(x|C_2)\mathbb{P}(C_2) > p(x|C_1)\mathbb{P}(C_1)$$

Si possono quindi usare come funzioni discriminanti:

$$\psi_1(x) = \mathbb{P}(C_1|x) \quad \psi_2(x) = \mathbb{P}(C_2|x)$$

e, usando la regola di decisione (2.10) minimizziamo l'errore di MISclassificazione
Si può avere come obiettivo quello di conoscere $\mathbb{P}(C_1|x) \quad \mathbb{P}(C_2|x)$.
Questo approccio è detto **discriminativo** e non può generare i dati.

Data: 10/05/2021

2.11 Scelta della funzione di errore

Sia $\{(\underline{x}^n, \underline{t}^n)\}_{n=1}^N$ il **dataset** a disposizione.
L'idea è: se al posto del fenomeno che ha generato il dataset si mettesse il modello di machine learning creato, si vorrebbe massimizzare la probabilità di di avere proprio quel dataset, $\mathbb{P}(DS)$.
Ovvero, il modello si comporta come il fenomeno stesso, è una sua buona approssimazione.

$$\mathbb{P}(DS) \equiv \mathcal{L} = \prod_{n=1}^{N} p(\underline{x}^n, \underline{t}^n)$$

Supponendo che ciascuna coppia è indipendente dall'altra. [10]
Questa quantità è detta *verosimiglianza* (o Likelihood) e può essere riscritta come

$$\mathcal{L} = \prod_{n=1}^{N} p(\underline{t}^n, |\underline{x}^n) p(\underline{x}^n)$$

Massimizzare questa probabilità significa minimizzare la seguente funzione di errore

$$E = -\ln \mathcal{L} = -\sum_{n=1}^{N} \ln p(\underline{t}^n, |\underline{x}^n) p(\underline{x}^n) = -\sum_{n=1}^{N} \ln p(\underline{t}^n, |\underline{x}^n) - \underbrace{\sum_{n=1}^{N} \ln p(\underline{x}^n)}_{\text{Non dipende dal modello}}$$

È possibile ridefinire la funzione di errore da minimizzare come

$$E = -\sum_{n=1}^{N} E^{(n)}$$

$$E^{(n)} - \ln p(\underline{t}^n, |\underline{x}^n)$$

Supponendo che i valori di $\underline{t^n} = (t_1^n, \ldots, t_c^n)$ siano indipendenti gli uni dagli altri, allora:

$$p(\underline{t}^n, |\underline{x}^n) = \prod_{k=1}^{C} p(t_k^n, |\underline{x}^n)$$

[10] $\mathbb{P}(A, B) = \mathbb{P}(A) \cdot \mathbb{P}(B)$ se A e B sono indipendenti

$$E^{(n)} = -\ln \prod_{k=1}^{C} p(t_k^n, |\underline{x}^n) = -\prod_{k=1}^{C} \ln p(t_k^n, |\underline{x}^n)$$

Questo vale sia per i problemi di classificazione che di regressione.

Si supponga che t_k^n sia una variabile continua:

$$t_k^n = f(\underline{x}^n) + \epsilon_k$$

ovvero è uguale ad una certa funzione $f(\underline{x}^n)$ più un errore ϵ_k. Ovviamente non si conosce f, ed in più i target presenti nel dataset sono influenzati da un errore dovuto alla misurazione.

Esempio Prevedere il livello di glicemia di una persona tra 30 minuti, partendo dal livello di glicemia attuale, l'età, il peso, etc..
Il calcolo del valore effettivo della glicemia (il target), misurato dopo 30 minuti, è influenzato dall'errore dovuto allo strumento di misurazione o all'agente umano.

L'ipotesi più plausibile, nel caso di una variabile continua, è quella di **errore gaussiano**.

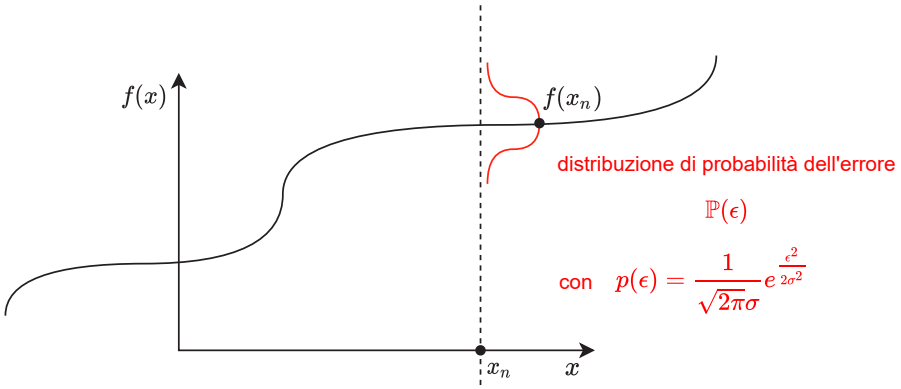


Figure 2.3: Errore gaussiano

Ovvero meno mi allontano dal valore medio, più ho probabilità di avere un valore vicino a quello vero e più mi allontano dal valore medio meno ho probabilità di avere un valore vicino a quello vero. Questo significa che $p(t_k^n, |\underline{x}^n))$ è una distribuzione di probabilità gaussiana centrata intorno ad $f(x^n)$, allora:

$$p(t_k^n|\underline{x}^n) = \frac{1}{\sqrt{2\pi}\sigma_k} e^{\frac{-((t_k^n - f(\underline{x}^n))^2}{2\sigma_k^2}}$$

Dove al posto di $f(\underline{x}^n)$ voglio utilizzare la risposta del modello, ovvero $y_k(\underline{x}^n; \Theta)$, allora:

$$p(t_k^n|\underline{x}^n) = \frac{1}{\sqrt{2\pi}\sigma_k} e^{\frac{-(y_k(\underline{x}^n;\Theta) - t_k^n)^2}{2\sigma_k^2}}$$

Ed ancora, posso scrivere:

$$E^{(n)} = -\sum_k \ln \frac{1}{\sqrt{2\pi}\sigma_k} e^{\frac{-(y_k(\underline{x}^n;\Theta)-t_k^n)^2}{2\sigma_k^2}} = -\underbrace{\sum_k \ln \frac{1}{\sqrt{2\pi}\sigma_k}}_{\text{Non dipende dal modello}} - \sum_k \frac{-(y_k(\underline{x}^n;\Theta)-t_k^n)^2}{2\sigma_k^2}$$

Allora posso scrivere:

$$E^{(n)} = \sum_{k=1}^{C} \frac{(y_k(\underline{x}^n;\Theta)-t_k^n)^2}{2\sigma_k^2}$$

Ponendo $\sigma_{min} = min\{\sigma_k\}$ e $y_k^n \equiv y_k(\underline{x}^n;\Theta)$, ottengo:

$$\sum_{k=1}^{C} \frac{(y_k^n - t_k^n)^2}{2\sigma_k^2} < \frac{1}{2\sigma_{min}} \sum_{k=1}^{C} (y_k^n - t_k^n)^2$$

$$E^{(n)} = \frac{1}{2} \sum_{k=1}^{C} (y_k^n - t_k^n)^2 \tag{2.11}$$

Minimizzare 2.11 significa trovare i parametri del modello che massimizzano la probabilità di avere il dataset a disposizione quando al posto del fenomeno, $f(x)$, utilizzo il modello: sto massimizzando la verosimiglianza $\mathcal{L}$

Caso discreto (classificazione)

Nel caso di t^n discreto, quindi nei problemi di classificazione, si avrà:

$$E^{(n)} = -\sum_{k=1}^{C} \ln p(y_k^n - t_k^n)$$

E non è più possibile assumere un errore Gaussiano perché ogni del modello può essere soltanto giusta o sbagliata.
Si consideri un problema a due classi, si avrà che $t_k^n = t^n$ con $t^n \in \{0, 1\}$ e si supponga che quando $t^n = 1$ sia q la probabilità *vera* di appartenere alla classe e quando $t^n = 0$ sia $(1 - q)$ la probabilità *vera* di appartenere all'altra classe.
In questo caso si può scrivere:

$$p(t|\underline{x}^n) = q^t(1-q)^{(1-t)}$$

con $t = 1 \implies q$ e $t = 0 \implies 1 - q$.
Sostituendo a q la risposta del modello

$$p(t|\underline{x}^n) = y(\underline{x}^n;\Theta))^t(1 - \underline{x}^n;\Theta))^{1-t}$$

ponendo $y^n \equiv y(\underline{x}^n;\Theta)$, allora:

$$E^{(n)} = -\ln (y^n)^{t^n}(1-y^n)^{(1-t^n)} = -\underbrace{[t^n \ln y^n + (1+t^n)\ln(1-y^n)]}_{\text{Cross-entropy}}$$

Ciò vale per due classi con una sola variabile di uscita (un unico neurone di output).

Nel caso di più classi:

$$E^{(n)} = -\sum_{k=1}^{C} \ln p(t_k^n|\underline{x}^n)$$

$$\text{con } p(t_k^n|\underline{x}^n) = (y_k^n)^{t_k^n}$$

$$E^{(n)} = -\sum_{k=1}^{C} \ln (y_k^n)^{t_k^n} = -\sum_{k=1}^{C} t_k^n \ln y_k^n$$

Utilizzabile per più classi (≥ 2), oppure per due classi quando scelgo di avere due variabili di uscita (2 neuroni di output).
Anche in questo caso, minimizzare la funzione di errore significa massimizzare la verosimiglianza L.

Osservazioni

Avendo, nel caso di due classi:

$$E^{(n)} = -[t^n \ln \underbrace{y^n}_{>0} + (1 - t^n) \ln \underbrace{(1 - y^n)}_{>0}]$$

allora $0 < y^n < 1$ e quindi è necessario utilizzare una funzione di attivazione che assicuri tale condizione.
Nel caso a c classi, con $c \geq 2$:

$$E^{(n)} = -\sum_{k} t_k^n \ln y_k^n$$

E' possibile fare operazioni di post-processing, ad esempio il **Soft-Max**:

Definition 2.11.1 (Soft-Max)**.**

$$z_k^n = \frac{e^{y_k^n}}{\sum_{k=1}^{C} e^{y_k^n}} \text{ con } 0 < z_k^n < 1$$

In letteratura, spesso, l'applicazione di questa funzione nell'ultimo livello viene definita come applicazione di una funzione di attivazione, in realtà è una funzione di attivazione non locale, cioé per sapere l'uscita di un nodo bisogna sapere l'uscita degli altri nodi. Per questo motivo può essere più "pulito" vedere questa funzione come una operazione di post-processing.
La funzione di errore quindi assume la seguente forma:

$$E^{(n)} = -\sum_{k} t_k^n \ln \frac{e^{y_k^n}}{\sum_{k=1}^{C} e^{y_k^n}}$$

CHAPTER 3

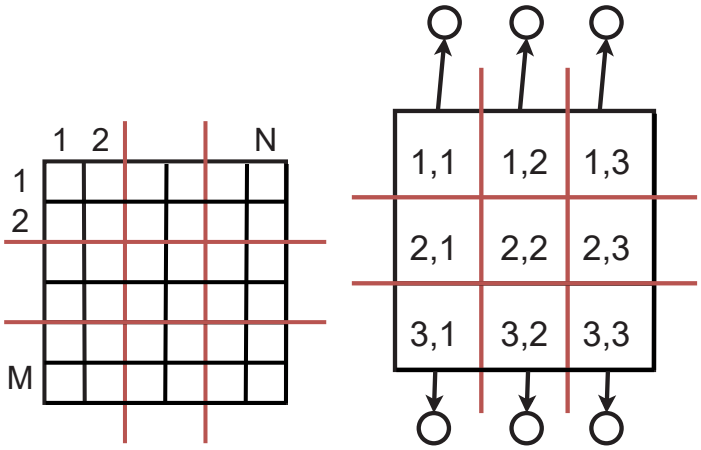
Reti convoluzionali

Data: 26/04/2021

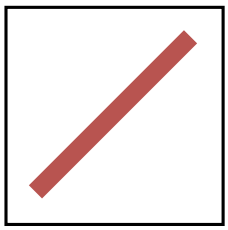
3.1 Convoluzione e neuroni

Le reti convoluzionali nascono per risolvere problemi di classificazione delle immagini.
Possiamo considerare un'immagine(il nostro dominio) come una matrice di M righe ed N colonne $\underline{x} \in R^{M \times N}$, l'input è quindi $x_{m,n} \in (0, 1)$ (dopo qualche normalizzazione dell'immagine, ad esempio portata in scala di grigi).
In una situazione Full Connected, ad ogni neurone arriverebbe tutta l'immagine (cella per cella).
L'idea è quella di processare, invece, l'immagine per regioni locali e distinte

Divisione in regioni



Nello strato interno avrò un neurone adibito a ciascuna regione quindi per ogni regione avrò un certo numero di connessioni che mappano i pixel di quella regione con il neurone associato alla regione.
Vorrei che ogni neurone fosse in grado di individuare una particolare caratteristica di quella regione.



Segmento a 45 gradi da identificare

Ad esempio un determinato neurone deve individuare una sbarretta obliqua a 45 gradi, se è presente nell'immagine allora quel neurone restituisce un output alto, altrimenti un output basso. L'ideale sarebbe che tutti i neuroni fossero capaci di estrarre la stessa caratteristica, cioé che tutti i neuroni siano selettivi per quella specifica caratteristica che stiamo cercando; in questo almeno uno dei neuroni avrà output alto per indicare che la caratteristica è presente nell'immagine. Formalmente, supponiamo che ogni regione abbia dimensione $S \times S$.

Per la prima regione l'output sarà il seguente:

$$\begin{aligned} z_{11} = g(&w_{11}x_{1,1} + w_{12}x_{1,2} + \dots w_{1S}x_{1,s} + \\ &w_{21}x_{2,1} + w_{22}x_{2,2} + \dots w_{1S}x_{2,s} + \\ &\dots \\ &w_{s1}x_{s,1} + w_{s2}x_{s,2} + \dots w_{sS}x_{s,s} + b_{ss}) \end{aligned}$$

Con pesi

$$W = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1s} \\ w_{21} & w_{22} & \dots & w_{2s} \\ & \vdots & & \\ w_{s1} & w_{s2} & \dots & w_{ss} \end{pmatrix}$$

Per la seconda regione:

$$\begin{aligned} z_{12} = g(&w_{11} \quad x_{1,s+1} + \dots + w_{1S} \quad x_{1,s+s} + \\ &w_{21} \quad x_{2,1} + w_{22} \quad x_{2,2} + \dots w_{1S} \quad x_{2,s} + \\ &\dots \\ &w_{s1} \quad x_{s,s+1} + \dots w_{ss} \quad x_{s,s+s} + b_{12}) \end{aligned}$$

La matrice dei pesi è invariata (anche il bias).
Quindi, in generale:

$$z_{ij} = g \begin{pmatrix} w_{11} & x_{(i-1)s+1,(j-1)s+1} + \dots + w_{1s} & x_{(i-1)s+s,(j-1)s+s} \\ & \vdots & \\ w_{s1} & x_{(i-1)s+1,(j-1)s+1} + \dots + w_{ss} & x_{(i-1)s+s,(j-1)s+s} + b \end{pmatrix}$$

$$z_{ij} = g(\sum_{h=1}^{s}\sum_{k=1}^{s} w_{hk} \quad x_{(i-1)s+h,(j-1)s+k} + b)$$

con w_{hk} che resta invariato per ciascuna regione, come se la matrice dei pesi si spostasse su ogni regione.
Posso supporre che dimensione e spostamento della matrice siano diverse: dimensione L, spostamento s con $L \neq s$ ed $s = 1, 2, ..., k$

$$z_{ij} = g(\sum_{h=1}^{L}\sum_{k=1}^{L} w_{hk} \quad x_{(i-1)s+h,(j-1)s+k} + b) \tag{3.1}$$

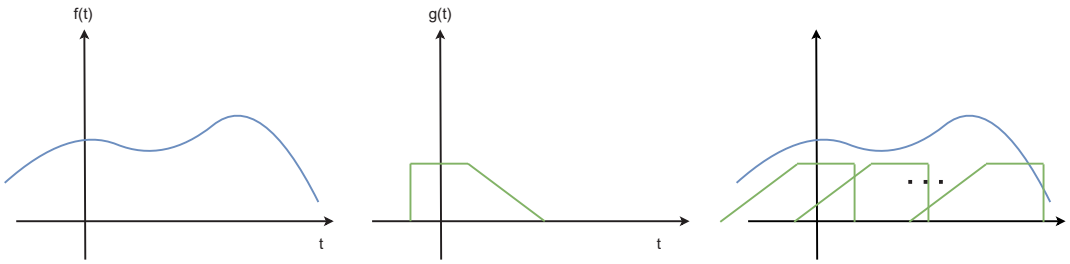


Figure 3.2: Esempio di convoluzione

Per $s = 1$ si avrà una finestra di dimensione $L \times L$ che si sposta in avanti di un pixel alla volta.
w_{hk} è detto filtro (o kernel). Quindi si ha un segnale dell'immagine ed un segnale del filtro, il calcolo di z_{ij} (3.1) è qualcosa di simile alla convoluzione tra X e W.

Convoluzione Dati due segnali $f(t)$ ed $s(t)$, allora si chiama convoluzione di f con g:

$$f \circ g(t) = \int_{-\infty}^{+\infty} f(\tau)s(t-\tau)d\tau$$

E corrisponde ad:

- si inverte g
- si "shifta" g invertito e lo si moltiplica per f

Le uscite (output) dei neuroni z_{ij} rappresentano una convoluzione tra X (l'immagine in ingresso) e W (il filtro) più l'applicazione, su tale risultato, della funzione di attivazione (g). Per questo motivo queste reti prendono il nome di reti neurali convoluzionali.
Data l'immagine X, ho in output un certo numero di neuroni (*feature map*): con un'immagine $M \times N$ ho $m_1 \times n_1$ neuroni con $m_1, n_1 \leq M, N$ (dipende dalla scelta di s). Posso avere quindi diverse feature map per riconoscere diverse caratteristiche dell'immagine in ingresso. In una rete convoluzionale, i livelli non sono full connected ma sono organizzati in filtri: un livello nascosto può contenere diverse feature map. INSERIRE SCHEMA RETE CONVOLUZIONALE CON DIVERSE FEATURE MAP

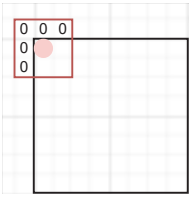


Figure 3.4: Padding di una sezione dell'immagine.

Data: 28/04/2021

Stride e Padding

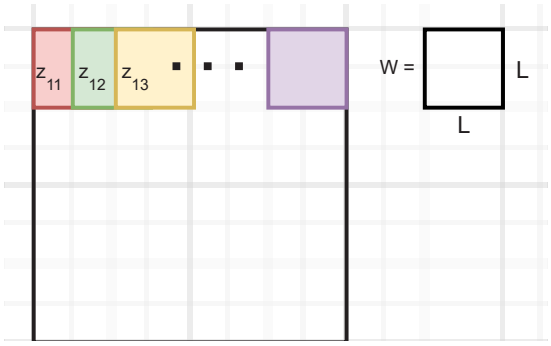


Figure 3.3: Spostamento del filtro sull'immagine con $s = 1$

$s = 1$ corrisponde a fare una convoluzione ed avrò un neurone corrispondente ad ogni shift del filtro creando quindi una Feature Map dove tutti i neuroni hanno gli stessi pesi W e dove ciascun neurone riceve connessione da pixel diversi (con sovrapposizione).

È possibile creare diverse feature maps considerando più filtri $W^1, W^2, \ldots, W^F$ (F filtri diversi con la stessa dimensione L, ad esempio 7×7) che saranno considerati come un unico strato hidden di $F \cdot 7 \cdot 7$ neuroni, infatti le connessioni vanno direttamente da x verso le feature maps e non ci sono connessioni tra i neuroni delle feature maps.
Per creare un secondo strato di feature maps, e quindi iniziare a strutturare la rete come rete *deep*, vanno fatte alcune considerazioni:

- Lo spostamento s del filtro è detto **stride**
- Quando il filtro si sposta sull'immagine è come se si stesse centrando su un pixel particolare. Per questo motivo alcuni pixel sono coinvolti nell'elaborazione ed altri invece non sono coinvolti: i pixel sono trattati in maniera diversa. Per risolvere questo "problema" si introduce una zona esterna all'immagine, cioé si aggiungo righe e colonne di pixel con valore 0 (**padding**) che servono solo per spostare i filtri in maniera tale da centrare i pixel che restano.
  In questo modo le feature maps hanno stessa dimensione dell'immagine

Un'immagine a colori è data da 3 matrici di dimensioni $M \times N$ ed $X \in R^{M \times N \times 3}$ ovvero è un volume. Allo stesso modo possiamo creare un filtro tridimensionale

$$z_{ij} = g(\sum_{l=1}^{L} \sum_{h=1}^{L} \sum_{k=1}^{L} w_{ijl} \quad x_{(i-1)s+h,(j-1)s+k,l} + b)$$

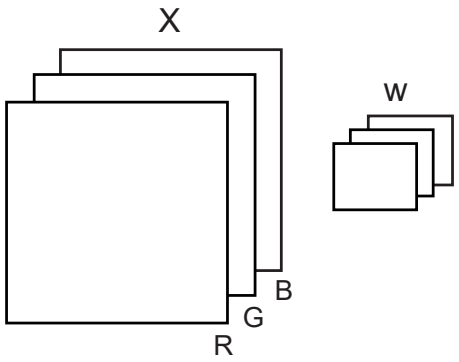


Figure 3.5: Immagine RGB e relativo filtro

Posso estendere il ragionamento per qualunque profondità p, X sarà un volume $M \times N \times P$ ed ugualmente il filtro sarà di dimensione $L \times L \times P$. In generale, altezza e larghezza del filtro possono anche essere differenti.

Si ottengono F feature maps di dimensione $M \times N$ dove F è il numero di filtri.
Si possono riposizionare le F feature maps in modo da ottenere un volume IMMAGINE che rappresenta sempre un unico hidden layer con connessioni provenienti da X ed opportunamente organizzate.

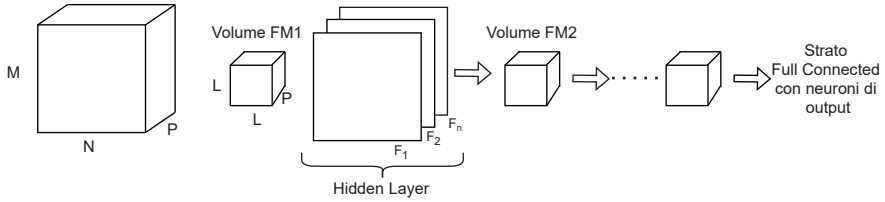
Su questo volume si può ripetere lo stesso ragionamento, applicando F_2 filtri di dimensione L_2 (Eventualmente con $L_2 \neq L$ ed $F_2 \neq F$) ottenendo un nuovo volume di feature map. Il nuovo volume rappresenta il secondo hidden layer e così si possono estrarre caratteristiche di più alto livello rispetto al primo. Posso creare diversi livelli hidden con le feature maps che rappresenta tutto il processo di estrazione di caratteristiche, mentre nell'ultimo livello full connected avviene la classificazione (o regressione). Quindi l'ultimo livello e il penultimo sono full connected.

In sintesi:

- Le feature maps estraggono caratteristiche sempre più di alto livello man mano che si allontanano dall'input.
- Tutti i pesi di tutti i filtri sono da apprendere, insieme ai pesi dell'ultimo livello.

Data: 03/05/2021

3.2 Reti Convoluzionali



Dato in input $x \in R^{M \times N \times P}$ con caso limite $P = 1$, si applicano diversi filtri di dimensione $L \times L \times P$ creando un nuovo strato hidden composto da più filtri andando a creare un nuovo volume (come x) con tante immagini quanti sono i filtri, sul quale si può ripetere nuovamente lo stesso procedimento applicando nuovi filtri e generando nuovi volumi fino ad arrivare ad un ultimo strato full connected che contiene i neuroni di output.

Possiamo intervallare ciascuna feature map con un'ulteriore elaborazione in modo tale da:

- ridurre le dimensioni di altezza e larghezza (lasciando invariata la profondità) per ridurre la complessità della rete
- aumentare la tolleranza agli spostamenti, ovvero alla posizione della caratteristica rispetto all'immagine

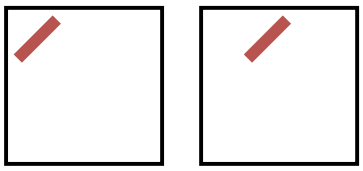


Figure 3.6: Diverse posizioni di una feature necessitano un certo livello di tolleranza

Esistono degli operatori che vanno a ridurre altezza e larghezza di un'immagine. È possibile suddividere una feature map in tante regioni e per ognuna di queste effettuare un'operazione di massimo[1] in quella regione, riducendo la feature map in termini di larghezza ed altezza ottenendo una nuova feature map $M' \times N' \times P$ (con $M', N' < M, N$) per ogni strato convolutivo.

Quindi, riassumendo, ogni strato convolutivo può essere caratterizzato dalle seguenti 3 operazioni:

[1] Il massimo del valore di output dei neuroni associati a quella regione della feature map, cioé il valore più "significativo" di quella zona

1. Convoluzione
2. Trasformazione non lineare (Funzione di attivazione, ad es. ReLU)
3. Riduzione di altezza e larghezza (ad es. tramite il massimo[2])

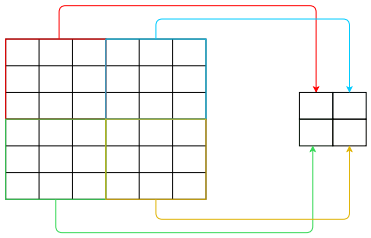


Figure 3.7: Riduzione della dimensione di un'immagine tramite operatore MAX

Operazione di massimo come neurone

L'operazione di massimo fatta su una feature map può essere vista come l'elaborazione fatta da un neurone. Considerando una finestra 3×3

$$a = \sum_{i=1}^{9} w_i x_i$$

$$y = g(a)$$

IMMAGINE NEURONI Con il comportamento di un neurone classico non è possibile calcolare il massimo, quindi, per poter considerare l'operazione di massimo come elaborazione di un neurone e per mantenere la struttura di rete feed forward, bisogna ridefinirne il comportamento.
Con opportune accortenze poi, è possibile applicare la back propagation.

Siano:

$$a = f_I((x_1, x_2, \ldots, x_d))$$

$$y = f_A(a + b)$$

con f_I funzione di input che, nel caso classico, corrisponde alla somma pesata, e f_A funzione di attivazione.

Operatore Massimo Scegliendo come funzione di input ($f_I \equiv MAX$) il massimo, $b = 0$ e come funzione di attivazione la funzione identità ($f_A \equiv id$), si ottiene che il neurone computa il massimo.

[2]Ma esistono anche altri operatori, ad es. la media

Operatore Media Se f_I è la somma pesata e si scelgono i pesi tutti pari ad $\frac{1}{d}$ dove d è il numero di ingressi al neurone, con $b = 0$ ed $f_A \equiv id$

Per aggiornare pesi e bias usi usa lo stesso approccio usato precedentemente, cioé si calcola:

$$\frac{\partial E^{(n)}}{\partial w_{ij}}$$

Per applicare la back-propagation bisogna usare delle accortenze rispetto all'approccio precedente, perchè ogni filtro è visto come una mappa di neuroni tutti con gli stessi pesi.
$w_1, w_2, \ldots, w_d$ è il filtro che proviene dalla regione R_1 dell'immagine per il primo, dalla regione R_2 per il secondo e così via fino al s-esimo, con filtro proveniente dalla regione R_s e per ogni regione si avranno input diversi ma stessi pesi. Usando quindi la definizione classica di back-propagation:

$$\frac{\partial E^{(n)}}{\partial w_i} = \delta_k \cdot x_i^k$$

con w_i peso del neurone k-esimo.
Quindi l'i-esimo peso può cambiare in maniera diversa per ciascun neurone k, ottenendo così dei valori diversi. Non si avrà più quindi un unico filtro che si sposta sulle regioni, ma filtri diversi, non facendo più convoluzione.[3]
In maniera intuitiva si può pensare di fare la somma:

- Chiamato w_i l'i-esimo peso di ciasciun neurone k, si calcola:

$$\frac{\partial E^{(n)}}{\partial w_i} = \sum_{k=1}^{s} \delta_k \cdot x_i^k$$

  E si utilizza tale derivata per aggiornare w_i in **tutti** i neuroni k

Più formalmente, nelle reti convoluzionali, w_{ij} è lo stesso peso per tutti i nodi i ($w_{1j} = w_{2j} = \cdots = w_{sj}$) quindi in questo caso w_{ij} è coinvolto nel calcolo di tutti gli input di tutti i neuroni dello strato:

$$\frac{\partial E^{(n)}}{\partial w_i} = \sum_{k=1}^{s} \frac{\partial E^{(n)}}{\partial a_k} \cdot \frac{\partial a_k}{\partial w_{ij}} = \sum_{k=1}^{s} \delta_k \cdot z_j^k$$

Possiamo quindi applicare la back-propagation. Per lo strato che realizza l'operazione di massimo (o equivalenti) non cambiamo i pesi ma propaghiamo all'indietro i delta. Ricordando che FORMULA

$$\delta_h = g'(a_h) \sum_k w_{kh} \delta_k$$

Nel caso del massimo, ripetendo lo stesso ragionamento :

$$a_k = MAX(y_1, y_2, \ldots, y_p)$$

Se y_h è il massimo allora $\frac{\partial a_k}{\partial y_h} = 1$, altrimenti $\frac{\partial a_k}{\partial y_h} = 0$

$$a_{v_{v_{v_{v_{v_{v_{v}}}}}}}$$

[3] Tuttavia, questo approccio non è "sbagliato": ci sono diverse proposte in letteratura in cui si utilizzano diversi filtri per regioni diverse.

Appendices

APPENDIX A

Funzioni di più variabili

A.1 Derivate parziali

Definition A.1.1 (Derivata parziale di una funzione di più variabili)**.** Sia

$$f : \mathbb{R}^p \to \mathbb{R}$$

e

$$p_o \in \mathbb{R}^p$$

Si definisce *derivata parziale* di f nel punto p_0 rispetto alla variabile x_i il valore:

$$\frac{\partial f(p_0)}{\partial x_i} = \lim_{t \to 0} \frac{f(p_0 + te_i) - f(p_0)}{t}$$

con e_i vettore canonico, cioé un vettore con tutti 0 tranne in posizione i, dove c'è 1.

Nota: Nei casi standard, la derivata parziale si calcola con le usuali regole di derivazione per le funzioni di una variabile, considerando le variabili diverse da x_i costanti. La definizione di derivata parziale può essere utile quando non è possibile usare la regola empirica, cioé per le funzioni definite "a pezzi", come la seguente:

$$f(x, y) = \begin{cases} \frac{xy}{x^2+y^2} & \text{if } (x, y) \neq (0, 0) \\ 0 & \text{if } (x, y) = (0, 0) \end{cases}$$

In questo caso (e solo nel punto $(0, 0)$) va usata la definizione con il limite.

A.2 Gradiente di una funzione di più variabili

Definition A.2.1 (Gradiente di una funzione di più variabili)**.** Sia

$$f : \mathbb{R}^p \to \mathbb{R}$$

Si chiama *gradiente di* f in $p_0 \in \mathbb{R}^p$

$$\nabla f(p_0) = (\frac{\partial f(p_0)}{\partial \theta_1}, \frac{\partial f(p_0)}{\partial \theta_2}, \ldots, \frac{\partial f(p_0)}{\partial \theta_p})$$

Quindi si tratta di un vettore che contiene in posizione i-esima la derivata parziale della funzione rispetto alla variabile i-esima nel punto considerato.