Reti Neurali

Appunti delle lezioni di
Paolo Alfano

Note Legali

Liberatoria, aggiornamenti, segnalazione errori:

Quest'opera viene pubblicata in formato elettronico senza alcuna garanzia di correttezza del suo contenuto. Il documento, nella sua interezza, è opera di Paolo Didier Alfano
e viene mantenuto e aggiornato dallo stesso, a cui possono essere inviate eventuali segnalazioni all'indirizzo *paul15193@hotmail.it*
Ultimo aggiornamento: 15 dicembre 2017

Indice

28 / 09 /2017

1 Introduzione

Durante il corso vedremo alcuni argomenti inerenti i problemi di learning *supervised* e *unsupervised*. Nello specifico esistono dei modelli di rete neurale legati a questi due modelli:

- Supervised: abbiamo i percettroni, le adaline, le reti feed-forward e le funzioni radial basis
- Unsupervised: reti di hopfield e self organizing maps

Ad ogni modo una rete neurale è un dispositivo di calcolo ispirato dal modo in cui la mente opera sui problemi di apprendimento. Una rete neurale è un processore distribuito e massivamente parallelo che raccoglie nei suoi *pesi* la conoscenza legata all'esperienza, rendendola disponibile per i futuri compiti da svolgere. L'addestramento di una rete neurale consiste proprio nel trovare il valore ottimale dei pesi.

I vantaggi di utilizzare una rete neurale sono i seguenti:

- Non linearità: il neurone è un dispositivo non lineare, ovvero effettua delle operazioni che introducono della non linearità tra i valori in ingresso e i valori in uscita.
- Corrispondenza: viene trovata una corrispondenza tra input e output grazie alla struttura della rete che fa fluire l'informazione con un procedimento di propagazione in avanti.
- Tolleranza ai fault: se una parte della rete smette di funzionare l'intera rete può continuare a funzionare adattandosi.
- Allenamento incrementale: è sempre possibile continuare ad allenare la rete se abbiamo nuovi dati a disposizione

Una rete neurale è specificata da

- Un'architettura: formata da neuroni e link. Ad ogni link viene associato un peso.
- Un modello di neurone: che stabilisce come processare l'informazione.
- Un algoritmo di apprendimento: che specifica come venga allenata la rete, dunque come vengono modificati i pesi.

Visto che solitamente il modello di neurone utilizzato è molto semplice, il potere computazionale di una rete neurale deriva dal grande numero di neuroni in essa presente e il suo obiettivo è di *generalizzare bene*, ovvero che si comporti bene su nuove istanze di un certo problema.
Inoltre possiamo suddividere le reti neurali in tre macro-categorie:

- Feed-forward a strato singolo: mostrate in Figura 1(a) , vengono dette a strato singolo perché, ad esclusione dello strato di input, abbiamo un solo strato di neuroni, ovvero lo strato di output. Vengono dette feed-forward perché l'informazione fluisce solo in una direzione.

- Feed-forward multi strato: mostrata inFigura 1(b) ,anche in questo caso l'informazione all'interno della rete fluisce in una sola direzione ma in questo caso abbiamo diversi strati che compongono la rete. Nel caso in cui la rete sia costituiti da moltissimi strati, viene detta rete neurale profonda. È stato alla base di una grande rivoluzione negli anni '80 grazie all'introduzione dei livelli intermedi nascosti. L'utilizzo di strati intermedi permise di risolvere un importante problema, ovvero quello di rappresentare problemi non linearmente separabili[1]. Il numero di livelli da utilizzare varia in basa al problema da analizzare, non esiste un numero "standard" di neuroni o di livelli da impiegare. In particolare la Figura 1(b) rappresenta una rete completamente connessa: ogni neurone riceve un input da tutti i neuroni del livello precedente e produce un output per tutti i neuroni del livello successivo.

- Ricorrente: mostrata in Figura 1(c). In questo tipo di rete, parte dell'informazione processata viene "riportata" ai neuroni degli strati precedenti per essere processata nuovamente

Notiamo quindi che esiste una dinamica di tipo intrinseco alle reti neurali: i primi due tipi ad un certo tempo forniscono un output solo se gli è stato fornito un input. Invece la rete ricorrente, a causa dei loop può continuare a far fluire l'informazione nel tempo, e quindi potrei riceve un output senza aver dato un input.

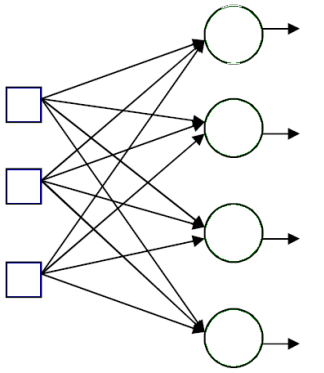
Avendo visto come sia strutturata una rete neurale, andiamo a vedere come è strutturato un singolo neurone. Graficamente possiamo rappresentarlo come in Figura 2.

Il neurone è formato da un insieme di link pesati che descrivono l'input pesato per il neurone. Successivamente viene eseguita la somma di tutti gli input pesati $u_j = \sum_{i=1}^{p} w_{ji} x_i$.
Una funzione di attivazione $\phi(v_j)$ limita il valore del risultato. Solitamente la funzione di attivazione è non lineare. In questo procedimento ha una sua importanza anche il bias, che applica una trasformazione affine[2] alla somma pesata u. Dopo che u ha subito l'effetto del bias b, viene detto *campo indotto*. Il bias ci permette di evitare che, nel caso in cui l'input sia nullo, anche l'output lo sia! Nel complesso, il valore che verrà dato alla funzione di attivazione dopo l'applicazione del bias è:

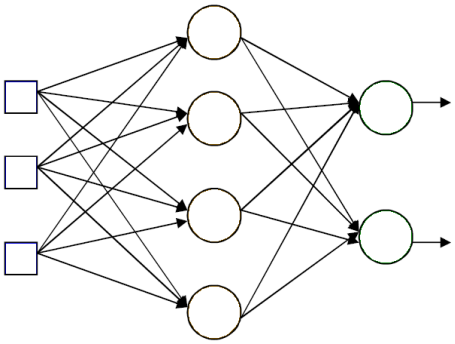
$$v_j = u_j - b_j$$

[1] Si veda il problema di rappresentazione dello XOR.

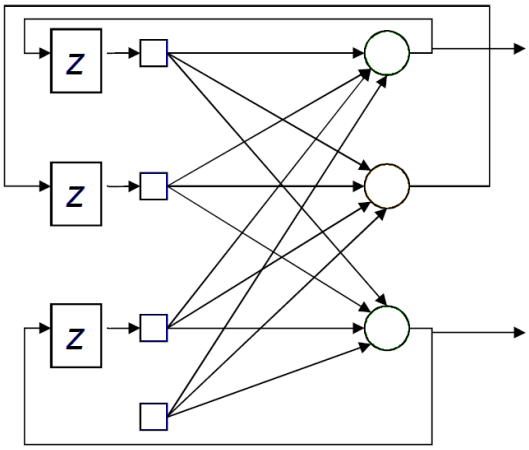
[2] Ricordiamo che una trasformazione affine è la composizione di una trasformazione lineare con una traslazione.



(a) Feed-forward strato singolo



(b) Feed-forward multi strato



(c) Ricorrente

Figura 1: Tipologie di rete neurale

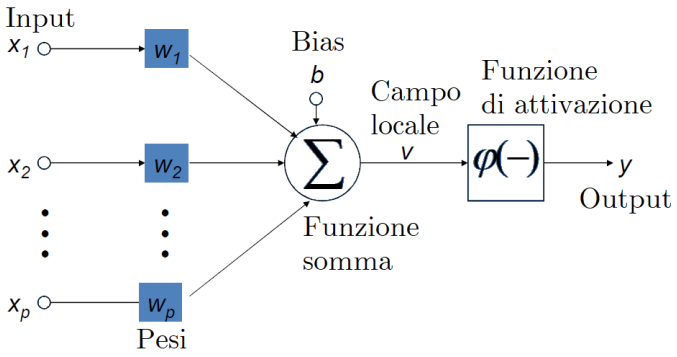


Figura 2: Modello di neurone

Di fatto, notiamo che l'effetto del bias è equivalente ad applicare una somma/-sottrazione al risultato della sommatoria. Dunque può essere interpretato come un ulteriore parametro w_0.

Notiamo che la scelta della funzione di attivazione determina il modello del neurone. Alcuni esempi di funzione sono:

- Step function: $\phi(v) = \begin{cases} a, & \text{se } v < c \\ b, & \text{se } v > c \end{cases}$
- Ramp function: $\phi(v) = \begin{cases} a, & \text{se } v < c \\ b, & \text{se } v > d \\ a + \dfrac{(v-c)(b-a)}{(d-c)}, & \text{altrimenti} \end{cases}$
- Relu function(rettified linear unit): compromesso tra le due precedenti. Vale 0 se l'input è negativo, ma vale v in caso l'input sia positivo. Di fatto coincide con la bisettrice per input positivi.
  $\phi(v) = \begin{cases} 0, & \text{se } v < 0 \\ v, & \text{se } v > 0 \end{cases}$
- Sigmoid function: $\phi(v) = \dfrac{1}{1+e^{-v}}$
- Gaussian function: $\phi(v) = \dfrac{1}{\sigma\sqrt{2\pi}}\, e^{-\frac{1}{2}(\frac{v-\mu}{\sigma})^2}$

Le prime due funzioni sono continue ma non derivabili.
Supponendo di creare una rete che consista di un solo neurone, generalizzare ad una rete a singolo strato è abbastanza semplice visto che l'output di ogni unità

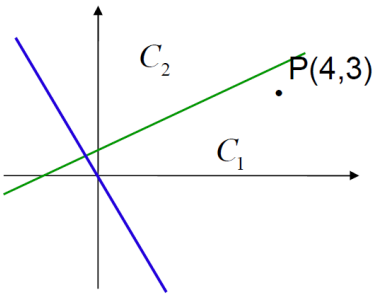


Figura 3: Classificazione dell'esempio 1

non dipende dagli altri neuroni allo stesso livello.

La prima rete neurale progettata da McCulloch e Pitts utilizzava come funzione ϕ la funzione non lineare *segno*:

$$\phi(v) = \begin{cases} +1, & \text{se } v \geq 0 \\ -1, & \text{se } v < 0 \end{cases}$$

2 Allenare una rete

Finora abbiamo nominato neurone e percettrone. Quale è la differenza tra queste due entità? Sono di fatto molto simili ma il percettrone implementa un algoritmo di apprendimento.
Come possiamo, in generale allenare una rete neurale su un certo problema di classificazione? Dobbiamo trovare dei valori per i pesi affinché geometricamente esista un iper-piano che separi i punti delle due classi. Di fatto deve esistere un iper-piano tale per cui:

$$\sum_{i=1}^{m} w_i x_i + w_0 = 0$$

03 / 10 /2017

Vediamolo su un esempio

Esempio 1 : supponiamo di avere la situazione mostrata in Figura 3.

Vorremmo che il punto P venga classificato positivamente, dunque dovremmo trovarlo al di sopra della retta che classifica. Vogliamo dunque determinare i tre coefficienti q, m, n della retta $q + mx - ny = 0$ che in forma esplicita diventa $y = \frac{m}{n}x + q$.

Supponiamo di scegliere inizialmente la tripla $[1, 1, -2]$, rappresentata dalla retta indicata in verde.

Purtroppo il punto non viene classificato correttamente. Dunque dobbiamo trovare una nuova retta. Per farlo consideriamo il punto P a cui

aggiungiamo una coordinata di bias -1. Dunque il punto P viene rappresentato come $[-1, 4, 3]$.
Sommiamo questo punto ai coefficienti della retta. Ovvero la nuova tripla è:

$$[1, 1, -2] + [-1, 4, 3] = [0, 5, 1]$$

La nuova retta rappresentata da $[0, 5, 1]$ è quella mostrata in blu ed effettivamente classifica correttamente il punto interessato.

Possiamo comunque generalizzare e definire l'algoritmo di apprendimento come segue:

```
n=1;
initialize w(n) randomly;
while (there are misclassified training examples)
Select a misclassified augmented example (x(n),d(n))
w(n+1) = w(n) + e d(n)x(n);
n = n+1;
end-while;
```

Notiamo che e (detto anche η) viene detto *parametro di apprendimento* o *iper parametro*(è un numero reale). Inoltre $d(n)$ rappresenta la classe attesa dalla classificazione. Il valore di $d(n)$ ci permette di "correggere il tiro" nel caso di classificazione errata.
Quali garanzie abbiamo rispetto alla convergenza dell'algoritmo appena mostrato? Solitamente nell'ambito delle reti neurali non abbiamo molte garanzie ma in questo caso esiste un teorema che garantisce la convergenza. Vediamolo di seguito

Teorema 2.1 (Convergenza dell'algoritmo di apprendimento)**.** *Supponendo che esistano due classi C_1, C_2 che siano linearmente separabili, allora l'algoritmo di apprendimento del percettrone applicato a $C_1 \cup C_2$ termina con successo dopo un numero finito di iterazioni.*

Dimostrazione. Consideriamo l'insieme $C = C_1 \cup C_2$ dove abbiamo trasformato tutti i punti di C_2 invertendone tutte le componenti. Supponiamo per semplicità che $\mathbf{w(1)} = 0$ e che $\eta = 1$.
Supponendo che esistano k punti $\mathbf{x}(1), \ldots, \mathbf{x}(k) \in C$ che non corrispondono la classificazione, applichiamo per ognuno di essi la tecnica di somma vista nell'esempio precedente andando a calcolare un nuovo vettore $\mathbf{w}$ come segue:
$\mathbf{w}(2) = \mathbf{w}(1) + \mathbf{x}(1)$
$\mathbf{w}(3) = \mathbf{w}(2) + \mathbf{x}(2)$
$\vdots$
$\mathbf{w}(k+1) = \mathbf{w}(k) + \mathbf{x}(k)$
Ma allora possiamo scrivere $\mathbf{w}(k+1)$ come:

$$\mathbf{w}(k+1) = \mathbf{x}(1) + \ldots + \mathbf{x}(k)$$

Visto che C_1 e C_2 sono linearmente separabili allora esiste $\mathbf{w}_*$ tale che
$\mathbf{w}_*\mathbf{x} > 0 \ \forall \mathbf{x} \in C$
Consideriamo il minimo punto che non soddisfa la classificazione, sia α. Ovvero:
$\alpha = min \ \mathbf{w}_*^T\mathbf{x}$
Visto che α è il punto minimo, allora sappiamo che sommando tutti i k punti otterremo qualcosa che è certamente maggiore o uguale a $k\alpha$. Ovvero:
$\mathbf{w}_*^T\mathbf{w}(k+1) = \mathbf{w}_*^T\mathbf{x}(1) + \ldots + \mathbf{w}_*^T\mathbf{x}(k) \geq k\alpha$
Ricordiamo adesso la diseguaglianza di Cauchy-Schwarz:

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

Dunque

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

Teniamo a mente questa diseguaglianza perché ci servirà per la conclusione.

Adesso consideriamo un altro modo di vedere $\mathbf{w}(k+1)$. Infatti sappiamo che $\mathbf{w}(k+1) = \mathbf{w}(k) + \mathbf{x}(k)$ e dunque sempre da Cauchy-Schwarz abbiamo che:
$\| \mathbf{w}(k+1) \|^2 = \| \mathbf{w}(k) + \mathbf{x}(k) \|^2 = (\mathbf{w}(k) + \mathbf{x}(k)) \ (\mathbf{w}(k) + \mathbf{x}(k)) =$
$\| \mathbf{w}(k) \|^2 + \| \mathbf{x}(k) \|^2 + 2\mathbf{w}^T(k) \ \mathbf{x}(k)$.

Ma poiché i k punti non sono classificati correttamente, il termine $2\mathbf{w}^T(k) \ \mathbf{x}(k)$ è negativo. Dunque vale che:
$\| \mathbf{w}(k+1) \|^2 \leq \| \mathbf{w}(k) \|^2 + \| \mathbf{x}(k) \|^2$
Dunque possiamo applicare questa formula a tutti i k punti ottenendo:
$\| \mathbf{w}(2) \|^2 \leq \| \mathbf{w}(1) \|^2 + \| \mathbf{x}(1) \|^2$
$\| \mathbf{w}(3) \|^2 \leq \| \mathbf{w}(2) \|^2 + \| \mathbf{x}(2) \|^2$
$\vdots$
Ovvero:
$\| \mathbf{w}(k+1) \|^2 \leq \ \sum_{i=1}^{k} \| \ x(i) \|^2$

A questo punto sia $\beta = max \ \| \ \mathbf{x}(n) \|^2$ con $x(n) \in C$
Ma dunque:
$\| \mathbf{w}(k+1) \|^2 \leq \ k\beta$.

Per concludere ripercorriamo idealmente quello che abbiamo mostrato finora: sappiamo che la quantità $\| \mathbf{w}(k+1) \|^2$ è vincolata dalla due diseguaglianze che seguono:

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

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

Notiamo che la prima cresce quadraticamente mentre la seconda linearmente.

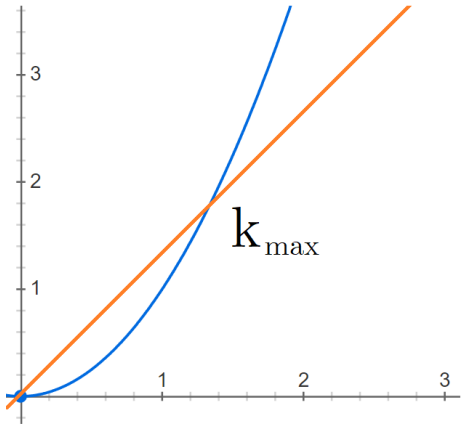


Figura 4: Esiste un intervallo compreso tra 0 e k_{max} in cui sono verificate entrambe le diseguaglianze.

Affinché entrambe queste diseguaglianze siano vere, deve esistere un intervallo come quello mostrato in Figura 4

Notiamo che però esiste un k_{max} finito per il quale entrambe le disequazioni sono verificate e l'algoritmo termina. Dunque abbiamo la certezza che al massimo dopo k_{max} iterazioni l'algoritmo termina. □

Vediamo come applicare quanto visto nella pratica. Prima di farlo introduciamo velocemente il concetto di *epoca*. Quando l'algoritmo di apprendimento esegue una iterazione di dati del dataset, abbiamo completato un epoca.

Esempio 2 : supponiamo di voler allenare un percettrone sui due seguenti insiemi:
$C_1 = \{(1,1),(1,-1),(0,-1)\}$ classificati come 1
$C_2 = \{(-1,-1),(-1,1),(0,1)\}$ classificati come -1

Supponiamo inoltre che il vettore dei pesi iniziali sia $\mathbf{w}(1) = [1,0,0]$ e che $\eta = 1$ Prima di tutto dobbiamo aggiungere in testa ad ogni punto il parametro aggiuntivo per il bias. Successivamente dobbiamo invertire tutti gli elementi della classe C_2. Nel complesso otteniamo i seguenti punti:
$(1,1,1),(1,1,-1),(1,0,-1),(-1,1,1),(-1,1,-1),(-1,0,-1)$
Andiamo ad analizzare lo sviluppo di ogni epoca ricordando che in ogni epoca dobbiamo verificare per ogni punto se:

$$w(n+1) = \begin{cases} w(n) + \eta x(n), & \text{se } w^T(n)x(n) \leq 0 \\ w(n), & \text{altrimenti} \end{cases}$$

Epoca 1:

- $[1,0,0] * [1,1,1] > 0$ dunque non aggiorno
- $[1,0,0] * [1,1,-1] > 0$ dunque non aggiorno

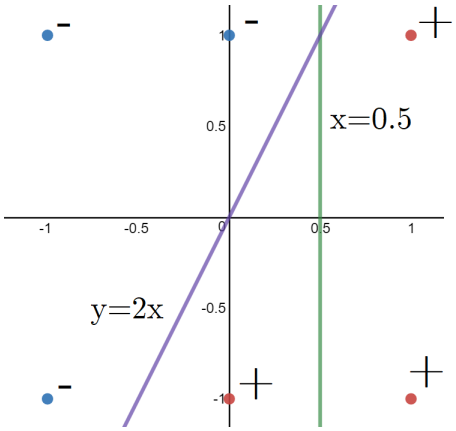


Figura 5: Le rette ottenute dopo le varie epoche

- $[1,0,0] * [1,0,-1] > 0$ dunque non aggiorno
- $[1,0,0]*[-1,+1,+1] \leq 0$ dunque devo aggiornare! Otteniamo $w(2)_{temp} = [0,1,1] + [-1,+1,+1] = [0,1,1]$
- $[0,1,1] * [-1,1,-1] \leq 0$ anche in questo caso devo aggiornare. Ottengo $w(2)_{temp} = [0,1,1] + [-1,1,-1] = [-1,2,0]$
- $[-1,2,0] * [-1,0,-1] > 0$ dunque non devo aggiornare

Dunque il nuovo vettore è $w(2) = [-1,2,0]$ che rappresenta la retta $x = 1/2$
Procediamo ad analizzare l'epoca 2.

Epoca 2:

- $[-1,2,0] * [1,1,1] > 0$ dunque non aggiorno
- $[-1,2,0] * [1,1,-1] > 0$ dunque non aggiorno
- $[-1,2,0] * [1,0,-1] \leq 0$ dunque dobbiamo aggiornare. Otteniamo $w(3)_{temp} = [-1,2,0] + [1,0,-1] = [0,2,-1]$
- $[0,2,-1] * [-1,1,1] > 0$ dunque non aggiorno
- $[0,2,-1] * [-1,1,-1] > 0$ dunque non aggiorno
- $[0,2,-1] * [-1,0,-1] > 0$ dunque non aggiorno

Dunque il vettore ottenuto è $w(3) = [0,2,-1]$ corrispondente alla retta $y = 2x$. Potremmo eseguire anche per la terza epoca ma non avremo alcun cambiamento ulteriore. Dunque l'algoritmo terminerebbe subito dopo la terza iterazione. Mostriamo in Figura 5 i risultati ottenuti dopo le varie iterazioni

Purtroppo il percettrone è semplice da allenare ma anche limitato visto che non riesce a risolvere casi non linearmente separabili.

Nel caso in cui abbiamo a disposizione un insieme di dati non linearmente separabili è preferibile utilizzare il *metodo Adaline*(Adaptive Linear Element). Con questo metodo ammettiamo una tolleranza agli errori e calcoliamo per ogni iterazione l'errore totale cercando di minimizzarlo. L'errore è calcolato con il metodo dei minimi quadrati. In particolare, l'errore su un certo punto è calcolato come:

$$E^{(k)}(w) = \frac{1}{2}(d^{(k)} - y^{(k)})^2 = \frac{1}{2}(d^{(k)} - \sum_{j=0}^{m} x_j^{(k)} w_j)^2$$

L'errore totale è quindi:

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

Per fortuna la funzione E_{tot} è derivabile ovunque. Dunque possiamo calcolare il gradiente dell'errore che ci darà la direzione in cui l'errore cresce. A quel punto basterà muoversi in direzione opposta. Ovvero:

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

Dunque un passo possibile dell'addestramento potrebbe essere un passo eseguito nella direzione calcolata dal gradiente:
$w(k+1) = w(k) - \eta(\nabla E^{(k)}(w))$

Una forma alternativa per il gradiente potrebbe essere quella che segue:
$$\frac{\partial E_{tot}}{\partial w_j} = \sum_{k=1}^{N} \frac{\partial E^{(k)}(w)}{\partial w_j} = \sum_{k=1}^{N} \frac{\partial}{\partial w_j} \frac{1}{2}(d^{(k)} - \sum_{j=0}^{m} x_j^{(k)} w_j)^2$$

Ma poiché sappiamo che la derivata di ogni singolo errore è pari a:

$$\frac{\partial E^{(k)}(w)}{\partial w_j} = \frac{\partial E^{(k)}(w)}{\partial y^{(k)}} \frac{\partial y^{(k)}}{\partial w_j} = -(d^{(k)} - y^{(k)}) x_j^{(k)}$$

Allora possiamo definire la *Delta rule* per l'aggiornamento dei pesi:

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

Su tale delta rule è basato l'algoritmo di addestramento del metodo Adaline.

10 / 10 /2017

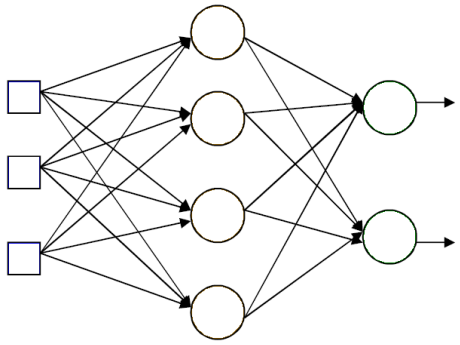


Figura 6: Rete neurale multistrato feed forward

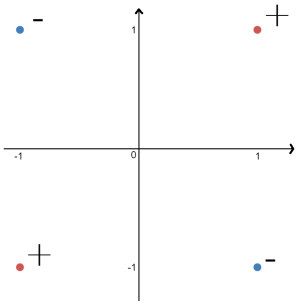


Figura 7: Problema dello XOR. Non è linearmente separabile

3 Multilayer Perceptrons

Lo scopo di questa sezione è quello di andare a studiare le reti multistrato e il loro apprendimento supervisionato. Mostriamo nuovamente, in Figura 6, lo schema di una rete multistrato feed-forward

Di fatto una rete di questo tipo possiede: neuroni di input, neuroni di output e neuroni nascosti. Lo scopo dei neuroni nascosti è quello di permetterci di gestire problemi non linearmente separabili. Vediamo un esempio di problema non linearmente separabile

Esempio 1 : il tipico problema non linearmente separabile è il *problema dello XOR* mostrato in Figura 7

Visto che non è possibile determinare una retta che separi linearmente i dati in positivi e negativi, dobbiamo usarne due. Ogni neurone gestisce una delle due rette.

Più in generale abbiamo che se un insieme di punti è linearmente separabile mediante n rette, serviranno n neuroni interni, uno per stabilire la migliore posizione di una delle rette.

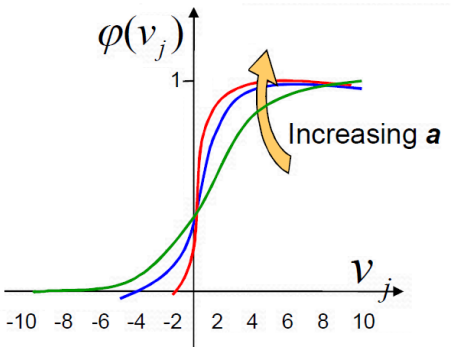


Figura 8: Funzione sigmoide

Riprendiamo la funzione sigmoide. È stata la funzione di apprendimento principe per molti anni visto che è continua e derivabile[3]. Riportiamo nuovamente la funzione sigmoide per un generico nodo j è:

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

dove $a > 0$ e $v_j = \sum_i w_{ji} y_i$. Il valore w_{ji} indica il peso del collegamento dal nodo i al nodo j, mentre y_i indica l'output del nodo i. Riportiamo in la funzione sigmoide

Notiamo che al crescere di a, aumenta anche la ripidità della sigmoide.
L'algoritmo di apprendimento per i valori dei pesi è un algoritmo di *backward propagation* detta anche *retropropagazione*. Consiste nel modificare il valore dei pesi per ottenere un errore minimo. Il minimo che troveremo non è quello assoluto ma non è un problema perché nel minimo assoluto esiste anche una maggiore probabilità di overfitting dei dati. Anche questo algoritmo procede per epoche.
Il primo passo consiste nel dare un valore casuale ad ogni peso della rete[4]. Tramite questo input riceviamo un output che possiamo verificare essendo in un caso di apprendimento supervisionato. Successivamente l'errore ottenuto in output può essere retropropagato all'indietro fino al livello successivo all'input. Mostriamo l'intero procedimento in Figura 9

La quantità da minimizzare è l'*errore quadratico medio*. Per farlo abbiamo a disposizione le coppie $(x(n), d(n))$ dove $x(n)$ è l'input sulla rete, mentre $d(n)$ è l'output atteso. Definiamo la quantità *errore* sull'n-esimo esempio come:

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

[3] Anche se è stata poi scalzata dalla funzione re-lu che non è derivabile. Ma questo non rappresenta particolarmente un problema perché la si può derivare a tratti

[4] Ricordiamo che esiste un peso per ogni collegamento fra due nodi

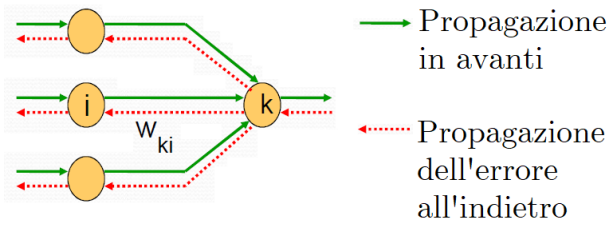


Figura 9: Propagazione in avanti e all'indietro all'interno della rete

che è fondamentalmente la differenza tra l'output atteso e quello da noi trovato. L'errore totale è la somma dei vari errori elevati al quadrato: $E(n) = \frac{1}{2}\sum_j e_j^2(n)$
L'errore quadratico medio è appunto la media degli errori $E(n)$, ovvero:

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

3.1 Derivazione della regola di apprendimento

La regola per l'aggiornamento dei pesi è basata sul metodo di discesa del gradiente ed è pari a:

$w_{ji} = w_{ji} + \Delta w_{ji}$

dove
$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$ con $\eta > 0$
Notiamo che η ci fornisce l'informazione su quanto ci spostiamo nel corso dell'apprendimento. Solitamente è un valore piccolo $0 < \eta < 1$ anche se non abbiamo indicazioni definitive in merito. In generale sappiamo che un η troppo basso rallenterà l'apprendimento ma un η troppo alto renderà l'intero algoritmo instabile.
Notiamo che dato l'input

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

Possiamo dire che:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial v_j}\frac{\partial v_j}{\partial w_{ji}}$$

Intuitivamente stiamo dicendo che l'errore E varia rispetto a $w_{ji}(\frac{\partial E}{\partial w_{ji}})$ perché andiamo a cambiare il valore dell'input $v_j(\frac{\partial E}{\partial v_j})$ che a sua vota cambia perché andiamo a modificare il valore dei pesi$w_{ji}(\frac{\partial v_j}{\partial w_{ji}})$.

Adesso, andando a definire come $\delta_j = -\frac{\partial E}{\partial v_j}$, visto che $\frac{\partial v_j}{\partial w_{ji}} = y_i$ allora otteniamo

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

Notiamo però che per calcolare Δw_{ji} dobbiamo conoscere δ_j. Per determinarlo abbiamo due casi possibili

1. Il neurone j è un neurone di output: in questo caso abbiamo che
   $\delta_j = -\frac{\partial E}{\partial v_j} = -\underbrace{\frac{\partial E}{\partial e_j}}_{=e_j}\underbrace{\frac{e_j}{\partial y_j}}_{=-1}\underbrace{\frac{\partial y_j}{\partial v_j}}_{=\varphi'(v_j)} = -e_j(-1)\varphi'(v_j)$

   Questo perché $e_j = d_j - y_j$ mentre $y_j = \varphi(v_j)$
   Quindi quando j è un nodo di output, il peso w_{ji} relativo al collegamento dal neurone i al neurone j è pari a:

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

2. Il neurone non è neurone di output: questo caso è leggermente più complesso, visto che il neurone non può sapere il risultato della classificazione, non essendo neurone di output. Per rimediare, il neurone va a verificare cosa è successo nel neurone successivo. Dunque:
   $\delta_j = -\frac{\partial E}{\partial v_j} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial v_j}$
   L'idea dietro alla formula precedente è che, visto che una variazione dell'input produce effetti a cascata, verifichiamo cosa è successo nei livelli successivi. In particolare osserviamo che:

   $\frac{\partial y_j}{\partial v_j} = \varphi'(v_j)$ e che $\frac{\partial E}{\partial y_j} = \sum_{k\ liv.\ succ.} \frac{\partial E}{\delta v_k}\frac{\delta v_k}{\partial y_j}$

   Allora abbiamo che:

   $\delta_j = -\sum_{k\ liv.\ succ.} \delta_k w_{kj}\varphi'(v_j)$

   Quindi, quando j non è nodo di output, il peso w_{ji} relativo al collegamento dal neurone i al neurone j è pari a:

   $$\Delta w_{ji} = \eta y_i \varphi'(v_j) \sum_{k\ liv.\ succ.} \delta_k w_{kj}$$

Complessivamente possiamo riassumere dicendo che la *delta rule* è
$\Delta w_{ji} = \eta \delta_j y_i$
dove:

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j), & \text{se j è nodo di output} \\ \varphi'(v_j) \sum_{k\ liv.\ succ.} \delta_k w_{kj}, & \text{se j è un nodo interno} \end{cases}$$

dove $\varphi'(v_j) = y'_j = ay_j(1 - y_j)$ se consideriamo una sigmoide. Notiamo che è particolarmente comoda perché la derivata del valore di output y_j dipende direttamente da y_j.

17 / 10 /2017

3.2 Parametri e iper parametri

Come dicevamo precedentemente, non abbiamo alcuna indicazione sul valore che debbano assumere gli *iper parametri* della rete, ovvero quei parametri inerenti la rete che non sono relativi ai collegamenti fra i nodi come i parametri, bensì illustrano alcune proprietà della rete, come il numero di nodi, il numero di collegamenti...
Un esempio di iper parametro incontrato precedentemente è η che determina il valore di quanto un algoritmo apprenda ad ogni passo.
Consideriamo proprio η. Come già detto un valore troppo basso rallenterà l'apprendimento, ma un valore troppo alto porterà instabilità. Questo perché supponiamo di essere diretti verso un minimo grazie all'algoritmo di apprendimento. Se η è un valore troppo grande da una iterazione all'altra rischiamo di "superarlo". Certamente all'iterazione successiva possiamo "tornare indietro"; ma nuovamente, se η è troppo grande potremmo non avere mai una convergenza al minimo.
Esistono diversi studi teorici che propongono di considerare il segno della derivata. Infatti dopo aver superato un minimo (o un massimo) il segno della derivata cambia e questo ci da l'informazione necessaria per capire che abbiamo superato il minimo.
Esiste invece una delta rule generalizzata che trae ispirazione da tali studi:

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

Con $0 \leq \alpha \leq 1$
Notiamo che la prima parte $\alpha \Delta w_{ji}(n-1)$ riguarda l'iterazione precedente. Intuitivamente ci dice che se stavamo procedendo nella discesa verso il minimo al passo precedente, e continuiamo a procedere in tale direzione, questo termine rafforza la nostra discesa. Se invece superiamo il minimo, il primo termine attenuerà il superamento, essendo di segno opposto il $\Delta w_{ji}(n-1)$. Notiamo che però abbiamo dovuto introdurre un nuovo iper parametro α che deve essere comunque stabilito da noi.

Una volta stabilita la generalizzazione, possiamo vedere le due branche di algoritmi di apprendimento più note:

1. Apprendimento per pattern: prendiamo un esempio e andiamo ad aggiornare il valore dei pesi della rete. Ripetiamo il procedimento per ogni esempio, finendo un'epoca. Ripetiamo l'intero procedimento per più epoche

2. Apprendimento per epoche: andiamo a considerare tutti i dati e solo alla fine di un'epoca andiamo ad aggiornare i valori dei parametri.

È stato verificato sperimentalmente che il secondo approccio è più corretto ma anche più lento. Dunque si preferisce utilizzare la tecnica per pattern. Ad ogni modo la tecnica di apprendimento per pattern presenta dei rischi. Il primo è che gli esempi forniti dovrebbero essere sempre mescolati in modo casuale. Infatti cercare di effettuare apprendimento su elementi di una stessa classe tutti consecutivamente rischia di condurre l'algoritmo di apprendimento verso un minimo che però vada bene solo per quella classe.

Una domanda che è interessante porsi è quando sia necessario fermarsi nell'apprendimento. La risposta è che dobbiamo interrompere l'apprendimento quando l'errore smette di ridursi o comunque quando decresce troppo lentamente tra un'iterazione e la successiva. Uno strumento per verificare quando sia il caso di interrompere l'addestramento è il *validation set*. Solitamente supponendo di avere un certo numero di esempi a disposizione:

- Il $30\% - 35\%$ viene utilizzato per effettuare il test
- Il restante $65\% - 70\%$ viene utilizzato per il training. Di questa parte, circa il 15% viene usato per il validation set, che di fatto è un insieme di esempi

In particolare, il validation set viene usato dopo che la rete è stata almeno in parte addestrata. Verifichiamo come la rete si comporti sugli esempi del validation set e se si comporta bene interrompiamo l'addestramento per andare ad effettuare il testing. Altrimenti proseguiamo con l'addestramento. Questo procedimento può essere effettuato più volte e possiamo riusare lo stesso insieme di dati come validation set ma possiamo anche usare una tecnica più avanzata, detta *cross-validation* che divide in n parti il training set e, ad ogni epoca, usa un validation set diverso per verificare il comportamento della rete.
Un altro problema in questo ambito sono i dati *outlayer* particolarmente sfortunati il cui valore differisce molto da quello degli altri soltanto a causa di un'errata presa dati. Questi dati andrebbero individuati ed eliminati prima di addestrare la rete poiché durante l'addestramento rischiano di guidarlo in modo errato. Ma forse ancor peggio, se gli outlayer sono contenuti nel test set, allora mettono in dubbio -indebitamente- l'intero processo di apprendimento.

Vediamo adesso qualche punto saliente relativo alla costruzione di una rete neurale:

- Rappresentazione dei dati: solitamente i dati con cui abbiamo a che fare sono numerici. Le varie caratteristiche dei dati possono però oscillare in intervalli molto diversi. Consideriamo per fare un esempio la classificazione di un frutto che consideri due parametri: il *ph* e il *peso* espresso in grammi. Il *ph* varia tra 1 e 14 mentre il peso potrebbe variare tra 1 e

1000. Ovviamente nell'allenamento dei pesi il valore 1000 peserebbe molto di più di un *ph* a 14, e questo non avrebbe senso. Per questo motivo normalizziamo i dati secondo la formula che segue

$$x_i = \frac{x_i - min_i}{max_i - min_i}$$

Dunque tutti i dati avranno un valore x_i tale che $0 \leq x_i \leq 1$
Questo è un accorgimento che deve essere sempre seguito se vogliamo avere una rappresentazione fedele

- Topologia della rete: per topologia della rete intendiamo il numero di livelli, di nodi e di collegamenti tra di essi. Inizialmente si utilizzavano reti aventi un solo livello nascosto. Dai primi anni del nuovo millennio, per gestire problemi più complessi sono stati introdotti molti più livelli. Reti di questo tipo possono essere costruite in due modi:

  1. Andando a costruire una rete inizialmente molto complessa effettuando un'operazione di *pruning*(sfoltimento) successivamente
  2. Partendo da una rete semplice e andando ad aggiungere i nodi quando necessario.

- Parametri della rete: per quanto riguarda i parametri e gli iperparametri abbiamo solo dei suggerimenti riguardo il valore da assegnare. Solitamente tali valori sono compresi tra -1 e 1, oppure tra -0.5 e 0.5.
  Ad esempio per il valore di η solitamente vale che: $0.9 \leq \eta \leq 1$(mai mettere $\eta = 0$!)

- Training: per l'allenamento di una rete dovremmo sempre considerare quanti parametri abbiamo e quanti esempi abbiamo a disposizione. La *thumb rule* dice che il numero di esempi dovrebbe essere sempre dalle quattro alle dieci volte il numero dei pesi da assegnare. Al di là del valore moltiplicativo, la thumb rule ci dice una cosa molto importante: deve esistere una proporzione tra il numero di esempi e la dimensione della rete che usiamo.
  Un'alternativa è usare la seguente disuguaglianza:

$$N > \frac{|W|}{(1-a)}$$

  Dove N è il numero di esempi, W il numero dei pesi, e a l'accuratezza attesa

Reti neurali di questo tipo possono essere utilizzate per approssimare funzioni booleane, funzioni continue per parti e funzioni continue. Esistono risultati teorici che sostengono questa tesi, come quello che segue.
Il *teorema di approssimazione*, di cui diamo soltanto l'enunciato, ci dice che:

Teorema 3.1 (Teorema di approssimazione)**.** *Sia $\varphi(n)$ una funzione non costante, limitata e crescente monotona. Sia I_{m_0} l'ipercubo unitario m_0-dimensionale, ovvero con $[0,1]^{m_0}$. Allora, presa una qualunque funzione $f \in C(I_{m_0})$ e $\epsilon > 0$, esiste un intero m_1 e un insieme di costanti reali α_i, b_i e w_{ij} tale che:*

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_0} \alpha_i \varphi(\sum_{j=1}^{m_0} w_{ij} x_j + b_i)$$

dove $F(x_1, \ldots, x_{m_0})$ è un'approssimazione di f, ovvero vale che:

$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \epsilon$$

Notiamo che $\sum_{j=1}^{m_0} w_{ij} x_j + b_i$ è l'input ad un certo nodo e che dunque $\varphi(\sum_{j=1}^{m_0} w_{ij} x_j + b_i)$ è il suo output. Infine $F(x_1, \ldots, x_{m_0})$ è l'output dell'intera rete.
Notiamo inoltre che ϵ può essere preso piccolo a piacere, dunque l'errore tra la funzione F che abbiamo creato e la f di cui vogliamo effettuare l'approssimazione, può essere ridotto quanto vogliamo. Quello che ci serve è avere una funzione φ con le proprietà richieste nella premessa del teorema.
Fortunatamente la funzione sigmoide e la tangente iperbolica rispettano queste condizioni.
Dobbiamo fare ancora alcune considerazioni riguardo il teorema.
È un teorema di implicazione, dunque se è vero l'antecedente è anche vero il conseguente, ma se l'antecedente non è vero, questo non ci dice nulla sul conseguente(dunque è una condizione sufficiente).
Un fatto altrettanto importante è che è un teorema di esistenza e non di costruzione. Se valgono le premesse del teorema sappiamo che una funzione F esiste, ma non sappiamo certamente quali siano i valori da settare per ottenerla.
Sotto queste condizioni, le reti neurali sono in grado di risolvere problemi non linearmente separabili come il riconoscimento dei caratteri, dei volti e della voce, oppure effettuare una classificazione di oggetti mediante caratteristiche salienti.
Un'ultima cosa da dire è che le reti neurali forniscono buone risposte su dati futuri se tale tali dati sono in qualche modo omogenei con i dati su cui la rete si è allenata. Se alleniamo una rete sotto certe condizioni e successivamente tali condizioni cambiano radicalmente, non possiamo aspettarci che la rete si comporti ancora bene.

24 / 10 /2017

4 Reti radiali e Extreme learning

Vediamo adesso due approcci alternativi al multilayer perceptron che sono simili fra loro ma molto diversi da quelli visti fino ad ora.

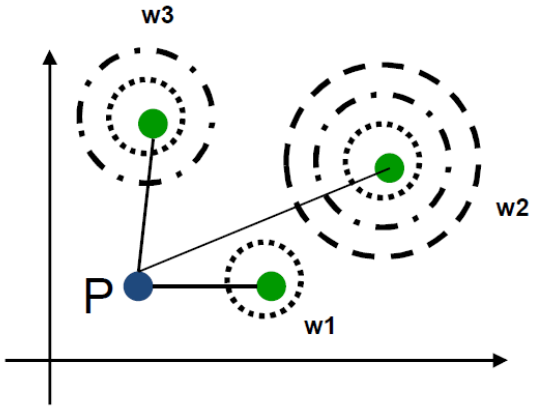


Figura 10: Ogni centro rappresenta una funzione radiale

4.1 Reti neurali radiali

Per parlare delle reti neurali radiali dobbiamo prima di tutto definire una *funzione radiale*. Intuitivamente una funzione radial-basis (RBF) fornisce un output che dipende dalla distanza fra un vettore di input e un vettore memorizzato, detto *centro*. Questa situazione è mostrata in Figura 10

Di fatto, se un input è troppo lontano dal centro, il neurone non si attiva. I cerchi concentrici intorno al centro rappresentano l'ipotetica deviazione entro la quale il neurone ancora riesce ad attivarsi.
Più formalmente abbiamo che una funzione radiale è così definita:

$$\phi_\sigma(\parallel x - t \parallel) \tag{1}$$

Dunque l'output dipende dalla distanza tra l'input x e il centro t, tenendo di conto di quanto sia lo scostamento o *spread* σ rispetto a t[5].
Cominciamo a comprendere che per allenare la rete dovremo giocare anche sui parametri t e σ.
Una differenza con le funzioni viste in precedenza(come la sigmoide) è che le precedenti funzioni erano *aperte*, ovvero potevano fornire un output pari a 1 anche per input infinitamente grandi. Qui invece abbiamo una zona ben localizzata che stimola l'attivazione del neurone.
L'output di una rete neurale che possieda una funzione radiale al penultimo livello, come quella mostrata in Figura 11 , sarà

$$y = w_1\phi_1(\parallel x - t_1 \parallel) + \ldots + w_{m1}\phi_{m1}(\parallel x - t_{m1} \parallel) \tag{2}$$

Cerchiamo però di generalizzare e consideriamo una rete che fornisca in output un vettore.Dunque consideriamo il *problema dell'interpolazione*.

[5]Intuitivamente σ rappresenta una varianza

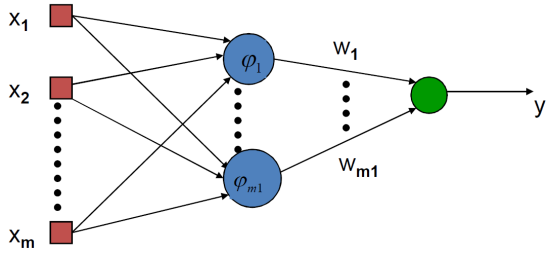


Figura 11: Rete che sfrutta una RBF e che fornisce uno scalare y in uscita

Dato un insieme di N punti distinti(esempi) $\{x_i \in \mathbb{R}^m, i = 1, \ldots, N\}$ e un insieme di N numeri reali(centri) $\{d_i \in \mathbb{R}, i = 1, \ldots N\}$ allora vogliamo cercare di trovare una funzione $F : \mathbb{R}^m \to \mathbb{R}$ tale per cui valga la condizione $F(x_i) = d_i$.
Se scegliamo come F la seguente:

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

Allora abbiamo che:

$$\underbrace{\begin{bmatrix} \phi_1(\| x_1 - x_1 \|) & \ldots & \phi_N(\| x_1 - x_N \|) \\ \vdots & & \vdots \\ \phi_1(\| x_N - x_1 \|) & \ldots & \phi_N(\| x_N - x_N \|) \end{bmatrix}}_{=\Phi} \underbrace{\begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}}_{w} = \underbrace{\begin{bmatrix} d_1 \\ \vdots \\ d_n \end{bmatrix}}_{d} \tag{3}$$

Ovvero vale che

$$\Phi w = d \tag{4}$$

Notiamo alcune cose: intanto la matrice Φ è quadrata. Inoltre abbiamo posto la condizione che i punti siano distinti perché sennò in certo punti di Φ avremo valori nulli. Questo non ci andrebbe molto bene perché fra poco vedremo che ci basiamo su una tecnica di inversione di matrice. E affinché una matrice sia invertibile deve valere che il determinante sia non nullo[6].
Notiamo inoltre che Φ è quadrata, dunque per ogni esempio abbiamo un neurone interno. Questo è esagerato sia in termini computazionali, sia perché con questa tecnica tendiamo a fare overfitting. Tra poco vedremo che preferiamo utilizzare un minore numero di neuroni nascosti.

Ad ogni modo esistono varie funzioni ϕ la cui matrice Φ è invertibile. Una è la *multiquadratics*

$$\phi(r) = (r^2 + c^2)^{1/2}$$

Dove $c > 0$ e $r = \| x - t \|$

[6]Dunque la matrice è non singolare o altrimenti detta invertibile

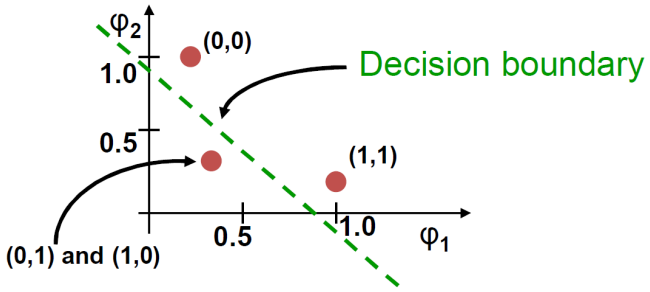


Figura 12: Il problema dello XOR mappato nello spazio ϕ_1, ϕ_2

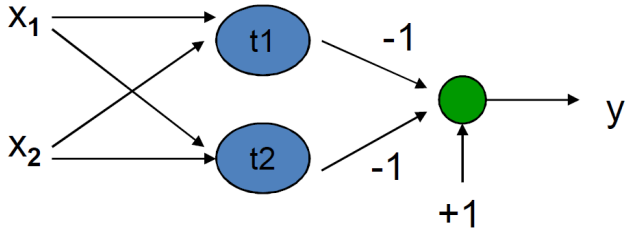


Figura 13: La rete relativa al problema dello XOR

Oppure abbiamo la funzione gaussiana che è anche la più usata:

$$\phi(r) = e^{-\frac{r^2}{2\sigma^2}}$$

con $\sigma > 0$
Andiamo ad applicare quanto visto sul solito esempio dell'OR esclusivo

Esempio 1 : consideriamo il solito grafico relativo allo XOR.
Supponiamo di utilizzare una funzione gaussiana. Otteniamo che, con i centri $t_1 = (1,1)$ e $t_2 = (0,0)$, le due funzioni gaussiane ottenute(una per ogni centro), sono:

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

Se applichiamo le due funzioni gaussiane ϕ_1 e ϕ_2 il grafico che otteniamo è quello mostrato in Figura 12

Abbiamo ottenuto uno spazio in cui i punti del problema dello XOR sono linearmente separabili. La rete che usiamo per gestire questa situazione è quella mostrata in Figura 13

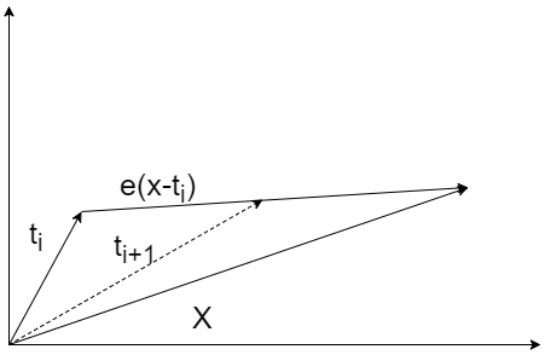


Figura 14: Spostamento del centro t_i verso il dato x. Notiamo che infatti il centro all'iterazione t_{i+1} si sarà avvicinato

Grazie a quanto visto finora possiamo ricavare che dobbiamo allenare tre cose all'interno delle reti radiali. Studiamole una per una e vediamo come allenarle:

- Allenare la posizione dei centri: visto che abbiamo diversi centri da posizionare, dobbiamo capire dove metterli. Nello specifico possiamo farlo in due modi. O li piazziamo a caso, oppure(meglio) usiamo un algoritmo di clustering che si compone delle fasi seguenti:

  1. Inizializzazione: settiamo casualmente gli m_1 centri
  2. Introdurre i dati: inseriamo un x preso dallo spazio di input
  3. Matching di similarità: troviamo l'indice del centro più vicino ad x minimizzando la distanza:

     $i(x) = arg\ min_k \parallel x(n) - t_k(n) \parallel$
  4. Aggiornamento dei centri: spostiamo ogni centro in base a tutti i dati che sono stati assegnati a quel centro

     $$t_i(n+1) = \begin{cases} t_i(n) + \eta(x(n) - t_i(n)), & \text{se } i = i(x) \\ t_i(n), & \text{altrimenti} \end{cases}$$

     Intuitivamente è il termine $\eta(x(n) - t_i(n))$ a spostare il centro. Questo è rappresentato in
  5. Iterare: procediamo a ripetere questi passi partendo dal punto 2 fino a quando la situazione cessa di cambiare

- Allenare lo spread: lo spread viene solitamente determinato dopo aver posizionato i centri, tramite normalizzazione. La formula usata è quella che segue: se d_{max} è la massima distanza tra i centri e m_1 è il numero di centri, allora

$$\sigma = \frac{d_{max}}{\sqrt{m_1}} \tag{5}$$

- Allenare i pesi: riprendiamo quanto dicevamo prima. Avere un centro per ogni input è un problema sia computazionalmente, sia perché causa overfitting. Per questo motivo decidiamo di utilizzare un numero inferiore di centri che ci fornisca un risultato approssimato. Ovvero cercheremo di fare in modo che per ogni esempio i valga che:

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

  Quindi la nostra rete non fornisce un risultato esatto su d_i ma solo una sua approssimazione. Per una rete che fornisca N esempi e $m1$ centri avremo che:

  $$\begin{bmatrix} \phi_1(\| x_1 - t_1 \|) & \ldots & \phi_{m1}(\| x_1 - t_{m1} \|) \\ \vdots & & \vdots \\ \phi_1(\| x_N - t_1 \|) & \ldots & \phi_{m1}(\| x_N - t_{m1} \|) \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_{m1} \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ d_N \end{bmatrix} \quad (6)$$

  Anche in questo caso possiamo scrivere:

  $\Phi w = d$

  Notiamo che a questo punto la matrice Φ non è più quadrata come era in precedenza. Dunque come possiamo invertirla? Non la invertiamo ma effettuiamo una *pseudo inversione*.
  Possiamo certamente scrivere che

  $\Phi w = d \iff \Phi^T \Phi w = \Phi^T d$

  Da cui otteniamo che:

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

  Andiamo a chiamare $\Phi^+ = (\Phi^T \Phi)^{-1} \Phi^T$ da cui:

  $w = \Phi^+ d$

  che viene detta *matrice pseudo inversa*

Cosa è cambiato rispetto ad una normale rete neurale? Esistono alcune differenze che meritano di essere menzionate

- Architetturalmente le reti radiali hanno un solo livello nascosto, mentre le feed forward ne hanno più di uno
- Il modello dei neuroni nascosti è diverso(non lineare) da quello dei nodi di output(lineare) mentre nelle feed forward è comune(entrambi non lineari)
- La funzione di attivazione nelle reti radiali sfrutta un concetto di distanza euclidea mentre le feed forward usano il concetto di prodotto scalare

- Dal punto di vista dell'approssimazione le reti radiali sfruttano funzioni gaussiane locali mentre nelle feed forward l'approssimazione è globale. Con questo intendiamo anche dire che nelle reti radiali i pesi sono calcolati localmente in un colpo solo mentre nelle reti feed forward sono calcolati di iterazione in iterazione

In comune abbiamo che le reti radiali e le feed forward sono esempi di reti feed forward non lineari e sono entrambi approssimatori universali.

4.2 Extreme Learning Machines

Questo approccio che introduciamo è alternativo alla bacpropagation per multy layer perceptron ma usa le tecniche di pseudo inversione viste nella sezione precedente.
L'extreme learning viene presentata come tecnica rivoluzionaria nel 2011. In realtà l'idea è buona ma l'autore l'ha scritta prendendo (senza citare) parte del contenuto del suo lavoro dal lavoro di altri. Vediamolo:
supponendo di avere un primo livello di pesi che viene determinato a caso, se le funzioni di attivazione sono infinitamente differenziabili[7], allora il livello successivo è determinabile analiticamente.
L'idea quindi, è proprio quella di usare al secondo livello della rete una funzione di attivazione radiale e utilizzare tecniche di pseudo inversione.
L'algoritmo procede allo stesso modo delle tecniche di pseudo inversione, partendo da $H\beta = y$ [8] da cui andiamo a ricavare che $\beta = H^{+}T$.
L'*algoritmo di Huang* sfrutta quindi quanto abbiamo detto precedentemente e soffre degli stessi problemi legati alla singolarità. Infatti per matrici quasi singolari abbiamo dei problemi e dobbiamo utilizzare un termine correttivo.

Le prestazioni ottenute dall'algoritmo di Huang, sebbene sbandierate come migliori di tre ordini di grandezza in tempo rispetto alle tecniche di backpropagation, non tengono di conto dell'inizializzazione dei parametri che viene fatta a caso e dunque può portare ad una esplorazione assai ridotta dello spazio dei parametri. Per questo motivo l'algoritmo di Huang deve essere eseguito molte volte con configurazioni iniziali differenti per ottenere dei risultati accettabili. Dunque la complessità in tempo di tale algoritmo è in realtà paragonabile a quella delle tecniche da noi analizzate precedentemente.

31 / 10 /2017

5 Deep Learning: parte 1

Le *reti neurali profonde* sono un particolare tipo di rete neurale costituite da almeno due livelli nascosti. Come mai introdurre più livelli se avevamo detto

[7] Precondizione che verrà smentita da studi successivi
[8] Che corrisponde a $\Phi w = d$

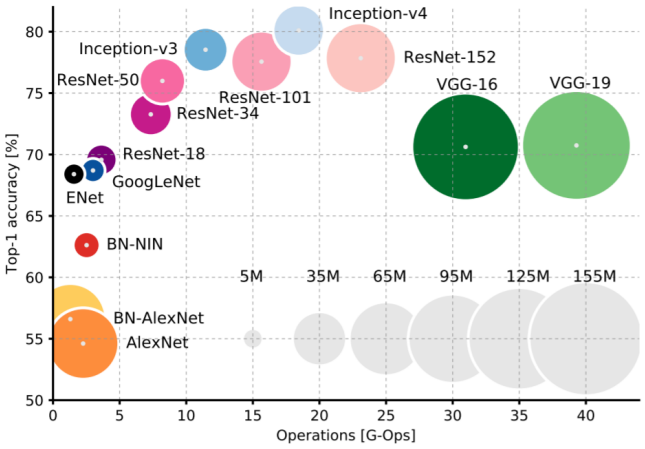


Figura 15: Confronto di accuratezza tra diversi modelli di deep learning

che già un solo livello nascosto rende la rete universale? Questo perché in certi casi ci evita di utilizzare un numero esponenziale di neuroni. E questo non si verifica nelle reti profonde. Inoltre la struttura gerarchica permette un più facile riutilizzo dell'informazione.
Solitamente vengono utilizzati tra i 7 e i 50 livelli nascosti, più raramente anche oltre 100 livelli nascosti. Volendo effettuare un confronto con il corpo umano, solitamente serve l'attivazione di una dozzina di livelli per coordinare l'occhio con la mano.
I parametri che utilizziamo per valutare la complessità di una rete neurale sono almeno il numero di livelli e il numero di pesi. Mostriamo in Figura 15 un confronto tra varie tecniche di deep learning:

Ovviamente un modello con una ridotta complessità e un'elevata accuratezza è il migliore che possiamo sperare di trovare.
Ad ogni modo esistono diverse categorie di modelli di reti neurali profonde

- Modelli feedforward per classificazione supervisionati: in questa categoria ricadono le *convolutional neural network*, le *fully connected deep neural network* e le *Hierarchical Temporal Memory(HTM)*
- Modelli per allenamento non supervisionato: in cui troviamo le *reti stacked*, le *Restricted Boltzmann Machine* e le *Deep Belief Networks*
- Modelli ricorrenti: in cui abbiamo le *Reti neurali ricorrenti* e le *Long Short-Term Memory*

Come mai i modelli di deep learning hanno avuto tanto successo? Le reti neurali convolutive, progettate nel 1998 da LeCun ottenero buone prestazioni su problemi di dimensione ridotta. Emergono in modo preponderante nel 2012 con

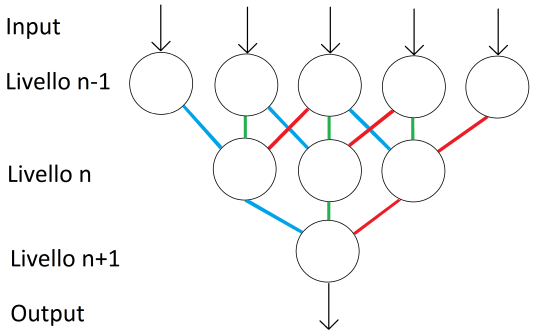


Figura 16: Confronto di accuratezza tra diversi modelli di deep learning

AlexNet perché possono essere addestrate con milioni di immagini e centinaia di classi. In secondo luogo perché permettono di sfruttare le GPU di potenza sempre crescente, riducendo il tempo di addestramento da mesi a giorni. Infine perché fanno uso di una differente funzione di attivazione detta *ReLu* che vedremo in seguito che permette di utilizzare la backpropagation.

5.1 Reti neurali convolutive

Come dicevamo vengono introdotte nel 1998 da LeCun e presentano alcune significative differenze con le reti multi layer perceptron:

- Processing locale dell'informazione: i neuroni sono connessi solo localmente ai neuroni del livello precedente. Questo vuol dire che non ogni neurone non riceve input da tutti i neuroni del livello precedente, bensì solo da alcuni
- Pesi condivisi: i pesi non sono diversi per ogni collegamento. Il valore associato ad un certo peso viene utilizzato in più collegamenti.

Entrambi questi aspetti vengono mostrati in Figura 16

Notiamo infatti che il un neurone del livello n è collegato solamente a tre neuroni su cinque del livello $n-1$. Inoltre abbiamo segnalato con stesso colore i collegamenti pesati allo stesso modo.
I due accorgimenti appena descritti permettono di ridurre significativamente il numero di parametri da determinare e il primo in particolare coglie uno degli aspetti fondamentali delle reti convolutive. La località dei neuroni permette di individuare all'interno di immagini(il principale campo applicativo delle reti convolutive) dei pattern in una certa porzione di immagine. Dunque ogni neurone processa allo stesso modo differenti porzioni di una immagine.

L'operazione su cui si basano le reti convolutive è, per l'appunto, la convoluzione. Nella convoluzione viene solitamente utilizzato un filtro che viene fatto scorrere lungo l'immagine. Questa situazione è riportata in Figura 17

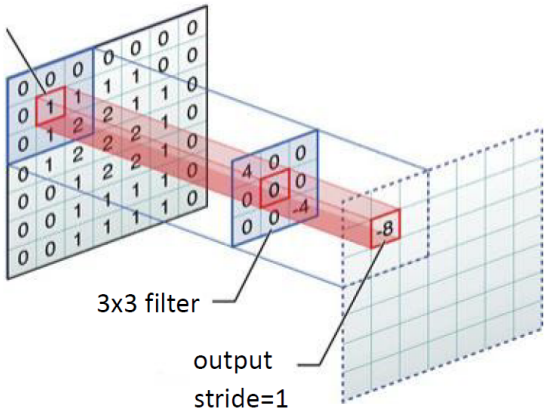


Figura 17: Filtro 3x3 applicato ad un'immagine

Notiamo che, ad esempio il risultato prodotto dal filtro mostrato in figura è -8 perché pari al risultato del prodotto scalare tra la matrice 3x3 in input e il filtro.
Dunque abbiamo due iper parametri da impostare per questo problema:

1. La dimensione del filtro
2. Lo scorrimento del filtro: infatti ogni volta che il filtro è applicato dobbiamo farlo scorrere. Ma di quanto? Un maggiore scorrimento riduce la sovrapposizione tra un'iterazione e l'altra.

Entrambi questi parametri determinano la dimensione della matrice in output. In realtà dobbiamo considerare una complicazione che si verifica nei problemi reali. Solitamente i filtri non sono rappresentati da matrici bidimensionali, bensì da matrici tridimensionali sia in input che in output. Questa situazione è mostrata in Figura 18

Consideriamo dapprima la dimensione tridimensionale della matrice in input $32 \times 32 \times 3$. Solitamente la terza dimensione è semplicemente dovuta ai tre canali RGB che compongono un'immagine. Dunque dobbiamo analizzare ognuno di questi.
La terza dimensione pari a 6 della matrice di output è dovuta alle *feature* che vogliamo individuare. Solitamente associamo ad ogni feature uno dei livelli. Le feature sono quelle caratteristiche che ci interessa rilevare all'interno di una immagine[9]. Dunque avremo un numero di pesi pari alla dimensione del filtro moltiplicata per il numero di feature.
In Figura 18 abbiamo un numero di collegamenti pari a $(28x28x6)x(5x5x3) = 352800$ ma soltanto $6x(5x5x3+1) = 456$ pesi. Dunque rispetto al modello multilayer perceptron abbiamo ridotto sensibilmente il numero di pesi!

[9] Ne sono esempi l'individuazione di una linea verticale, di una orizzontale...

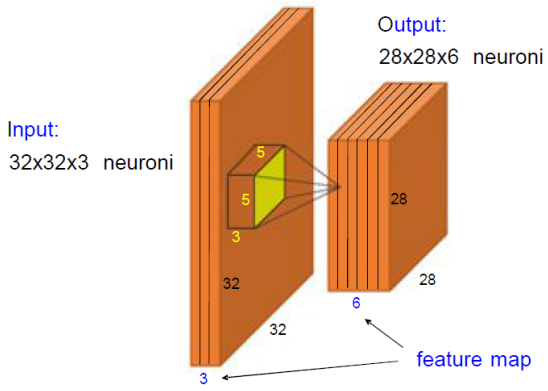


Figura 18: Filtro 5x5x3 applicato ad un'immagine

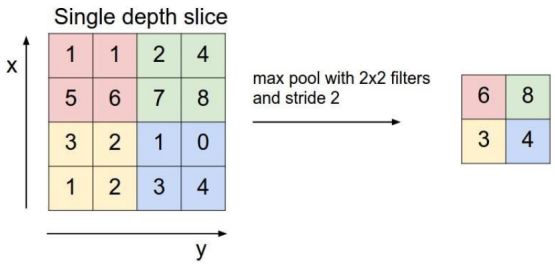


Figura 19: Esecuzione dell'operazione di pooling utilizzando la funzione max pooling. La matrice di output ha dimensione ridotta rispetto a quella di input

Solitamente, dopo un livello di convoluzione abbiamo un livello di *pooling*. In questi livelli andiamo ad effettuare un'aggregazione dell'informazione riducendo a piacere la dimensione delle feature map.
Solitamente l'operazione di pooling consiste nel prendere una certa porzione della matrice e applicarvi una funzione. Le due funzioni più note a riguardo sono la *average pooling* che calcola la media, e la *max pooling* che individua il massimo. Un esempio di operazione di max pooling viene mostrato in Figura 19

Dopo il livello di pooling viene solitamente utilizzato un livello di attivazione che fa uso della funzione ReLu. Viene abbandonato il modello tramite sigmoide perché problematico nella back propagation. Infatti la derivata della sigmoide è minore di 1. Dunque molti parametri tendono a essere moltiplicati per valori minori di uno che tendono a far svanire il gradiente. Ancora peggio in quelle regioni in cui la derivata è zero e dunque il gradiente tende a svanire istantaneamente. La funzione ReLu è invece definita come

$$f(u) = max(0, u) \tag{7}$$

La funzione ReLu, rappresentata in Figura 20 , possiede derivata pari a zero per valori negativi e pari a uno per tutti gli altri.

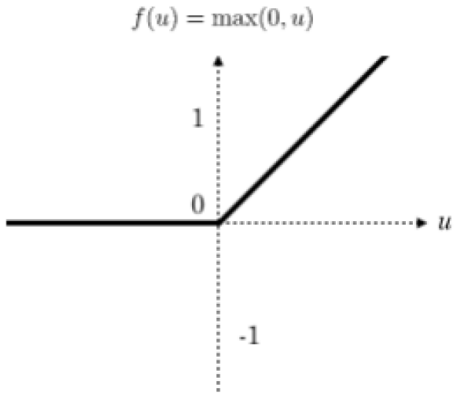


Figura 20: Funzione di attivazione ReLu

Questo ci permette di avere un'insieme di neuroni che si "spengono" quando il loro valore è negativo, o che sono completamente attivi quando il loro valore è positivo.

Solitamente in una rete neurale convolutiva abbiamo un insieme di strati ognuno composto da: un livello convolutivo, uno di pooling e uno di attivazione. Dopo questi strati abbiamo un ridotto numero di livelli completamente connessi. La funzione di attivazione dei neuroni degli ultimi livelli (quelli completamente connessi) è

$$z_k = f(net_k) = \frac{e^{net_k}}{\sum_{c=1,\ldots,s} e^{net_c}} \tag{8}$$

Una visione intuitiva per questa equazione è quella della *normalizzazione*. Il valore di e^{net_k} viene diviso per la somma di tutti gli altri, quindi possiamo vedere $f(net_k)$ come una probabilità.

Per valutare l'errore commesso in output utilizziamo invece una funzione di *entropia* molto nota nell'ambito dell'apprendimento automatico

$$H(p, q) = -\sum_{v} p(v) log(q(v)) \tag{9}$$

dove $p(v)$ è la funzione da determinare mentre $q(v)$ è quella determinata da noi.

Mostriamo in Figura 21 una possibile architettura convolutiva complessiva.

Nonostante tutti questi accorgimenti la rete mostrata in Figura 21 possiede $60 \cdot 10^6$ parametri da determinare. Come farlo? Solitamente vengono utilizzati dei tool come Caffe, Theano, Torch, TensorFlow, Digits.

Una domanda che potrebbe essere naturale porsi è: visto che l'addestramento richiede così tanto tempo e produce un elevato numero di parametri, potremmo pensare di riutilizzarli per problemi simili? Esistono due tecniche a riguardo:

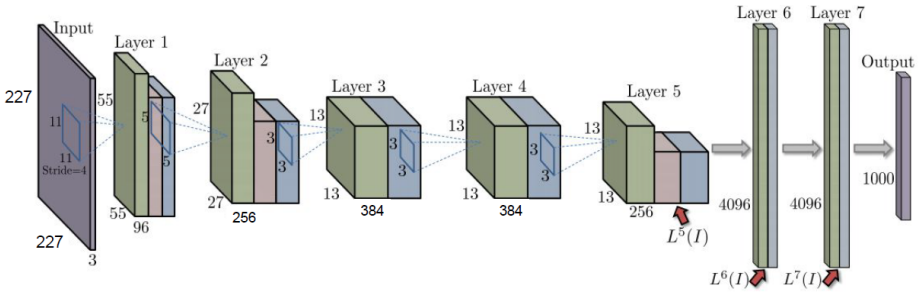


Figura 21: Struttura complessiva di un'architettura convolutiva. Sono indicati in verde i livelli convolutivi, in rosa quelli di pooling e in blu quelli di attivazione. Gli ultimi due livelli sono di dimensione maggiore poiché completamente connessi

- Fine tuning: rimpiazziamo il livello di output con un nuovo strato. Impostiamo come valori iniziali dei pesi quelli della rete pre allenata, ad eccezione per le connessioni tra il penultimo e l'ultimo livello, infine effettuiamo diverse iterazioni per ottimizzare i pesi rispetto alle particolarità del nuovo dataset
- Riutilizzo di features: usiamo le feature prodotte da un livello intermedio e le usiamo in un classificatore esterno come la SVM.

07 / 11 /2017

5.2 Tecniche avanzate di deep learning

Un fatto a cui avevamo accennato è che il deep learning esiste già da parecchio tempo ma solo recentemente ha avuto i maggiori sviluppi. Il fatto che prima non si potessero praticare tecniche di deep learning è dovuto ai due motivi seguenti

- Dati: non vi erano a disposizione la mole di dati sufficiente per addestrare i modelli
- Hardware: non esisteva ancora un hardware sufficientemente potente da analizzare i dati

I modelli di deep learning -e alcune loro applicazioni-possono essere

- Uno a uno: i modelli più semplici, detti *vanilla neural networks*, solitamente usate per problemi di classificazione
- Uno a molti: che possono essere utilizzate per etichettare immagini
- Molti a uno: che possono essere usate per la *classificazione di sentimenti*, ovvero data una serie di parole individuare un sentimento associato

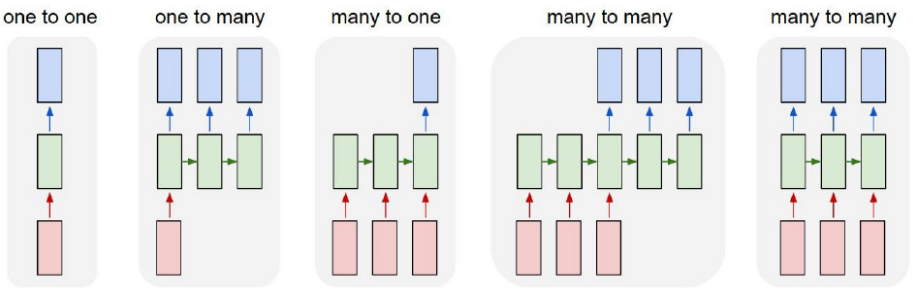


Figura 22: Possibili modelli di deep learning

- Molti a molti: che possono essere usate per la classificazione video

Mostriamo in Figura 22 la rappresentazione dei modelli appena elencati

Come già detto in precedenza tutti questi modelli possono essere descritti da una rete single layer, ma a causa di problemi di stabilità e tempi di addestramento si preferisce utilizzare uno schema multilayer. Notiamo che un aspetto importante in questo procedimento sia la condivisione dei parametri.
A un livello più avanzato rispetto alle normali reti neurali, non basta più l'algoritmo di backpropagation, ma dobbiamo utilizzare un algoritmo più complicato detto *backpropagation through time*. Per arrivare a comprenderlo dobbiamo partire dal *grafo di computazione* che intende rappresentare un intero procedimento di computazione. Questo grafo viene solitamente utilizzato nel procedimento di *unfolding* in cui cerchiamo di "srotolare" la computazione nel tempo. Tutto questo vorrebbe cercare di comprendere come i dati si influenzino a vicenda con lo scorrere del tempo. Un esempio a riguardo è quello della computazione dinamica: per conoscere lo stato di un certo sistema al tempo t di solito dobbiamo sapere qualcosa relativamente al tempo $t-1$. Dunque:

$$s(t) = f(s(t-1), \theta) \tag{10}$$

dunque lo stato al tempo t dipende dallo stato al tempo precedente e da un insieme di parametri θ.
Ad ogni modo una rappresentazione effettiva dell'unfold è quella mostrata in Figura 23.

Più formalmente la ricorrenza ottenuta dal procedimento di unfolding può essere schematizzata come segue:

$$h(t) = g(t)(x(t), x(t-1), \ldots, x(2), x(1)) = f(h(t-1), x(t), \theta) \tag{11}$$

Ovviamente l'intero procedimento di passaggio da uno stato all'altro può essere visto sotto forma matriciale come segue

$$h(t) = g(Uh(t-1) + Vx(t)) \tag{12}$$

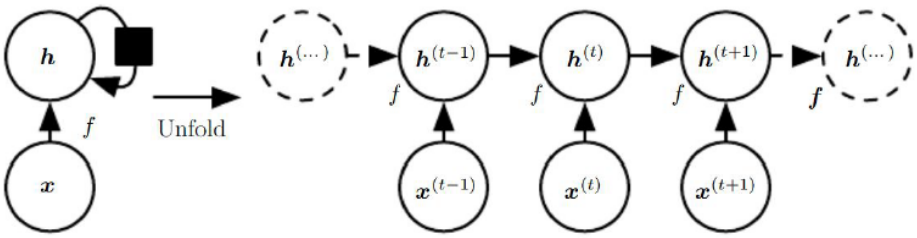


Figura 23: A sinistra abbiamo una rete neurale ricorrente mentre a destra abbiamo la stessa rete ottenuta mediante il procedimento di unfolding

Che fine hanno fatto i parametri θ? Sono inclusi nelle matrici U, V.
Dunque l'idea dell'unfolding si basa su due princìpi

- Indipendentemente dalla lunghezza dell'input, ci aspettiamo che abbia sempre la stessa forma
- È possibile usare in tal caso una stessa funzione f con gli stessi parametri ad ogni passo

Solitamente in base a questo procedimento otteniamo i tre seguenti risultati:

$h(t) = tanh(Wh(t-1) + Ux(t) + b)$
$o(t) = Vh(t) + c$
$\hat{y}(t) = softmax(o(t))$

dove b e c sono parametri di bias mentre U, V e W rappresentano rispettivamente le matrici dei pesi relative a input a livello nascosto, livello nascosto ad output e infine livello nascosto a livello nascosto.
Vediamo un esempio per comprendere meglio

Esempio 1 : supponiamo di voler riconoscere una certa parola *hello* dato il seguente dizionario $\{h, e, l, o\}$. Supponiamo di avere a nostra disposizione le matrici dei pesi U, V e W. Ovviamente per determinare l'output dati i fonemi della parola dobbiamo fare uso delle due formule viste poco fa

$h(t) = tanh(Wh(t-1) + Ux(t) + b)$
$o(t) = Vh(t) + c$

L'input che riceviamo è una schematizzazione di ognuno dei fonemi. In questo caso la rappresentazione fa sì che l'esempio non sia molto intelligente visto che partiremo dalla rappresentazione di una lettera sotto forma di vettore per arrivare ad una rappresentazione identica. In realtà al posto dell'input a quattro componenti avremmo potuto benissimo usare un'altra rappresentazione. Ad esempio avremmo potuto utilizzare la trasformata

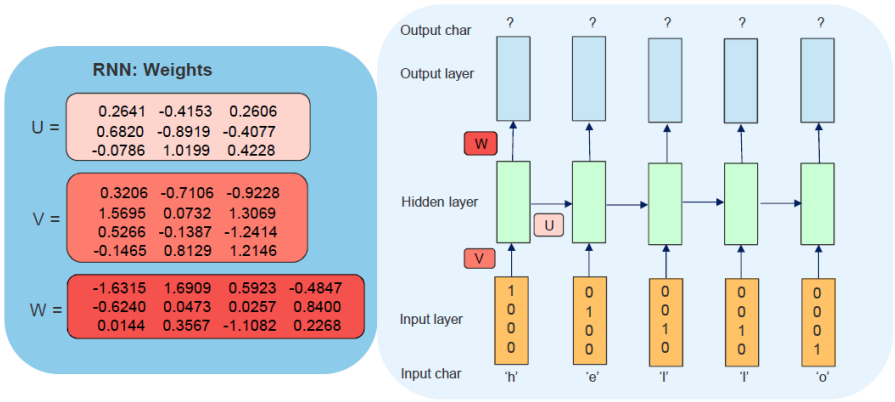


Figura 24: Situazione iniziale dell'esempio 1

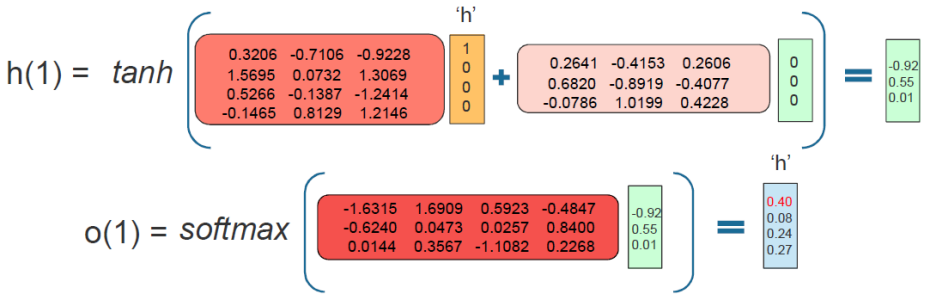


Figura 25: Moltiplicazioni tra matrici per ottenere $h(1)$ e $o(1)$

di Fourier del fonema.
Ad ogni modo la situazione complessiva iniziale è quella mostrata in Figura 24

Il calcolo che dobbiamo effettuare per ottenere $h(1)$ e $o(1)$ viene mostrato in Figura 25

Eseguendo il procedimento un certo numero di volte otteniamo un output come quello mostrato in Figura 26

A questo punto possiamo parlare del procedimento di backpropagation through time. In questo procedimento l'algoritmo di discesa del gradiente cambia come segue:

$$\Delta w_{ij} = -\eta \frac{\partial L_{tot}(t_0 t_1)}{\partial w_{ij}} = -\eta \sum_{t=t_0}^{t_1} \frac{\partial L(t)}{\partial w_{ij}} \tag{13}$$

dove L_{tot} rappresenta la Loss totale utilizzando un metodo di entropia o simili. Cosa è cambiato rispetto al normale algoritmo di backpropagation? Che abbia-

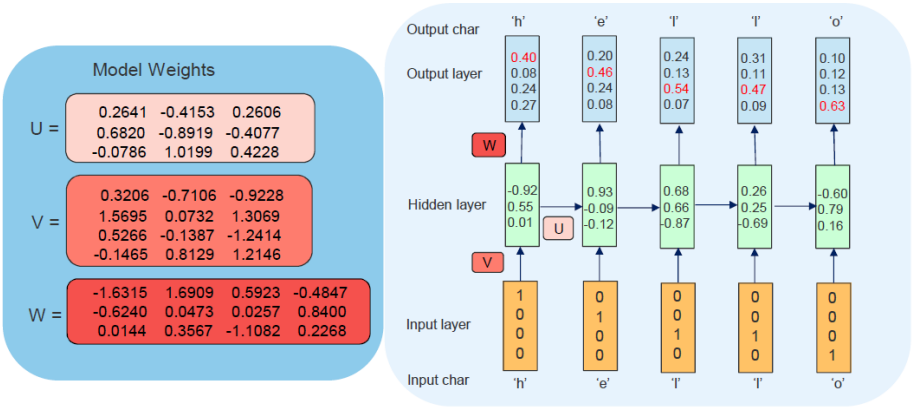


Figura 26: Risultati finali dell'esempio 1. Ottenuti dopo aver calcolato $h(5)$ e $o(5)$

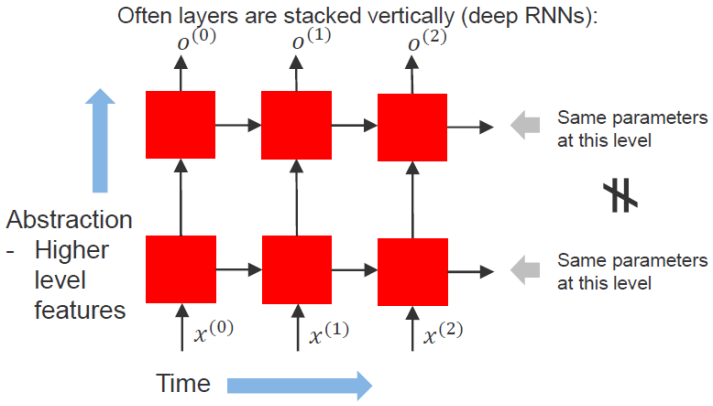


Figura 27: Partizionamento orizzontale e verticale della rete neurale

mo introdotto una sommatoria sul tempo. Tale sommatoria esplica la relazione nel tempo fra i dati.
In una rete allenata in questo modo abbiamo diversi livelli sia verticali che orizzontali. Questa situazione è mostrata in Figura 27

Notiamo che esiste un partizionamento orizzontale che rappresenta le feature sempre più complesse, ma allo stesso tempo esiste anche un partizionamento verticale che indica lo scorrere del tempo.

L'ultimo problema davvero importante da affrontare è il già noto problema del *Vanishing Gradient*. Questa è la situazione in cui un gradiente, a lungo andare o tende a zero o tende a un valore molto grande. Questo è dovuto al fatto che spesso le derivate in questo contesto danno valori minori di uno e dunque

andando ad effettuare n prodotti minori di uno il valore complessivo tende a zero. Viceversa se moltiplichiamo molti valori maggiori di uno, il valore finale tenderà ad essere molto grande.
Questo è un problema se vogliamo andare ad analizzare sequenze molto lunghe. Una soluzione tipica è quella di spezzare il procedimento di unfolding andando ad aggiornare i pesi ogni n passi. In questo modo la minimizzazione o l'esplosione del gradiente è limitata.
Un'altra soluzione è quella del *gradient clipping* in cui preveniamo l'esplosione del gradiente andando a normalizzarlo di modo che la norma dello stesso sia pari al più ad un particolare valore ϵ.
Un ulteriore tecnica fa uso delle *Long Short Term Memory*(LSTM) in cui una complessa entità software cerca di valutare quando sia il caso di tralasciare elementi passati troppo lontani nel tempo.
Esistono anche altri modelli di rete per trattare questo problema: le *reti bidirezionali* e le *Gated Recurrent Units*(GRU).

In conclusione questi modelli avanzati possono comprendere delle dipendenze nel tempo fra i dati. Sono molto flessibili e questo permette di applicarle in moltissimi campi. D'altro canto molti studi sono in corso, in particolare non tanto per ottenere modelli più complessi, quanto per rendere più efficienti i modelli più semplici.

14 / 11 /2017

6 Self-Organizing maps

Le *self-organising maps*(SOM) sono un particolare tipo di rete neurale pensato per l'apprendimento non supervisionato. Cerchiamo di fare una panoramica scendendo successivamente nel dettaglio.
Esistono due tipi fondamentali di SOM:

- Monodimensionali: hanno una struttura monodimensionale come quella mostrata nella parte sinistra della Figura 28
- Bidimensionali: la struttura è a griglia come quella mostrata nella parte destra della Figura 28

Un aspetto importante da notare è che i neuroni all'interno delle SOM sono esattamente gli stessi dei modelli che abbiamo visto in precedenza. Dunque basati su un sistema di pesi.
Intuitivamente cosa va ad apprendere la rete? L'idea è che per input in zone vicine della rete, l'output dovrà essere simile. Questo tipo di rete è particolarmente apprezzata dagli psicologi cognitivi perché il cervello si comporta in modo simile. Ovvero zone spazialmente vicine del cervello tendono a processare informazioni simili.
Quale input possiamo fornire alla SOM? Di fatto tutto quello che sia schematizzabile tramite un vettore. Vediamo un esempio

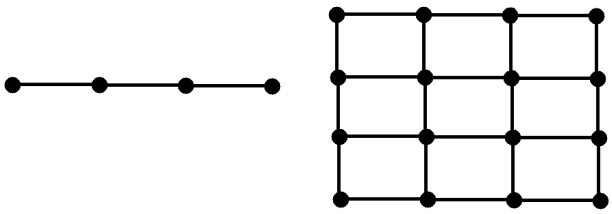


Figura 28: A sinistra una SOM monodimensionale. A destra una SOM bidimensionale

Esempio 1 : supponiamo di voler classificare un insieme di animali. Potremmo andare a costruire un vettore che possieda varie caratteristiche: la lunghezza, la lunghezza delle gambe, il peso...

Dopo aver schematizzato un dataset sotto forma di vettore, possiamo fornire un esempio alla rete e vedere quale neurone si *attiva di più*(maggiori dettagli su questo nel seguito). Stabilito quale sia il neurone più attivo, andiamo a modificare i suoi pesi e quelli dei neuroni vicini. Dopo aver allenato la rete possiamo effettuare una classificazione dei nuovi esempi.

Cerchiamo adesso di fornire qualche dettaglio ulteriore.
Come selezionare il neurone più attivo in relazione ad un certo input? Prendiamo quello che massimizza il prodotto scalare tra vettore dei pesi del neurone i-esimo $W_i = (w_1, \ldots, w_n)$ e l'input $x = (x_1, \ldots, x_n)$come segue:

$$W_i^T x = w_1 x_1 + \ldots + w_n x_n \tag{14}$$

Questo calcolo viene fatto per ogni neurone. Di fatto il prodotto scalare tra due vettori cresce quanto più i vettori sono simili fra loro. In particolare raggiunge il massimo se sono coincidenti. Dunque a livello implementativo il neurone da selezionare è quello che minimizza la distanza euclidea tra l'input X e il vettore dei pesi W:

$$i : \forall j \parallel x - w_i \parallel \leq \parallel x - w_j \parallel \tag{15}$$

Questa fondamentale equazione determina il miglior neurone, ovvero la *best matching unit*(BMU).

Una volta selezionato il neurone, come possiamo aggiornare il peso del neurone? Solitamente viene usata una formula simile ad una vista in precedenza per l'apprendimento supervisionato. La modifica dei pesi per il neurone i-esimo segue la seguente equazione

$$W_i(n+1) = W_i(n) + \eta(n)(x - W_i(n)) \tag{16}$$

Notiamo che il learning rate può variare a seconda dell'epoca. In particolare tende a decrescere nel tempo secondo la seguente legge:

$\eta(n) = \eta_0 \cdot e^{-\frac{n}{\tau}}$.
Cerchiamo conferma che la rete si adatta all'input con il seguente esempio

Esempio 2 : supponiamo di avere un dato di input della forma $x = [1, 0.8]$. Supponiamo che $w_i(n) = [0.2, 1]$ e che $\eta(n) = 0.5$. Dalla formula di aggiornamento dei pesi abbiamo che:

$$W_i(n+1) = [0.2, 1] + 0.5([1, 0.8] - [0.2, 1]) = [0.6, 0.9]$$

Notiamo che il vettore dei pesi dal passo n al passo $n+1$ si è avvicinato al vettore di input.
Notiamo inoltre che se poniamo $\eta(n) = 1$ allora w_i diventa pari a x

Come dicevamo in precedenza, non solo il best matching unit viene aggiornato, ma anche tutti i suoi vicini. La formula che viene utilizzata per aggiornarlo è del tutto simile a quella dell'aggiornamento del neurone, con l'aggiunta di un fattore moltiplicativo che tenga di conto della distanza dal neurone modificato. Ovvero:

$$W_j(n+1) = W_j(n) + \eta(n) h_{j,i}(n)(x - W_j(n)) \tag{17}$$

Solitamente il fattore di distanza $h_{j,i}(n)$ è definito da una funzione gaussiana. Ovvero vale che $h_{j,i}(n) = e^{-\frac{d_{j,i}^2}{2\sigma(n)^2}}$ e $\sigma(n) = \sigma_0 e^{-\frac{n}{\tau}}$ con τ costante.
Inoltre notiamo che anche $h_{j,i}$ è in funzione di n. Infatti dopo un certo periodo di tempo riduciamo l'apprendimento dei vicini.
Dunque un aspetto importante da considerare è che neuroni vicini tra loro assumono pesi simili e risponderanno in modo simile allo stesso input.
Solitamente nell'apprendimento abbiamo due fasi:

1. Una prima fase di *auto organizzazione* della durata di mille epoche, in cui il fattore di apprendimento η decresce da 0.1 a 0.01. In questa fase nelle prime epoche una correzione dei pesi si propaga a tutti i vicini. Verso la fine di questa fase una modifica dei pesi intacca solo il neurone di best matching

2. Una seconda fase di *convergenza* della durata di cinquecento epoche in cui viene ridotto ulteriormente il fattore η e la funzione $h_{j,i}$ va a modificare solo il neurone di best matching

Complessivamente l'algoritmo si articola nelle seguenti fasi:

1. Inizializzare i pesi dei neuroni ad un valore casuale

2. Prendere un input

3. Trovare il neurone best matching

4. Aggiornare i pesi del neurone di best matching e dei vicini

5. Ripetere fino a quando cessiamo di avere cambiamenti significativi

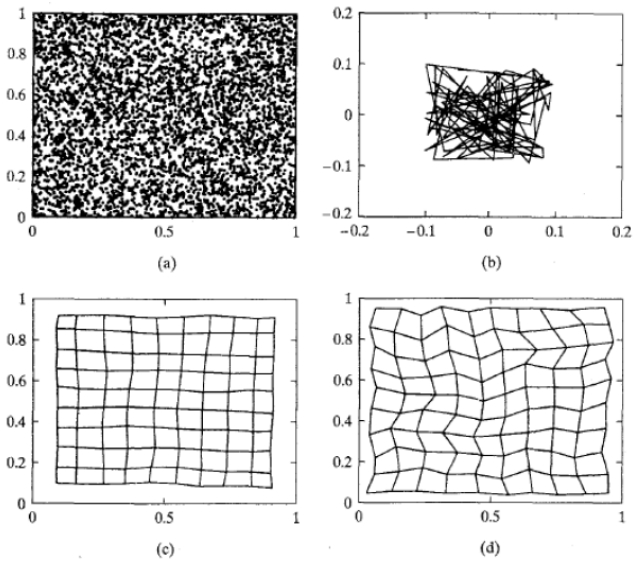


Figura 29: a) dataset iniziale b) condizioni iniziali dopo aver assegnato i pesi in modo casuale c) condizioni del reticolo dopo la fase di auto organizzazione d) condizioni del reticolo alla fine della fase di convergenza

Quando l'algoritmo termina restituisce una rappresentazione succinta dello spazio di input. A titolo d'esempio mostriamo in Figura 29 l'esecuzione dell'algoritmo di apprendimento

21 / 11 /2017

7 Reti di Hopfield

Sviluppate agli inizi degli anni '80, possono cercare di ricostruire un insieme di dati partendo da delle *memorie fondamentali* e da un dato parzialmente corrotto. Dunque vengono solitamente utilizzate per quei compiti di *pattern completion*. Questa è una caratteristica importante utilizzata spesso anche dal cervello umano. Ad esempio se scorgiamo parte di un auto che sporge da un muro, possiamo -entro certi limiti- prevedere quali forme ci aspettano dietro al muro stesso.
Solitamente quindi, forniremo ad una rete di Hopfield addestrata un certo input, e in seguito la rete convergerà al risultato "più simile" contenuto nelle sue memorie. Questo succede quando lo stato raggiunto è uno *stato stabile*.

Andiamo a descrivere queste reti. Possiedono un certo numero di neuroni come quelli che già conosciamo e tali neuroni sono collegati, ma mai con loro stessi. Inoltre in queste reti il neurone può assumere due soli stati: $+1$ oppure -1.
Come viene determinato lo stato di un neurone? Viene fatto analizzando il

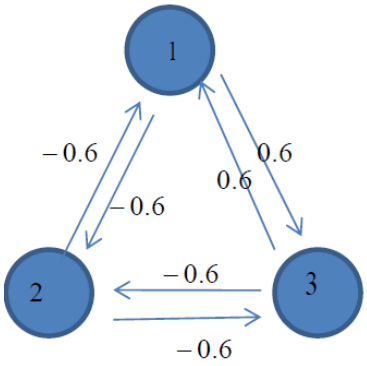


Figura 30: Rete di Hopfield dell'esempio 1

neurone stesso e i risultati che riceve dai neuroni vicini(pesati sul valore del collegamento). Di fatto, il neurone j assume valore v_j secondo la seguente uguaglianza:

$$v_j(n) = \sum_{i=0}^{N} w_{ji} y_i(n) \tag{18}$$

Dove $y_j(n) = \varphi(v_j(n))$ e

$$\varphi(v_j(n)) = \begin{cases} 1, & \text{se } v_j(n) > 0 \\ -1, & \text{se } v_j(n) < 0 \\ \varphi(v_j(n-1)), & \text{se } v_j(n) = 0 \end{cases} \tag{19}$$

Quindi stiamo dicendo che l'output di un certo neurone dipende dal suo stato interno. Vediamo un esempio per comprendere meglio

Esempio 1 : consideriamo la rete di Hopfield mostrata in Figura 30

Supponiamo che lo stato iniziale sia $[v_1, v_2, v_3] = [-1, -1, 1]$
Possiamo chiederci se è uno stato stabile. Per verificarlo calcoliamo il valore del neurone 1 in base all'equazione vista poco fa:

$v_1 = \sum_{i=1}^{3} w_{1i} y_i(n) = 0.6 + 0.6 = 1.2$

In base a questo, la funzione φ ci dice che, essendo $v_1 > 0$ allora lo stato del neurone v_1 deve essere pari a 1. Dunque il vettore dello stato passa da essere $[-1, -1, 1]$ a essere $[1, -1, 1]$.
Andando a calcolare v_2 otteniamo

$v_2 = -0.6 - 0.6 = -1.2$

Dunque, secondo la funzione φ lo stato del neurone v_2 deve essere negativo. Visto che già lo è, allora lo stato della rete resta invariato. Andando

a calcolare v_3 otteniamo un risultato positivo, dunque non cambiamo lo stato della rete. Nel complesso $[1, -1, 1]$ è uno stato stabile della rete.

Notiamo che se avessimo analizzato i neuroni in ordine diverso, avremmo avuto una convergenza ad un altro stato stabile della rete, ovvero $[-1, 1, -1]$

Dall'esempio precedente abbiamo imparato che possono esistere più stati stabili a cui convergere. La probabilità di convergere al giusto stato stabile dipende da quanto sia "corrotto" il dato in input. Nel seguito vedremo anche che iterando l'analisi dei neuroni convergiamo sempre ad uno degli stati stabili.

Il problema che abbiamo è però il seguente: come fare affinché l'informazione che vogliamo memorizzare, ovvero le nostre memorie fondamentali, coincidano con degli stati stabili?
Intuitivamente, andando a considerare i neuroni due a due, questi daranno uno stato stabile se

- Lo stato dei due neuroni è di segno concorde e sono collegati da un collegamento a peso positivo
- Lo stato dei due neuroni è di segno discorde e sono collegati da un collegamento a peso negativo

Quindi quello che potremmo fare è di settare adeguatamente i pesi affinché i dati da memorizzare siano stati stabili. Seguiamo quindi il *principio di Hebb* con cui rafforziamo le connessioni tra unità aventi stessa attivazione e indeboliamo le connessioni tra unità con attivazione opposta. Questo principio possiede una controparte biologica, infatti nel cervello tendiamo a rafforzare le sinapsi tra unità che spesso sono attive allo stesso tempo ed indeboliamo le sinapsi tra i neuroni che non sono attivi simultaneamente.

La formula con cui in generale andiamo a stabilire i pesi della rete è la seguente

$$w_{ji} = \frac{1}{M} \sum_{k=1}^{M} f_k(i) f_k(j) \tag{20}$$

dove $f_k(i)$ è il valore che assume la k-esima memoria fondamentale sulla componente i. Notiamo che $j \neq i$ e dove M è la cardinalità delle memorie fondamentali. Applichiamo la formula ad un esempio:

Esempio 2 : supponiamo di avere a disposizione le seguenti memorie fondamentali:

$$[1, -1, 1] \qquad\qquad [-1, 1, -1]$$

Notiamo che in entrambe le memorie: la prima componente e la seconda sono discordi, la seconda e la terza discordi, la terza e la prima concordi.

Intuitivamente ci aspettiamo che $w_{12} = w_{21} = -1$, che $w_{23} = w_{32} = -1$ e infine che $w_{13} = w_{31} = 1$. Verifichiamolo:

$$w_{13} = \frac{1}{2}\sum_{k=1}^{2} f_k(3)f_k(1) = \frac{1}{2}(1 \cdot 1 + (-1) \cdot (-1)) = 1$$
$$w_{12} = \frac{1}{2}((-1) \cdot 1 + 1 \cdot (-1)) = -1$$
$$w_{23} = \frac{1}{2}(1 \cdot (-1)(-1) \cdot 1) = -1$$

Un aspetto teorico importante a cui avevamo accennato in precedenza è che esiste un teorema relativo alla convergenza

Teorema 7.1. *Dato uno stato qualunque di una rete di Hopfield, la rete converge ad uno stato stabile*

Non vediamo per intero la dimostrazione, ma cerchiamo di darne un'intuizione. Supponiamo di avere una rete ad N stati. Allora esistono 2^N possibili stati della rete. Ad ognuno di questi stati possiamo associare un valore di *energia*. È possibile dimostrare che ogni cambiamento della rete che segua le regole stabilite nelle pagine precedenti, porta ad un abbassamento dell'energia. Esiste poi un momento in cui nulla cambia più e non possiamo muoverci in altri stati. Possiamo definire la funzione di energia come segue:

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

Notiamo che, in ogni prodotto $w_{ij}y_iy_j$, se y_i e y_j hanno stesso segno e w_{ij} è positivo, oppure se sono di segno opposto e w_{ij} è negativo allora abbiamo una diminuzione dell'energia.
Di fatto tanto più bassa è l'energia, quanto maggiore è l'armonia tra lo stato attuale e quello delle memorie fondamentali.
Andiamo a considerare i due stati E ed E', dove E' è lo stato che otteniamo dopo avere eseguito un passo. Consideriamo allora la quantità $E - E' = -\sum_j w_{kj}y_j(y_k - y'_k)$. Possiamo distinguere due casi

1. y_k passa da $+1$ a -1. Dunque $y_k - y'_k > 0$ e $\sum_j w_{kj}y_j < 0$ allora:

   $-\sum_j w_{kj}y_j(y_k - y'_k) > 0$, ovvero $E > E'$

2. y_k passa da -1 a $+1$. Dunque $y_k - y'_k < 0$ e $\sum_j w_{kj}y_j > 0$ allora:

   $-\sum_j w_{kj}y_j(y_k - y'_k) > 0$, ovvero $E > E'$

Dunque ad ogni passo l'energia decresce.

In conclusione quali sono le qualità delle reti di Hopfield? Ci permettono di completare pattern parziali, di generalizzare dando risposte coerenti tra loro quando ricevono input simili tra loro. Sono tolleranti ai guasti(dati corrotti), e permettono di estrarre dei prototipi quando ricevono diverse informazioni da

Figura 31: A sinistra due minimi locali spazialmente vicini, a destra il nuovo minimo nella locazione intermedia

cui possono apprendere.
I principali problemi delle reti di Hopfield sono che non tutti gli stati stabili corrispondono necessariamente ad una memoria fondamentale. Inoltre possiedono una limitata capacità di memorizzare.

28 / 11 /2017

8 Reti di Boltzmann

8.1 Problemi nelle reti di Hopfield

Nel precedente capitolo abbiamo concluso andando ad analizzare i problemi delle reti di Hopfield.
Tali problemi possono possono essere riassunti come segue: nelle reti di Hopfield la speranza è che ad ogni memorizzazione si venga a creare un nuovo minimo energetico da far coincidere con la memoria fondamentale. Nel momento in cui due esempi vengono associati a due minimi spazialmente vicini tra loro, è possibile che tali minimi si fondano tra loro in una locazione intermedia. Questa situazione viene mostrata in Figura 31.

Questa situazione è problematica perché la rete tenderà a raggiungere tale minimo intermedio che però rappresenta una ricostruzione intermedia delle due memorie fondamentali!
Per evitare che si verifichino situazioni come quella appena descritta, solitamente si riduce il numero di memorie fondamentali che andiamo ad introdurre nella rete, o alternativamente aumentiamo il numero di neuroni. Il problema è che non possiamo fare un buon uso della capacità di memorizzazione della rete. Infatti in presenza di N neuroni, è consigliabile di far memorizzare alla rete soltanto $0.15N$ memorie fondamentali.
Una soluzione proposta da Hopfield e da altri consiste nel effettuare l'apprendimento partendo da uno stato iniziale casuale e successivamente andare a disimparare relativamente ai minimi spuri. Questa tecnica funziona e ha ispirazione biologica, come sostenuto dal premio Nobel Crick. Infatti quando sogniamo andiamo a dimenticare parte di quanto abbiamo appreso durante la giornata. Il problema di questo approccio è che non esiste una teoria ben strutturata che ci dica "quanto" la rete debba dimenticare.
Un'altra soluzione è stata proposta da Elizabeth Gardiner la quale è riuscita a

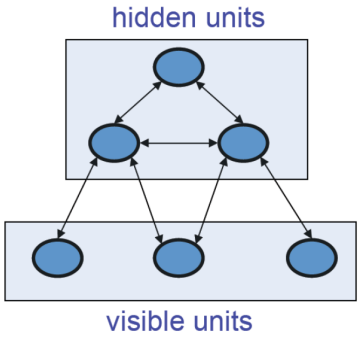


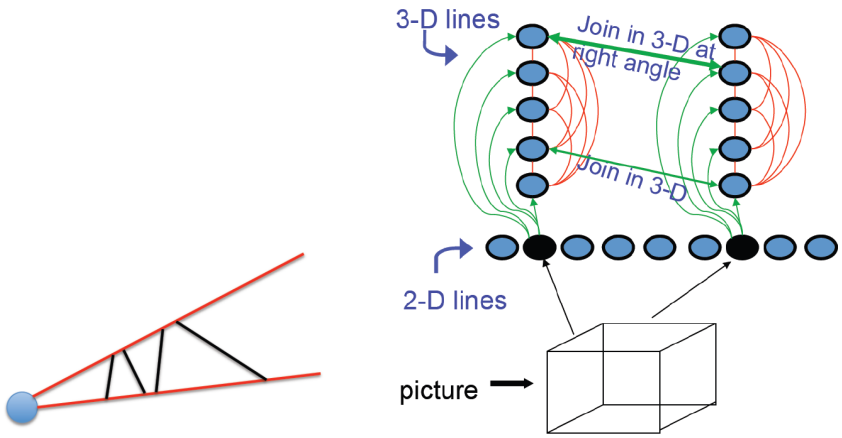
Figura 32: Una rete di Boltzmann

sfruttare completamente la capacità di memorizzazione della rete. L'intuizione dietro a tale lavoro è che, invece di salvare le memorie in un colpo solo, possiamo tornare ciclicamente su ognuna di esse per effettuarne una migliore memorizzazione.

Un approccio diverso è quello di utilizzare delle reti di Boltzmann. In tali reti abbiamo due unità nuove: *unità nascoste* e *unità visibili*.
Mostriamo un esempio di rete di Boltzmann in Figura 32

Le unità visibili rappresentano l'input per la rete.
Le unità nascoste costituiscono una rete di Hopfield che serve per costruire interpretazione dell'input che vogliamo che la rete memorizzi. La qualità dell'interpretazione appresa è legata al valore di energia della rete per quell'interpretazione.
Hinton ha fornito una rappresentazione intuitiva di quello che succede all'interno di una rete di Boltzmann. Per farlo consideriamo dapprima la Figura 33(a).

Quando l'occhio umano percepisce il mondo che lo circonda, va a costruire un'interpretazione 2D di uno spazio 3D. Infatti quello che va perso è la terza dimensione, ovvero la profondità. Per questo motivo può succedere che diverse linee, se sovrapposte fra loro potrebbero essere percepite come una linea unica dal nostro occhio.
Alla luce di questo consideriamo la Figura 33(b). Supponiamo di avere un cubo che se osservato abbastanza a lungo può avere due interpretazioni(la faccia "frontale" può puntare verso il basso o verso l'alto). Ovviamente a queste due interpretazioni corrisponderanno due minimi locali.
Osservando il cubo, possiamo vederne in contemporanea al più 9 spigoli da noi percepiti in 2D. L'idea è allora quella di costruire una rete di Boltzmann in cui abbiamo un neurone visibile per ogni spigolo 2D visibile. Poi, visto che ogni spigolo 2D potrebbe essere la rappresentazione di molti spigoli 3D possibili, allora per ognuno di questi spigoli 2D andiamo ad associargli un numero di neuroni pari al numero di interpretazioni 3D di quello stesso spigolo. Questo secondo livello sarà quello dei neuroni hidden che costituiscono l'interpretazione



(a) Linee 3D che vengono percepite come 2D da un occhio umano

(b) Cubo interpretabile da due punti di vista, e la corrispondente rete neurale

Figura 33: Rappresentazione intuitiva di una rete di Boltzmann

della realtà da noi percepita. Ovvero stiamo dicendo che i neuroni visibili sono legati a quello che noi percepiamo del cubo(spigoli 2D), i neuroni hidden sono invece legati alla "vera" natura del cubo(spigoli 3D).
Andremo poi ad aggiungere dei collegamenti tra i neuroni hidden ogniqualvolta si congiungano all'interno dell'immagine da noi percepita. Questo perché se due spigoli si congiungono hanno un legame che deve essere rappresentato in qualche modo. In particolare tale legame sarà ancor più forte se i due spigoli si uniscono formando un angolo retto. Questo perché probabilmente costituiranno un punto particolarmente importante per il cubo(un angolo del volume ad esempio).

8.2 Tempra simulata

Un altro problema che avevamo nelle reti di Hopfield è che la rete prende sempre la decisione che riduca il più possibile l'energia. Questo comporta una convergenza verso un minimo che potrebbe essere locale. Difficilmente la rete si sposterà da un minimo locale per andare a ricercare un minimo globale.
Solitamente preferiamo utilizzare delle reti "rumorose" di modo che sia possibile passare da un minimo locale al minimo successivo. Questo meccanismo viene detto *tempra simulata*. Possiede questo nome perché si basa su un concetto di "temperatura". Con questo meccanismo andiamo a sostituire il neurone binario basato su una "soglia" del modello di Hopfield con un neurone binario stocastico. In questo caso, indicando con $s_i = 1$ il fatto che il neurone i-esimo sia acceso, diremo che questo evento si verifica con la seguente probabilità:

$$p(s_i = 1) = \frac{1}{1 + e^{-\Delta E_i / T}} \tag{21}$$

dove T è per l'appunto la temperatura. E dove:

$\Delta E_i = E(s_i = 0) - E(s_i = 1) = b_i + \sum_j s_j w_{ij}$

Solitamente all'inizio del procedimento di apprendimento la temperatura è molto alta e questo conduce ad avere una $p(s_i = 1) \simeq 0.5$.
Procedendo nell'addestramento questa temperatura viene progressivamente abbassata e dunque $p(s_i = 1)$ sarà pari a 0 o a 1 a seconda del segno di ΔE.
Notiamo inoltre che per $T \to 0$ ci riconduciamo al modello di Hopfield!
Vediamo con un esempio come mai il modello di tempra simulata ci permetta talvolta di allontanarci dai minimi locali alla ricerca di altri minimi -sperabilmente- migliori.

Esempio 1 : calcolare $p(s_i = 1)$ nei seguenti casi

- $E(s_i = 0) = 3$ mentre $E(s_i = 1) = 4$ da cui segue che $\Delta E_i = -1$ e dunque
  $p(s_i = 1) = \frac{1}{1 + e^1} = 0.27$ e dunque l'energia peggiora poiché
  $p(s_i = 1) < 0.5$
  L'aspetto importante da notare in questo caso è che sebbene l'energia del sistema aumenti (passa da 3 a 4) esiste comunque una probabilità che il neurone si accenda! Certo non è molto probabile percorrere tale via in quanto la probabilità è pari a 0.27. Ma ad ogni modo potrebbe succedere ed è per questo che il modello del neurone viene detto stocastico. Infatti non abbiamo una diminuzione deterministica dell'energia. Ma solo una diminuzione probabile.
- $E(s_i = 0) = 4$ mentre $E(s_i = 1) = 3$ da cui $p(s_i = 1) = 0.73$ e dunque l'energia migliora poiché $p(s_i = 1) > 0.5$
  In questo caso possiamo notare che il passaggio da uno stato ad energia maggiore ad uno con energia minore è favorito dalla probabilità. Ma anche stavolta tale diminuzione non è certa, è solo probabile!
- $E(s_i = 0) = 3$ mentre $E(s_i = 1) = 3$ da cui $p(s_i = 1) = 0.5$ e dunque l'energia rimane invariata poiché $p(s_i = 1) = 0.5$
  In questo caso possiamo notare che se da uno stato al successivo l'energia rimane invariata, allora il neurone si accende o si spegne in modo prettamente casuale, ovvero con probabilità 0.5

Esempio 2 calcolare $p(h_1 = 1)$ nel caso in cui la rete possieda due neuroni visibili e uno nascosto, come quella mostrata in Figura 34.

In questo caso abbiamo che:
$\Delta E = b_i + \sum_j s_j w_{ij} = 0 + (1 \cdot 0.01 + 0 \cdot 0.02) = 0.01$
Dunque:
$p(h_1 = 1) = \frac{1}{1 + e^{-0.01}} = 0.5025$

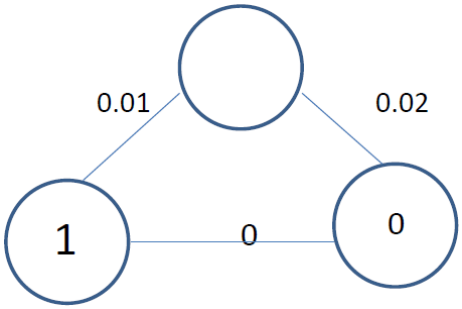


Figura 34: Rete dell'esempio 2

Complessivamente l'idea della tempra simulata è un metodo molto potente per ricercare un ottimo globale ma in questo contesto ci distrae dallo scopo del corso. Dunque da ora in poi utilizzeremo delle unità binarie stocastiche che hanno una temperatura pari a 1.
Ricordiamoci che lo scopo dell'apprendimento nelle reti di Boltzmann è quello di massimizzare le probabilità da assegnare ai vettori binari del training set. Questo perché se la rete di Boltzmann si adatta per riconoscere con massima probabilità i vettori del training set, riuscirà con buona probabilità a riconoscere anche gli input di test successivi.
Ma dunque massimizzare la probabilità per un insieme di vettori è equivalente a voler massimizzare la somma delle probabilità poste all'interno del logaritmo. Un fatto abbastanza importante da notare è che per massimizzare tale quantità dobbiamo evidenziarne la derivata. Sorprendentemente ne emerge che la derivata è semplicemente pari a:

$$\frac{\partial\ log(p(v))}{\partial w_{ij}} = \langle s_i s_j \rangle_v - \langle s_i s_j \rangle_{model} \tag{22}$$

dove v è un vettore del training set e $\langle x \rangle$ rappresenta il valore atteso di x.
Cerchiamo di dare un significato intuitivo alla derivata. Come cambia il peso tra i e j è legato a due quantità:

1. Il legame si rinforza tanto più i due neuroni si attivano contemporaneamente quando ricevono l'input v

2. Il legame si indebolisce tanto più i due neuroni si attivano contemporaneamente quando non ricevono in input il vettore v

Alternativamente possiamo pensare al primo termine come al termine di memorizzazione per la macchina di Boltzmann e al secondo termine come quello che cerca di cancellare i minimi spuri.
In generale l'apprendimento può essere difficoltoso perché supponendo di avere una catena di neuroni dove quelli visibili sono quelle agli estremi, per andare a cambiare efficacemente i pesi dei neuroni visibili dobbiamo conoscere anche il valore dei pesi nascosti.

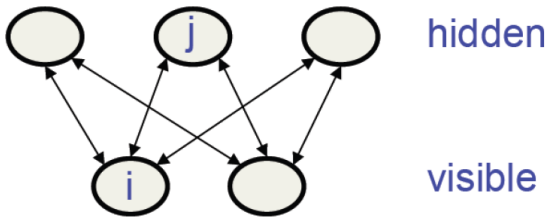


Figura 35: Esempio di RBM con due neuroni visibili e tre hidden

8.3 Restricted Boltzmann machines

Per ovviare ai problemi di cui abbiamo parlato poco fa, solitamente riduciamo il numero possibile di connessioni in una rete di Boltzmann per apprendere più facilmente. Queste reti "semplificate" vengono dete *restricted Boltzmann machines*(RBM) e hanno le tre seguenti limitazioni:

- Un solo livello hidden
- Nessuna connessione tra le unità hidden
- Nessuna connessione tra le unità visibili

Notiamo quindi che una RBM è di fatto un grafo bipartito. Mostriamo in Figura 35 un esempio di RBM.

In questo caso la probabilità che un neurone si attivi è pari a:

$$p(h_j = 1) = \frac{1}{1 + e^{-(b_j + \sum_{i \in vis} v_i w_{ij})}} \tag{23}$$

Un aspetto importante da considerare è che, essendo il grafo bipartito, ogni neurone hidden è collegato solo a neuroni visibili. Questo implica che il calcolo della quantità $\langle v_i h_j \rangle_v$ è indipendente dagli altri neuroni e quindi immediato.

05 / 12 /2017

Un problema delle reti di Boltzmann è che il procedimento di allenamento consiste nell'alternare una computazione di aggiornamento sul livello visibile a una sul livello nascosto, fino a quando lo riteniamo necessario. Questa situazione di inefficienza viene mostrata in Figura 36.

In questo caso per calcolare l'aggiornamento dei pesi dovremmo valutare l'espressione che deriva direttamente dalla Equazione 22.

$$\Delta w_{ij} = \varepsilon \cdot (\langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^\infty) \tag{24}$$

dove ε è un learning rate, il primo termine è quello relativo al primo aggiornamento, e il secondo termine è quello relativo all'aggiornamento dopo infiniti passi.

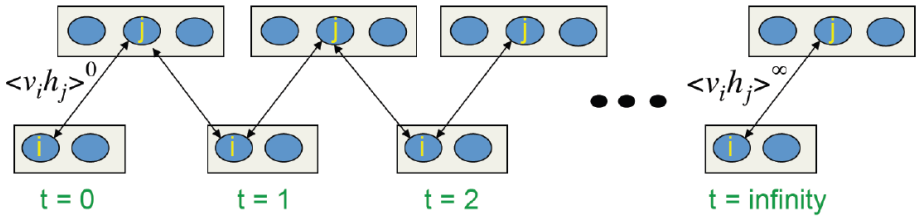


Figura 36: Computazione inefficiente di una rete di Boltzmann

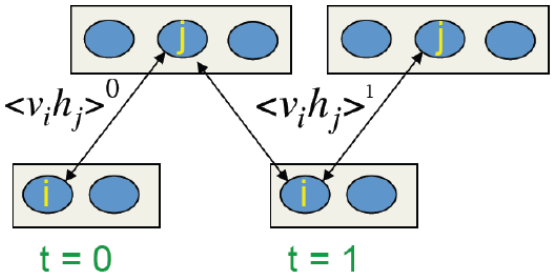


Figura 37: Computazione efficiente

Notiamo che ovviamente nella pratica dovremmo terminare dopo aver eseguito un numero di passi sufficientemente grande a piacere.
In realtà è stato verificato nel 2002 da Hinton che è possibile allenare la rete in due soli passaggi come mostrato in Figura 37.
Questo può essere fatto andando a calcolare il Δw_{ij}

$$\Delta w_{ij} = \varepsilon(< v_i h_j >^0 - < v_i h_j >^1) \tag{25}$$

Perché questa scorciatoia funziona? Di fatto quando partiamo dai dati possiamo valutare la direzione che stiamo prendendo in pochi passi e dunque quando vediamo che i pesi non sono buoni possiamo fermarci subito senza perdere tempo. Andiamo a verificarlo tramite un esempio

Esempio 3 : supponiamo di avere una rete come quella mostrata in Figura 38 vogliamo cercare di correggere i pesi affinché il pattern riconosciuto dalla rete sia $\{1, 0\}$. Supponiamo che un neurone si attivi se la probabilità di attivazione è maggiore di 0.5. Supponiamo che il bias sia pari a 0 e che $\varepsilon = 1$.

Cominciamo col calcolare la probabilità che si attivi il neurone hidden.

$$p_1(h_1 = 1) = \frac{1}{1 + e^{-(b_j + \sum_{i \in vis} v_i w_{ij})}} = \frac{1}{1 + e^{-1}} \simeq 0.73$$

Poiché $p(h_1 = 1) > 0.5$ il neurone hidden si attiva.

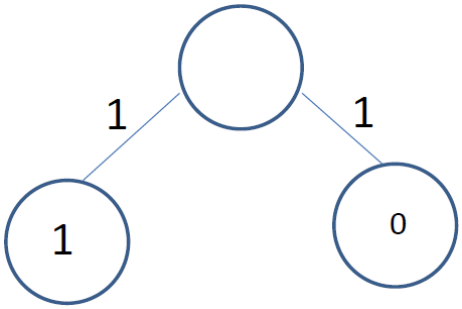


Figura 38: Rete dell'esempio 3

A questo punto andiamo a calcolare la successiva computazione sullo stato successivo dei neuroni visibili sapendo che il neurone hidden si attiva.

$$p_2(v_1 = 1) = \frac{1}{1 + e^{-1}} \simeq 0.73 \qquad p(v_2 = 1) = \frac{1}{1 + e^{-1}} \simeq 0.73$$

Dunque entrambi i neuroni si attivano. Andiamo ad effettuare nuovamente il calcolo sul neurone nascosto sapendo che ora entrambi i neuroni sono attivi.

$$p_2(h_1 = 1) = \frac{1}{1 + e^{-2}} \simeq 0.88$$

Dunque alla seconda computazione il neurone hidden resta attivo.
A questo punto possiamo calcolare il valore Δw_{ij} come segue:

$\Delta w_{v_1 h_1} = (v_1 h_1)^1 - (v_1 h_1)^2 = 1 - 1 = 0$
$\Delta w_{v_2 h_1} = (v_2 h_1)^1 - (v_2 h_1)^2 = 0 - 1 = -1$

Quindi il peso tra v_1 e h_1 resta invariato mentre il peso tra v_2 e h_1 viene decrementato di uno.
In conclusione, affinché la rete riconosca il pattern 10 dobbiamo lasciare invariato il primo peso e andare a ridurre di uno il secondo peso.

9 Deep learning: parte 2

9.1 Deep learning e Boltzmann

Come possiamo riconsiderare le reti neurali profonde alla luce di quanto visto nel capitolo precedente? Di fatto uno dei problemi principali delle reti profonde è che la probabilità di finire in un minimo locale è molto elevata a causa del numero elevato di neuroni, dunque di pesi e di parametri associati. Per questo motivo un classico allenamento di una rete neurale profonda consta di due fasi successive, *pre-train*, e *fine-tuning*

1. Fase di *pre-train*: andiamo ad effettuare un aggiornamento dei pesi della rete considerando due livelli della rete alla volta. L'effettivo aggiornamento viene fatto utilizzando o una tecnica basata sugli *autoassociatori*, oppure utilizzando delle *Restricted Boltzmann Network*(RBM)

   - Autoassociatori: andando a considerare soltanto due livelli della rete replichiamo l'input e cerchiamo di ricostruire dal livello hidden il livello di input. Ovviamente dovremo stare attenti a che il livello nascosto non ponga semplicemente pari a 1 tutti i pesi fornendoci una soluzione triviale.
     Questo tipo di apprendimento è non supervisionato perché di fatto l'algoritmo utilizzato(backpropagation) richiederebbe l'output ma in questo caso il nostro output coincide con l'input.
     E dove non abbiamo a disposizione l'input? In quel caso dobbiamo andare a propagare l'input nei livelli più interni andando di fatto a costruire un training set per essi.
     Ogni livello nascosto va a catture una diversa -e possibilmente più astratta- rappresentazione dell'input.
   - RBM: con questa tecnica andiamo ad impilare una serie di reti Boltzmann in cui ognuna vada a scoprire delle regolarità nell'input ricevuto dal livello precedente.
     La prima rete basata su questo paradigma è stata costruita nel 2006 impilando quattro RBM.
     Una rete costruita in questo modo viene detta *deep belief net*. Il vero e proprio allenamento consiste in una prima fase di *codifica* in cui un immagine attraversa una serie di livelli con un numero decrescente di neuroni il cui output è una rappresentazione succinta dell'immagine. Da questa rappresentazione andare ad effettuare una *codifica* che fa convergere la rappresentazione a una possibile risposta. L'accuratezza della risposta viene migliorata ulteriormente tramite la fase finale di fine tuning.

   L'idea complessiva del pre-training è che tramite esso si possa far partire la rete in una posizione buona nello spazio dei pesi da cui raggiungere un buon minimo.

2. Fase di *fine-tuning*: nella fase di fine tuning vengono modificati solo leggermente i valori dei pesi associati per cercare di migliorare la prestazione della rete.