Introduzione

Una Rete Neurale è un modello matematico che si ispira alla rete neurale biologica umana.

Una rete neurale è composta da neuroni connessi fra loro mediante link . Ogni link ha un peso, ovvero un parametro numerico. A seconda del problema, la variazione di questi pesi consente di addestrare la rete e garantire output adeguati.

L'obiettivo è quindi di addestrare una rete neurale, cioè trovare il valore ottimale per i pesi.

Una rete neurale è specificata da:

- **Una architettura**: che descrive come vengono interconnessi i neuroni e i relativi pesi;
- **Neurone**: che rappresenta l'unità di processamento dell'informazione;
- **Un algoritmo di apprendimento**: utilizzato per addestrare la rete neurale. Definisce delle regole per l'aggiornameto dei pesi.

I neuroni possono essere di tre tipi:

- **Input**: contengono l'input;
- **Hidden**: sono neuroni che si occupano del processamento;
- **Output**: che restituiscono l'output;

Il neurone è descritto da:

- **Un insieme di connessioni pesate**: descrivono i dati in ingresso del neurone. Ogni connessione è caratterizzata da un valore numerico, il peso;
- **Funzione di somma**: effettua la sommatoria degli input pesati;
- **Funzione di attivazione:** che limita l'ampiezza dell'output del neurone. Generalmente si limita l'output all'intervallo [0, 1], oppure [-1, 1];

I parametri quindi sono:

- Vettore input x
- Vettore pesi w
- Funzione somma: $u_j = \sum_{i=1}^{p} w_{ji} x_i$
- Funzione di attivazione $\varphi(-)$: è differenza fra il risultato della funzione somma e il bias: $\varphi(u_j - b_j)$.
- Campo indotto v: il valore u dopo che ha subito l'effetto del bias b. Il bias evita che se l'input è nullo, anche l'output lo sia. È come applicare una somma o una differenza alla quantità u. Può essere visto come un parametro aggiuntivo.
- Output del neurone $y = \varphi(u_j - b_j) = \varphi(v_j)$

Le reti neurali possono essere di due tipologie

- **Supervisionate**: gli esempi etichettati e sono rappresentati da coppie input-output atteso. Viene specificato alla rete cosa deve imparare attraverso il training set di esempi.
- **Non supervisionate:** alla rete non viene specificato cosa deve imparare. Sarà la rete ad individuare regolarità nell'input in modo da procedere con la categorizzazione.

Reti neurali supervisionate

Il Percettrone

Il percettrone è una rete neurale a livello singolo, la più semplice. È formata da un solo neurone in uscita e tanti in ingresso. Ogni input pesato va da un neurone di ingresso a quello di uscita.

Per l'addestramento: si cerca il valore dei pesi affinché gli esempi siano correttamente classificati. Dal punto di vista geometrico equivale a trovare un iperpiano che separi gli esempi delle due classi. Ciò equivale a trovare un retta $\sum_{i=1}^{m} w_i x_i + w_0 = 0$ che separa il piano in due regioni, ognuna delle quali descrive una classe. Gli input sono rappresentati utilizzando vettori m dimensionali.

Algoritmo di apprendimento: dati degli esempi appartenenti alle classi $C1$ e $C2$, bisogna allenare il percettrone in modo tale che classifichi correttamente gli esempi delle due classi. In sostanza:

- Se l'output del percettrone è 1, allora l'input (esempio) è assegnato alla classe $\boldsymbol{C1}$;
- Se l'output del percettrone è -1, allora l'input (esempio) è assegnato alla classe $\boldsymbol{C2}$.

Supponiamo allora di avere la retta r di separazione fra le 2 classi r: $ax + by + c = 0$

Supponiamo adesso di avere un punto P(a, b) che dev'essere classificato in $C1$. Dobbiamo allora determinare i valori di a, b e c in modo tale che sostituendo il punto P(a, b) l'equazione sia positiva.

Inoltre, assumendo come valore del bias c, possiamo definire il vettore x_n come tripla del bias più le coordinate del punto: $xn = [c, a, b]$

La regola di aggiornamento dei pesi è la seguente:

$$w_{n+1} = \begin{cases} w_n & se\ w_n^T \cdot x_n > 0\ e\ x_n \in C_1 \\ w_n & se\ w_n^T \cdot x_n < 0\ e\ x_n \in C_2 \\ w_n - \eta x_n & se\ w_n^T \cdot x_n > 0\ e\ x_n \in C_2 \\ w_n + \eta x_n & se\ w_n^T \cdot x_n < 0\ e\ x_n \in C_1 \end{cases}$$

Possiamo definire la pseudocodifica dell'algoritmo come segue:

1. n=1;

2. inizializza w(n) casualmente;

3. while (vi sono esempi classificati non correttamente)

a. prendi un esempio non correttamente classificato [x(n),d(n)];

b. w(n+1) = w(n) + ηd(n)x(n);

c. n = n+1;

4. end while

η= parametro di tasso di apprendimento (numero reale)

Limitazioni del percettrone

Sebbene sia semplice da allenare, il percettrone risulta limitato perché non è in grado di risolvere problemi non linearmente separabili. Ciò significa che è in grado di modellare funzioni booleane come AND o OR, ma non è in grado di modellare lo XOR. Nel caso di problemi non linearmente separabili si utilizza l'adaline.

Adaline

L'adaline (Adaptive Linear Element) è una rete neurale supervisionata, simile al percettrone, che permette di risolvere problemi di classificazione binaria non linearmente separabili. La funzione di trasferimento è la funzione identità. L'obiettivo è quello di ottenere un separatore lineare che minimizzi l'errore quadratico medio (somma dei quadrati degli errori).

L'adaline è composta da un modello di neurone lineare e utilizza l'algoritmo di apprendimento dei minimi quadrati.

In base al k-esimo esempio (x^k , d^k), l'errore è definito come:

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

Dove x_j^k è il k-esimo esempio di dimensione j (descritto da j caratteristiche). L'errore rappresenta la differenza tra ciò che ci aspettiamo, cioè d^k e quello che la rete dà in output, cioè y^k

L'errore totale è la somma degli errori quadratici di ciascun esempio, quindi per N esempi risulta:

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

La funzione è derivabile ovunque e quindi è possibile calcolare il gradiente dell'errore, che indica la direzione di crescita dell'errore. A questo punto per minimizzare l'errore totale della rete basterà muoversi nella direzione opposta al gradiente.

Il gradiente di una funzione a più variabili è un vettore le cui componenti sono le derivate dell'errore rispetto a ciascun peso. L'algoritmo si basa sulla **discesa incrementale del gradiente**: ad ogni iterazione l'algoritmo selezionerà un esempio e ne decrementerà l'errore.

1. Si parte da un vettore di pesi random;
2. Si calcola il gradiente e ci si muove nella direzione opposta negando il tutto (∇: nabla, il gradiente):

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

3. Si fa un passo di dimensione eta nella direzione opposta al gradiente:

$$w_{k+1} = w_k - \eta\left(\nabla E_w^k\right)$$

Regola di aggiornamento dei pesi

Derivando parzialmente l'errore totale rispetto al peso w_j otteniamo:

$$\frac{\partial E_{tot}}{\partial w_j} = \sum_{k=1}^{N} \frac{\partial E^{(k)}(w)}{\partial w_j} = \sum_{k=1}^{N} \frac{\partial}{\partial w_j} \frac{1}{2} \left(d^{(k)} - \sum_{j=0}^{m} x^{(k)}{}_j w_j \right)^2$$

Poiché sappiamo che la derivata di ogni singolo errore rispetto al peso w_j è:

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

La regola di aggiornamento dei pesi è:

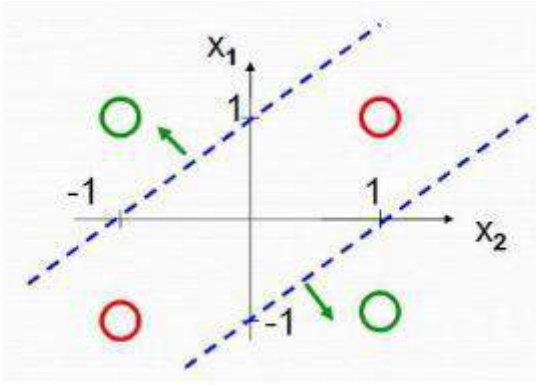
$$w_{k+1} = +\eta\left(d^k - y^k\right)x^k$$

RN Feed-Forward Multilayer

Il Multilayer perceptron, permette di risolvere problemi non linearmente separabili grazie all'utilizzo di uno o più livelli nascosti. I livelli nascosti sono utilizzati come unità di elaborazione. Non vengono visti dall'esterno e non ricevono direttamente input né inviano output. Avere livelli nascosti permette di proiettare l'input in una dimensione maggiore, quindi un qualunque iperpiano sarà in grado di separare le classi.

Per capire il funzionamento ricorriamo al problema del XOR

Lo XOR è un problema non linearmente separabile, tale funzione restituisce VERO quando gli argomenti sono diversi, e FALSO quando sono uguali.



Da questo grafico, si può intuire come costruire la rete neurale. Avendo 2 bande delineate dalle 2 rette, possiamo costruire un livello nascosto con 2 neuroni nascosti, ciascuno dei quali fa la sua classificazione. Il fatto che i 2 oggetti verdi siano nella stessa classe VERO, implica la necessità di un terzo neurone in cascata che faccia la somma, ovvero combini i risultati dei 2 neuroni hidden, dando l'output finale. Questa rete neurale usa la funzione segno come funzione di attivazione e i pesi vengono aggiornati con il meccanismo della backpropagation.

Modello di rete Feed-Forward con funzione Sigmoide

L'algoritmo di apprendimento si basa sul metodo della discesa del gradiente, per questo motivo la funzione deve essere continua e derivabile ovunque. Si utilizza la funzione Sigmoide:

$$Sigmoid: \varphi(v) = \frac{1}{1+e^{-av}}$$

L'argomento v è la somma pesata degli input, *a* è un parametro positivo il cui valore deve essere determinato. Al crescere di *a*, la funzione sigmoide cresce rapidamente e approssima la funzione step. La funzione sigmoide è compresa tra 0 e 1; per valori molto positivi tende automaticamente a 1.

Tale funzione presenta un limite in quanto si hanno zone in cui essa si appiattisce (zone asintotiche), il risultato della derivazione sarà nullo e quindi non si avrà più la possibilità di cambiare pesi. Tale problema è detto problema di saturazione.

Backpropagation: algoritmo di retro-propagazione dell'errore

L'algoritmo di retro-propagazione dell'errore cerca dei valori dei pesi che minimizzino l'errore totale della rete sul training set. L'algoritmo è composto da due step ripetuti iterativamente:

- **Passo in avanti**: i pesi vengono scelti casualmente e i valori di input vengono propagati in avanti verso i nodi di output in modo da calcolare l'errore sull'output;
- **Passo all'indietro**: L'errore della rete è utilizzato per aggiornare i pesi. L'errore è retro-propagato all'indietro attraverso la rete, livello per livello, fino al livello successivo a quello di input. Si calcola in maniera ricorsiva il gradiente locale di ogni pesi in modo da minimizzare l'errore quadratico medio.

Visto che l'apprendimento è supervisionato, l'insieme di addestramento è formato da coppie $[x(n), d(n)]$, dove $x(\mathrm{n})$ è l'n-simo esempio e $d(n)$ è il target che si vuole ottenere. d(n) può essere un reale o un vettore multidimensionale. La dimensione corrisponde al numero di classi.

Errore associato al singolo neurone di output j

L'errore associato al neurone di output *j* dopo l'attivazione della rete sull'n-simo esempio è pari a:

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

Avremo un e_j diverso per ogni neurone *j* al livello di uscita. Questo per ogni esempio n che presentiamo alla rete. Quindi l'indice *j* conta il numero di neuroni presenti nel livello di uscita, *n* conta il numero di esempi che stiamo analizzando.

Errore associato al singolo esempio n

L'errore relativo all'n-simo esempio sarà dato dalla somma al quadrato degli errori prodotti da tutti i neuroni di output per quell'esempio:

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

Errore quadratico medio totale

È la somma su tutti gli esempi, degli errori relativi

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

Minimizzare questa quantità, dal punto di vista del calcolo dei gradienti, trattandosi di una somma, equivale a minimizzare ciascuno dei termini E(n).

Regola di aggiornamento dei pesi

La regola di aggiornamento dei pesi si basa sulla discesa del gradiente, quindi ci si muove nella direzione opposta al gradiente dell'errore. La regola è così definita:

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

Il pes w_{ji} è modificato sommandogli una variazione. Tale quantità è un passo di dimensione eta nella direzione opposta al gradiente:

$$\Delta w_{ji} = -\eta \frac{\mathrm{dE}}{dw_{ji}} \qquad con\, \eta > 0$$

Il campo di ingresso v_j nel primo livello nascosto è uguale all'input dei neuroni di ingresso; nei livelli nascosti successivi sarà l'uscita del livello nascosto precedente quindi in generale:

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

Spezzettiamo la componente della quantità di variazione rappresentata dal gradiente in prodotto di derivate:

$$\frac{\mathrm{dE}}{dw_{ji}} = \frac{dE}{dv_j}\frac{dv_j}{dw_{ji}}$$

La lettura di quelle derivate al contrario, mi riproduce quello che succede nell'ultimo livello della rete neurale: variando il peso, mi varia l'ingresso del neurone, e mi varia anche la sua uscita, e di conseguenza varia anche l'errore.

Derivando i due fattori che la compongono otteniamo: $\delta_j = -\frac{dE}{dv_j} \quad \frac{dv_j}{dw_{ji}} = y_i$

La regola di aggiornamento dei pesi è data dal prodotto di 3 quantità:

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

Per calcolare il segnale di errore dobbiamo tener conto di due casi:

1) **Il neurone j è al livello di uscita:** spezzettiamo il segnale nel prodotto di 3 derivate

$$-\frac{dE}{dv_j} = -\frac{dE}{de_j}\frac{de_j}{dy_j}\frac{dy_j}{dv_j}$$

Questo ci fa capire che: variando il peso varia anche l'ingresso e di conseguenza variando l'ingresso, varia anche l'uscita e quindi varierà anche l'errore commesso dal neurone.
Si ottiene così:

$$-\frac{dE}{de_j}\frac{de_j}{dy_j}\frac{dy_j}{dv_j} = -e_j(-1)\varphi'(v_j)$$

Da qui la quantità di variazione del peso è data da:

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

Dove:

- (dj – yj) modifica il peso in maniera proporzionale alla discrepanza tra output atteso e quello ottenuto
- $\varphi'(vj)$ ci permette di controllare la divergenza della modifica del peso (ma potrebbe impedirlo con la derivata nulla)
- yi che indica l'output del neurone precedente
- 0< η <1 indica quanto sarà grande la modifica

2) **Il neurone j è un neurone hidden**: In questo caso il segnale di errore δj è calcolato utilizzando i segnali di errore dei livelli successivi .
In questo caso il segnale di errore è:

$$\delta_j = -\frac{dE}{dv_j} = -\frac{dE}{dy_j}\frac{dy_j}{dv_j} \quad \text{dove} \quad \frac{dE}{dy_j} = \sum_{k\ in\ lvl\ succ} \frac{dE}{dv_k}\frac{dv_k}{dy_j}$$

Il segnale d'errore ottenuto è:

$$\delta_j = -\sum_{k\ in\ lvl\ succ} \frac{dE}{dv_k}\frac{dv_k}{dy_j}\varphi'(v_j) = -\sum_{k\ in\ lvl\ succ} \delta_k w_{kj}\ \varphi'(v_j)$$

dE/dvk per definizione è il segnale di errore δk del neurone k-esimo, dvk/dyj è la derivazione del campo in ingresso vk rispetto a yj, che è l'uscita. Il campo in ingresso è la somma pesata dell'uscita del livello precedente. Se derivo rispetto all'uscita, rimane solo il peso che contribuiva a formare quella quantità, quindi wkj.

La variazione è data da:

$$\Delta w_{ji} = \eta\, \delta_j y_i = \eta \sum_{k\ in\ lvl\ succ} \delta_k w_{kj}\ \varphi'(v_j)\, y_i$$

Infine, possiamo vedere la regola di aggiornamento descritta come:

$$\delta_j = \begin{cases} \varphi'(v_j)(d_j - y_j) & se\ j\ è\ al\ livello\ di\ uscita \\ \varphi'(v_j) \sum_{k\ in\ lvl\ succ} \delta_k w_{kj} & se\ j\ è\ in\ un\ livello\ nascosto \end{cases}$$

Dove in particolare, usando la funzione Sigmoide, possiamo esprimere:

$$\varphi'(v_j) = a y_j (1 - y_j)$$

Regola delta generalizzata ed euristiche per gli iperparametri

Vi sono alcuni punti cruciali nella derivazione della regola di addestramento ad esempio lo stesso parametro η il quale se risulta essere troppo piccolo allora l'algoritmo apprende molto lentamente, al contrario se troppo grande, può causare un comportamento instabile con oscillazioni dei pesi.

Una tecnica che aiuta a controllare questo problema è la regola delta generalizzata:

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

Il termine di correzione del generico peso Δwji è formato non soltanto dal termine $\eta\ \delta jyi$, ma dall'aggiunta di un termine $\alpha\Delta wji$ al passo precedente n-1. Questa quantità $\Delta wji\ (n-1)$ relativa al passo precedente è in qualche modo ispirato dal tenere in conto la derivata al passo precedente, perché se siamo in un regime di oscillazione, Δwji cambia segno ad ogni step.

Con questa formula è possibile osservare che se non vi è presente oscillazione, l'aggiunta di questo termine rafforza la discesa del gradiente, quindi la rende più veloce, perché abbiamo la somma di tue termini con stesso segno e la variazione delta sarà maggiore. Invece se siamo in un regime di oscillazione, avremo la somma di un segno negativo ed uno positivo e ciò frena la discesa permettendo di avvicinarsi al valore minimo più precisamente.

Inoltre, si introduce un iperparametro $\boldsymbol{\alpha}$, positivo, con valori simili ad η, quindi $0 < \alpha < 1$.

In fine si può tener presente di due euristiche:

- Ogni peso ha il proprio valore η;
- Ogni η può variare da un'iterazione all'altra.

Modelli di addestramento

1) **Per pattern:** consideriamo il training set e passiamo alla rete un esempio alla volta, quindi i pesi della rete vengono aggiornati ogniqualvolta presentiamo un esempio. Gli esempi devono essere presentati in maniera casuale, altrimenti si rischia che la rete decrementi l'errore solo per una determinata classe. È possibile ripetere il procedimento per più epoche. È più veloce.

2) **Per epoche:** in questo caso presentiamo tutto il training set e solo successivamente aggiorniamo i pesi. Il procedimento viene ripetuto per più epoche, fino a quando non si raggiunge il criterio di arresto. È la metodologia più corretta perché il nostro obiettivo è minimizzare l'errore sull'intero dataset. È più lento

Criteri d'arresto

L'obiettivo è quello di minimizzare l'errore sul training set. Ci si ferma quando l'errore tra le varie epoche si stabilizza (varia di 0.1 o 0.01). Inoltre apprendere troppo bene il dataset porta ad overfitting. Per testare la capacità di generalizzazione della rete si partiziona il dataset in questo modo:

- 65/70% **training set**: utilizzato per addestrare la rete
  - Di cui 10/15% **validation set**: utilizzato per testare la capacità di generalizzazione della rete in fase di training. Inoltre viene utilizzato per determinare il valore degli iperparametri.
- 30/35% **test set**: utilizzato per il test quando l'addestramento è finito. Contiene esempi mai visti dalla rete.

Progettazione di una rete neurale

Al fine di progettare una buona rete neurale, dobbiamo tenere ben conto di questi elementi:

- **Rappresentazione dei dati**: i valori che vengono usati per descrivere un problema possono avere differenti range. È opportuno utilizzare la normalizzazione in modo da avere valori in un unico range ed evitare che delle features prevalgano su altre.

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

- **Topologia della rete**: riguarda il numero di neuroni, di livelli nascosti e il modo in cui sono connessi i neuroni. Abbiamo due tecniche per scegliere il numero di neuroni per livello:
  - **Pruning:** si da una rete sovradimensionata e si eliminano neuroni e connessioni fino quando non degradano le prestazioni le prestazioni .
  - Si parte da una rete sottodimensionata e si aggiungono neuroni a poco a poco finché non si raggiungono le prestazioni desiderate.
  - **Tecnica prof**: si parte da una rete sottodimensionata, si registrano le performance per ogni epoca e man mano si aggiungono neuroni finché non si raggiungono le prestazioni desiderate.

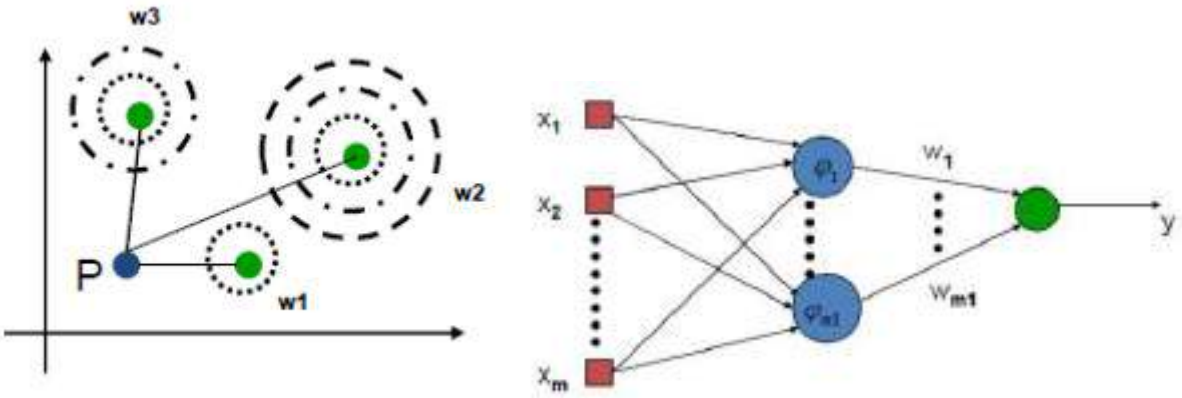
- **Parametri della rete**:
  - I pesi sono scelti casualmente negli intervalli [-1,1] o [-0.5,0.5]
  - Per η il valore dipende dal problema
  - Per il numero di esempi vi sono due regole empiriche:
    - **Rule of thumb**: il numero di esempi di training dovrebbe essere da 4 e 10 volte il numero dei pesi della rete. Se si usa una rete troppo grande rispetto al numero di

esempi overfitting; se invece la rete è troppo piccola rispetto al numero di esempi, la rete non apprende bene.
    - Si considera un numero N di esempi di addestramento tale che: $N > |W|/(1 - a)$ dove $|W|$ è il numero dei pesi e a è l'accuratezza attesa sull'insieme di test.
- **Applicazioni**:
    - Funzioni booleane: possono essere rappresentate da un singolo livello nascosto.
    - Funzioni continue limitate a tratti: quindi funzioni continue su un insieme limitato, possono essere approssimate arbitrariamente bene utilizzando un livello nascosto.
    - Funzioni continue: possono essere approssimate arbitrariamente bene utilizzando due livelli nascosti

Reti a funzioni radiali

Una rete radiale è una rete in cui i neuroni del livello nascosto sono caratterizzati da una funzione di attivazione radiale, ovvero una funzione di attivazione il cui output dipende dalla distanza tra l'input della rete e un vettore memorizzato, detto centro. Le funzioni radiali restituiscono valori più elevati in un intorno, quindi si attivano maggiormente per input vicini al centro.

Una rete radiale è composta da un livello nascosto con funzioni di attivazione radiali e un livello di uscita con neuroni lineari. Il livello di uscita effettuata la somma pesata delle uscite del livello nascosto.



Architettura

Una funzione radiale è della forma: $\varphi(||x - t||)$, dove x-t è la distanza dell'input x dal centro t. I pesi in ingresso al livello nascosto portano informazione sulle coordinate del centro della funzione. I pesi che collegano il livello nascosto al livello di uscita sono di tipo classico. Nel caso di un neurone di uscita abbiamo m1 pesi. I bias non sono presenti. A differenza delle classiche funzioni di trasferimento, che sono aperte e quindi danno valori non molto differenti anche per input molto grandi, le funzioni radiali delimitano una zona circolare. I valori che non cadono all'interno della zona assumono valori molto piccoli.

L'output y per un singolo neurone è:

$$y = w_1\varphi_1(||x - t_1||) + w_2\varphi_2(||x - t_2||) + \cdots + w_{m1}\varphi_{m1}(||x - t_{m1}||)$$

Una funzione radiale ha due parametri:

- σ: rappresenta lo spread, ossia l'ampiezza del centro della funzione
- t: è il centro della funzione.

Interpolazione

Nel problema dell'interpolazione consideriamo una rete radiale che ha più neuroni di output, quindi l'output della rete potrà essere rappresentato utilizzando una matrice.

Teorema (condizione di interpolazione)

Dato un insieme di N differenti punti $\{xi \in \Re^m, i = 1\ldots N\}$ (neuroni di uscita) ed un insieme di N numeri reali $\{di \in \Re, i = 1\ldots N\}$ (N target per N neuroni di uscita), si vuole trovare una funzione $F: \Re^m \rightarrow \Re$ che soddisfi la condizione di interpolazione: $F(xi) = di$. Ossia la rete neurale F applicata all'esempio *xi* restituisce il target *di*. La dimensione di *xi* è parti al numero di neuroni in ingresso.

Se:

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

Allora avremo:

$$\begin{bmatrix} \varphi(\| x_1 - x_1 \|) & \ldots & \varphi(\| x_1 - x_N \|) \\ & \ldots & \\ \varphi(\| x_N - x_1 \|) & \ldots & \varphi(\| x_N - x_N \|) \end{bmatrix} \begin{bmatrix} w_1 \\ \ldots \\ w_N \end{bmatrix} = \begin{bmatrix} d_1 \\ \ldots \\ d_N \end{bmatrix} \Rightarrow \Phi w = d$$

Ogni riga è un'applicazione di F ad ogni esempio. Applicare la condizione di interpolazione significa valutare F in ogni punto, avremo tante equazioni quante sono i punti. I punti devono essere diversi altrimenti avremo valori nulli e questo non va bene perché la matrice deve essere invertita. Questa condizione implica la presenza di N neuroni nascosti, uno per ogni esempio. Cosa del tutto irrealistica in quanto darebbe overfitting.

Teorema di Michelli

Sia $\{x_i\}$ i=1,…,N un insieme di punti distinti in $\Re^m$, allora la matrice di interpolazione Φ dove l'elemento di indice ji è della forma $\varphi_{ji}(||x_j - x_i||)$ è non singolare, quindi invertibile.

Parametri

Vi sono tre parametri fondamentali per le reti radiali:

1) **I centri di attivazione delle funzioni**: ovvero i punti di maggiore attivazione di ogni funzione di trasferimento per ogni neurone nascosto. Essi possono essere impostati in due modi:
    a. Sono scelti in modo random dal training set

b. Clustering: vengono selezionati in maniera random dal training set e successivamente raffinati:
   i. Inizializzazione: $ti(0)\ rand,\ i = 1 \ldots m1$
   ii. Campionamento: si disegnano gli esempi x
   iii. Matching di similarità: si cerca il centro più vicino all'esempio x secondo la formula:
   $$i(x) = \arg\min_k \left|\left|x(n) - t_i(n)\right|\right|$$
   iv. Le coordinate dei centri vengono modificate secondo l'equazione:
   $$t_i(n+1) = \begin{cases} t_i(n) + \eta[x(n) - t_i(n)] & se\ i = i(x) \\ t_i(n) & altrimenti \end{cases}$$
   Ovvero il centro che è stato selezionato come il più vicino ($se\ i = i(x)$) viene modificato sommando alle sue coordinate originali $ti(n)$ una quantità pari alla differenza tra l'esempio e il centro, il tutto moltiplicato per η.
   Si continua in questo modo incrementando x, quindi considerando tutti gli esempi o fino a quando i centri non cambiano.

2) **La larghezza (spreads) delle funzioni di attivazione:** è scelta tramite la normalizzazione. Varia in funzione della distanza media dei centri.

$$\sigma = \frac{distanza\ massima\ fra\ 2\ centri\ qualsiasi}{\sqrt{numero\ dei\ centri}} = \frac{d_{max}}{\sqrt{m_1}}$$

3) **Pesi:** sono scelti utilizzando il metodo della pseudoinversione.

Pseudo inversione

Per ogni esempio vorremmo avere y(xi)=di. Ciò è possibile solo se utilizziamo un numero di neuroni nascosti pari al numero di esempio, il che sarebbe altamente inefficiente e inoltre darebbe overfitting. Il nostro obiettivo sarà quello di approssimare di utilizzando un numero m1 di neuroni nascosti minore del numero N di esempi. (m1<N).

$$y(x_i) \approx w_1\varphi_1\left(\left|\left|x_i - t_1\right|\right|\right) + \cdots + w_{m_1}\varphi_{m_1}\left(\left|\left|x_i - t_{m_1}\right|\right|\right) \approx d_i$$

In forma matriciale per il singolo esempio:

$$\left[\varphi_1\left(\left|\left|x_i - t_1\right|\right|\right) + \cdots + \varphi_{m_1}\left(\left|\left|x_i - t_{m_1}\right|\right|\right)\right]\begin{bmatrix} w_1 \\ \cdots \\ w_{m_1} \end{bmatrix} = d_i$$

Per N esempi risulta:

$$\begin{bmatrix} \varphi_1\left(\left|\left|x_i - t_1\right|\right|\right) + \cdots + \varphi_{m_1}\left(\left|\left|x_i - t_{m_1}\right|\right|\right) \\ \cdots \\ \varphi_1\left(\left|\left|x_N - t_1\right|\right|\right) + \cdots + \varphi_{m_1}\left(\left|\left|x_N - t_{m_1}\right|\right|\right) \end{bmatrix}\begin{bmatrix} w_1 \\ \cdots \\ w_{m_1} \end{bmatrix} = [d_1 \ldots d_N]^T$$

Abbiamo N righe (N il numero di esempi) ed m1 colonne (m1 numero di neuroni nascosti), la matrice non è quadrata, ma rettangolare (lunga), quindi non è invertibile.

Chiamo le 3 matrici Φ, w *e* d. Sostituendo le etichette a ciò che ho sopra, ottengo: $\Phi w = d$.

Il vettore w è la soluzione del sistema lineare: $\Phi w = d$ oppure $\Phi^T \Phi w = \Phi^T d$

Il prodotto $\Phi^T\Phi$ trasforma la matrice in una matrice quadrata, poiché moltiplicare una qualsiasi matrice per la sua trasposta, darà come risultato una matrice quadrata. Otteniamo w : $w=\Phi^T d/\Phi T\Phi$

Matrice pseudoinversa

Definiamo $\Phi^+ = (\Phi^T\Phi)^{-1}\,\Phi^T$ la matrice pseudo inversa di Φ. Possiamo ottenere i pesi utilizzando la seguente formula:

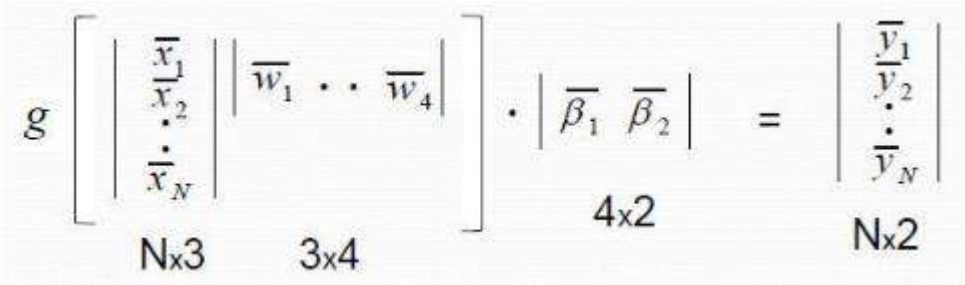
$$[w_1 \dots w_{m_1}]^T = \Phi^+ [d_1 \dots d_N]^T$$

Questo consente di ottenere tutti i pesi del livello di uscita in un solo passaggio. Solitamente per valutare Φ^+ dobbiamo memorizzare l'intera matrice Φ. Quindi perdiamo la possibilità di calcolare i pesi per ogni esempio.

Extreme learning machine

Extreme Learning Machine è un'alternativa alla back-propagation ed è utilizzato per addestrare un multilayer perceptron utilizzando un meccanismo di pseudo inversione.

I pesi del primo livello e i bias possono essere scelti in modo random, se le funzioni di trasferimento siano infinitamente differenziabili (si è poi dimostrato che non è necessario), allora è possibile determinare analiticamente i pesi di secondo livello β



Quindi, l'equazione Φw = d qui diventa: $H\beta = y$

Quindi risulta:

$$\beta = H^{-1}y \;\rightarrow\; H\beta \approx T \;\rightarrow\; \beta = H^{+}T$$

Dove y è il valore prodotto dalla rete, T i target avevamo in d. Vogliamo che $H\beta$ sia circa T, dove H è la matrice delle funzioni di attivazione.

L'algoritmo di Wang al posto della back propagation, determina i pesi di primo livello w in maniera random, date le funzioni di attivazione g (le stesse del multilayer perceptron, es sigmoidi) e i pesi random si può fare il prodotto e ottenere la base per determinare i pesi di secondo livello β.

L'algoritmo non è più veloce della backpropagation perché il fatto di scegliere i pesi di primo livello in maniera random porta a dover fare diverse iterazioni per trovare la combinazione giusta dei pesi.

Algoritmo

Dato un insieme di addestramento N={(xi, ti) | $xi \epsilon \Re^n$, t$i \epsilon \Re^m$, i = 1,…,N }, una funzione di attivazione g(x) e un numero di nodi nascosti pari a Ñ:

1) Si inizializzano casualmente i pesi wi e i bias bi i=1,…, Ñ (per tutti i neuroni nascosti)
2) Calcoliamo la matrice di output H del layer nascosto
3) Calcolcoliamo i pesi di output $\beta = H^+T$ dove $T = [t_1 \dots t_N]^T$
   In questo modo, la matrice H+ ha questa forma:

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

Per evitare di avere matrici quasi singolari (cioè con determinante non nullo, ma prossimo allo zero) si inverte una matrice H^+ a è stato aggiunto un contributo λI. Con questo approccio la soluzione è:

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

Deep Learning

Una rete profonda è una rete con più di due livelli nascosti organizzati gerarchicamente, ciò permette di elaborare e riutilizzare l'informazione in diversi step che si ripetono, rendendo questo modello più scalabile rispetto ai modelli classici. Il passaggio a queste reti è dovuto al fatto che alcune funzioni per essere approssimate in maniera efficiente richiedono molti livelli nascosti.

La complessità di queste reti è definita dalla profondità (numero di livelli, che variano da 7 a 50), dal numero di neuroni, di connessioni e di pesi. Maggiore sarà il numero di pesi e maggiore sarà il numero di operazioni necessarie per l'addestramento.

Le reti profonde risalgono al 1998, ma hanno avuto successo pochi anni fa con l'introduzione del modello AlexNet. Pur non introducendo grandi modifiche sono cambiate diverse cose:

- Per poter sfruttare al massimo le reti profonde servono grandi quantità di dati etichettati. L'avvento dei big data ha permesso ciò.
- L'evoluzione delle tecniche di calcolo, come la computazione su GPU ha permesso una drastica riduzione dei tempi di addestramento.
- Utilizzano la funzione di attivazione RELU che consente di utilizzare la backpropagation senza incorrere nel problema del gradiente che sparisce (come avveniva per la sigmoide che aveva derivata nulla in alcuni punti).

Reti convoluzionali

Le reti convoluzionali sono utilizzate principalmente per il processing di immagini e si basano su due principi di funzionamento, che rappresentano anche le differenze con il MLP:

- **Processing locale:** ogni neurone non è connesso a tutti i neuroni del livello precedente. In questo modo, ogni neurone esegue un processing locale del segnale in ingresso. Quindi vi è una forte riduzione del numero di connessioni.
- **Pesi condivisi:** Più neuroni hanno gli stessi pesi, cioè sono legati a quelli del livello precedente condividendo lo stesso valore dei pesi. Questo consente di ridurre il numero dei pesi.

Queste due caratteristiche permettono di ridurre il numero di parametri rendendo l'addestramento più leggero. Inoltre nel processing di immagini il fatto di avere il processing locale, pesi condivisi e pooling rende il modello più semplice ed efficiente rispetto ai modelli completamente connessi.

Architettura

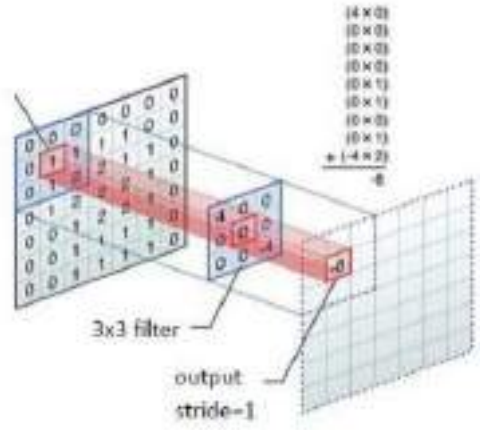
L'architettura di una rete convoluzionale è composta da una gerarchia di livelli, in particolare:

- **Il livello di input** è direttamente connesso ai pixel dell'immagine da processare;
- **I livelli intermedi** invece sono differenti fra loro ed utilizzano connessioni locali e pesi condivisi. Sono formati da 3 strati che si ripetono
    - **Strato convoluzionale**
    - **Strato di pooling**
    - **Stato ReLu**
- **Gli ultimi livelli:** fungono da classificatore e generalmente sono totalmente connessi.
- **Il livello di output:** contiene la classe dell'immagine calcolata con la funzione di attivazione SoftMax

Il fatto di avere connessioni locali e pesi condivisi fa sì che i neuroni processino allo stesso modo diverse porzioni di un'immagine. Ciò risulta fondamentale perché in un'immagine abbiamo dei pezzi ricorrenti che contengono le stesse informazioni.

Convoluzione

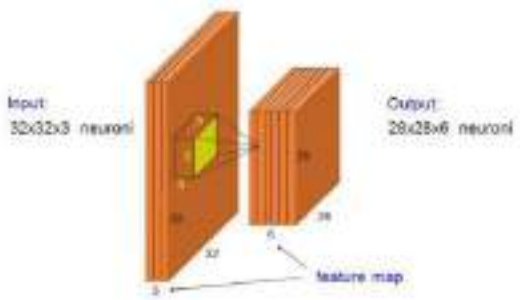
L'operazione su cui si basano le reti convoluzionale è la convoluzione, che consiste nell'applicare un filtro digitale all'immagine. In pratica si analizza l'immagine mediante un filtro. Il filtro è l'insieme dei pesi che viene fatto scorrere su diverse porzioni dell'input. Per ogni posizione viene generato in output il prodotto scalare tra il filtro e la porzione di input coperta.



Un parametro importale è lo stride, ossia di quanto far shiftare il filtro. Un problema della convoluzione è rappresentato dagli effetti di bordo, che si verificano quando il filtro esce dai bordi dell'immagine. La rete deve sapere come comportarsi.

Convoluzione 3D

Quello che in realtà si effettua è una convoluzione 3D: i livelli sono rappresentati da blocchi. Il filtro opera su una porzione di volume e non di superficie. Ogni neurone del livello ha associati i pesi (filtro), quindi ogni neurone processa un volume. Visto che parliamo di image processing, ogni livello di ingresso rappresenta uno dei colori fondamentali (RGB).



In ogni blocco ci sono tanti neuroni. I neuroni dello stesso blocco condividono i pesi, ciò significa che ciascun neurone del blocco andrà a cercare la stessa caratteristica in diversi punti dell'immagine: ad esempio, se i pesi di un blocco sono settati in modo da essere sensibile ad un edge verticale, tutti i neuroni del blocco cercheranno un edge verticale in tutta l'immagine).

I blocchi vengono dette **features map** perché vanno a cercare una determinata caratteristica nell'immagine. Il numero dei blocchi dipende dal numero di caratteristiche dell'immagine.

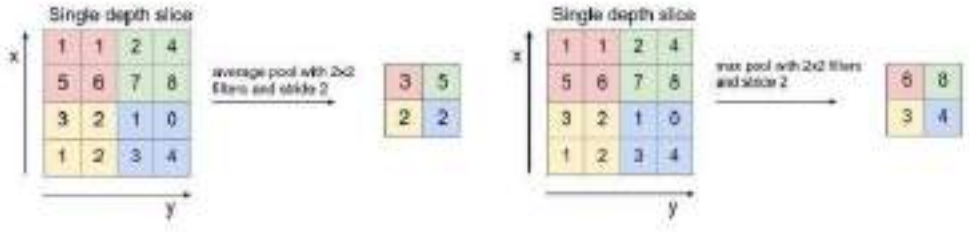
Strato di pooling

Il livello di pooling serve ad aggregare le caratteristiche processare nel livello precedente in modo da generare features map di dimensioni minori. L'obiettivo è ottenere invarianza rispetto alle trasformazioni dell'input, mantenendo l'informazione importante per la distribuzione. L'aggregazione funziona solo a livello di features map, quindi a livello di pooling abbiamo lo stesso numero di blocchi presenti nel primo livello convoluzionale. La dimensione di questi blocchi è però inferiore e dipende dallo stride che scegliamo.

Esistono due modi per fare il pooling:

1) **Max pooling:** consiste nel prendere una zona del primo livello convoluzionale e considerare i valori massimi.

2) **Average pooling:** consiste nel prendere una zona del primo livello convoluzionale e considerare esclusivamente il valore medio di quella zona

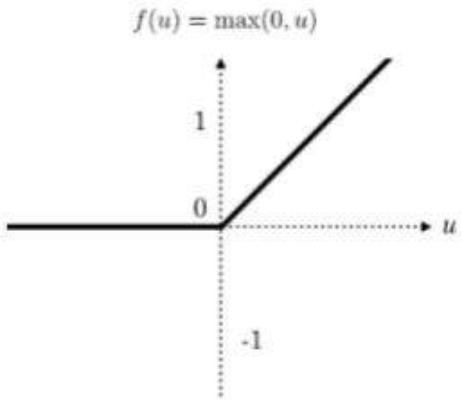


Funzione di attivazione ReLu

Nelle reti MLP la funzione di attivazione più utilizzata è la sigmoide. Nelle reti profonde l'utilizzo della sigmoide è problematico per la retropropagazione, a causa del gradiente che sparisce. Ciò può accadere per due motivi:

- Il primo è che la derivata della sigmoide è tipicamente <1, quindi l'applicazione della regola di derivazione a catena fa sì che si abbiamo molti termini minori di 1 moltiplicati tra di loro, con la conseguenza che il gradiente si riduce in maniera esponenziale per reti con molti livelli.
- Il secondo motivo è che in alcuni punti la derivata può essere nulla e quindi il gradiente si annulla.

Per questi due motivi si utilizza la funzione ReLu. Rispetto alla sigmoide, per valori negativi o nulli, anche la derivata è nulla. Questo porta ad avere dei neuroni spenti e questo aiuta in quanto aiuta a ridurre l'overfitting.



Strato fuly connected: Softmax e Cross-entropy

La funzione di trasferimento SoftMax è utilizzata per la classificazione e consiste di un numero *s* di neuroni, uno per ogni classe, completamente connessi ai neuroni del livello precedente. Il neurone di input net_k è calcolato come segue:

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

Rapportiamo l'esponenziale del campo in ingresso al neurone e^{nekc} a tutti i neuroni che costituiscono il livello. Rappresenta una sorta di normalizzazione. Essendo il rapporto tra una parte e la sommatoria di tutte le parti, non potrà mai essere maggiore di 1 e sarà sempre una quantità positiva perché si utilizza l'esponenziale. In questo modo eliminiamo i valori negativi.

I valori di z_k possono essere visti come una probabilità (sono compresi tra 0 e 1 e la loro somma è 1).

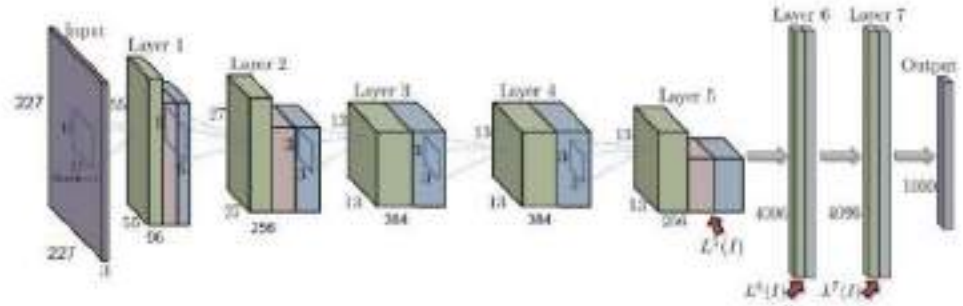
La risposta del neurone è funzione di quello che vedono gli altri neuroni.

Inoltre, al posto di utilizzare l'errore quadratico medio si utilizza la **cross entropy,** che rappresenta un modo diverso di vedere la discrepanza tra il valore atteso e il valore di output. Tra due distribuzioni discrete *p* e *q,* fissando p, si misura quanto q differisce da p.

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

Struttura di una rete convoluzionale

Ecco come si presenta una rete convoluzionale:



Abbiamo quindi:

- **Blocchi viola:** livello di ingresso (227x227x3) e livello di uscita (1000);
- **Blocchi verdi:** livelli convoluzionali;
- **Blocchi rosa:** livelli di pooling (approccio max pooling);
- **Blocchi blu:** funzione di attivazione Relu.

Transfer Learning

Il training di reti convoluzionali su dataset di grandi dimensioni può richiedere giorni o settimane. Inoltre il training di una rete convoluzionale su un nuovo problema richiede un training set etichettato di notevoli dimensioni. Per questi due motivi spesso non si effettua un training da zero, ma si utilizza il transfer learning, ossia si utilizza una rete neurale già addestrata per un task simile a quello che dobbiamo svolgere.

Ci sono due modalità:

- **Fine-Tuning:** si parte da una rete pre-allenata su un problema simile a quello che abbiamo, e:
    - Si sostituisce il livello di uscita con dei nuovi livelli di neuroni SoftMax in numero congruo al problema di classificazione che vogliamo risolvere.
    - Come valori iniziali dei pesi si usano quelli che abbiamo, tranne per le connessioni fra il penultimo e l'ultimo livello, in cui i pesi saranno inizializzati casualmente.
    - Infine si effettuano nuove operazioni di addestramento per ottimizzare i pesi rispetto alle peculiarità del nuovo dataset.

- **Riutilizzo delle features**: si utilizza una rete pre-allenata senza applicare il fine tuning. Si estraggono dai livelli intermedi le features generate dalla rete durante il passo in avanti sul nuovo dataset e si utilizzano queste features per addestrare un classificatore esterno in modo che sia in grado di classificare i pattern tipici del nuovo dominio.

Self Organizing Maps (SOM – Kohonen Maps)

Le SOM fanno parte delle reti neurali ad apprendimento non supervisionato: le mappe imparano da sole a classificare un set di input associando input simili a neuroni simili (neuroni vicini). I dati non sono etichettati, ma sarà la rete ad individuare input simili e raggrupparli. Il meccanismo è diverso da quello di backpropagation.

Le SOM possono essere di due tipi:

- **Monodimensionali:** in cui i neuroni sono organizzati secondo relazione di vicinanza in una struttura monodimensionale:
- **Bidimensionali:** in cui i neuroni sono organizzati in una struttura bidimensionale (una griglia).

La mappa effettua associa a zone vicine della mappa input simili e a zone lontane della mappa input diversi. In pratica la mappa è in grado di riconoscere somiglianze o differenza all'interno di un input e organizza questi input in base ad una relazione di vicinanza/lontananza. Per questo motivo le SOM hanno un funzionamento simile al cervello umano.

Passi per l'apprendimento: competitive learning

Il procedimento di apprendimento di queste reti è quindi il seguente:

1) Si inizializzano i pesi in modo casuale.
2) Si dà in pasto alla rete un primo stimolo sotto forma di vettore:
    - Per ogni input si attiva un meccanismo di competizione trai i neuroni, vedendo quale neurone si attiva di più rispetto allo stimolo. Da quel momento il neurone che ha vinto la competizione farà lo stesso per input simili.
3) Si aggiornano i pesi del neurone che ha vinto in modo che sia sempre lui a rispondere a stimoli simili a quelli per cui si è attivato.
4) Infine abbiamo il meccanismo di cooperazione, in cui l'aggiornamento dei pesi viene fatto anche per i neuroni vicini. In questo modo si organizza la mappa.

In questo modo i neuroni in zone vicine della mappa risponderanno a stimoli simili, quelli lontani non risponderanno.

Algoritmo di apprendimento

1) **Inizializzazione:** si inizializzano i pesi in maniera casuale;
2) **Competizione:** si presenta lo stimolo alla rete e si trova la BMU, che è la somma pesata degli input. Per il neurone j risulta:

$$w_j^T x = w_{j1}x_1 + w_{j2}x_2 + \cdots + w_{jn}x_n$$

La BMU sarà il neurone j che ha attivazione maggiore:

$$w_i^T x = \max_{j=1}^{m}\left(w_{j1}x_1 + w_{j2}x_2 + \cdots + w_{jn}x_n\right) = w_{i1}x_1 + w_{i2}x_2 + \cdots + w_{in}x_n$$

La BMU equivale a trovare il neurone i la cui distanza euclidea dall'esempio x è minima

$$w_i^T x = \left||x - w_i|\right| \leq \left||x - w_j|\right| \qquad j = 1 \ldots m$$

Se ci sono più neuroni che si attivano si sceglie quello la cui distanza euclidea è minima.

3) **Modifica dei pesi:** viene effettuata in modo da ridurre la differenza tra il vettore dei pesi $w_i(n)$ e lo stimolo corrente x, il tutto modulato dal learning rate η.

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

Dopo la correzione, wi è più simile ad x. Per evitare che il peso equivalga all'input si utilizza η<1. In generale il learning rate viene decrementato ad ogni iterazione nel seguente modo:

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

Dove η_0 è il learning rate iniziale, solitamente 0.1, e τ è una costante.

4) **Cooperazione:** nella fase di cooperazione i pesi che dei neuroni che non sono BMU vengono cambiati. Per la correzione si utilizza il concetto di **neighborhood,** cioè di vicinanza, che rappresenta un insieme di neuroni j più o meno vicini alla BMU e che vengono coinvolti nella correzione dei pesi della BMU (neurone i). La relazione di vicinanza tra i e j è definita da una funzione gaussiana di vicinanza:

$$h_{j,i}(n) = \exp\left(-\frac{d_{j,i}^2}{2\sigma(n)^2}\right) \qquad \sigma(n) = \sigma_0 \exp\left(-\frac{n}{\tau}\right) \qquad \tau\ costante$$

Dove:

- dj,i è la distanza tra i e j.
- σ determina l'ampiezza della funzione
- $h_{j,i}$ avrà valore massimo quando j=i e mano a mano minore quando la distanza tra i e j aumenta.

La funzione di neighborhood determina il modo in cui le unità vicine alla BMU sono coinvolte nell'apprendimento e quindi come cambiano i pesi:

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

Il cambiamento dei pesi dei neuroni j sarà proporzionale alla distanza dalla BMU. I pesi non saranno aggiornati quando la $h_{j,i}$ è nulla. Andando avanti con le iterazioni la funzione di neighborhood assumerà valori sempre più piccoli e quindi i neuroni considerati vicini alla BMU saranno sempre di meno. In questo modo i neuroni vicini avranno pesi simili e risponderanno in maniera simile ad input simili, di conseguenza avremo una organizzazione topologica della mappa.

Fasi algoritmo di apprendimento

L'algoritmo di apprendimento si compone di due macro-fasi:

1) **Auto-organizzazione:** in questa fase appare l'ordine topologico, richiede circa 1000 epoche. Il learning rate decresce da 0.1 a 0.01, mentre la funzione di neighborhood all'inizio include tutti i neuroni e poi si restringe fino a considerare solo la BMU.
2) **Fase di convergenza:** in questa fase la corrispondenza tra i pesi dei neuroni e gli input diventa più precisa. Dura al più 500 iterazioni, il learning rate viene impostato a 0.01 e la funzione di neighborhood contiene solo la BMU.

L'addestramento termina quando non si hanno cambiamenti consistenti nei pesi.

Differenze tra FFNN e RBF

Architettura

- **RBF**: un solo livello nascosto
- **FFNN**: può avere più livelli nascosti

Modello neurale

- **RBF**: il modello per i neuroni hidden è *non lineare*, diverso da quello per i nodi output (*lineare*)
- **FFNN**: il modello per i due tipi di neuroni è identico (*non lineare*)

Funzione di attivazione

- **RBF**: l'argomento della funzione per ogni nodo nascosto è una funzione radiale che computa una *distanza euclidea* tra il vettore di input e il centro di ogni unità
- **FFNN**: l'argomento della funzione per ogni nodo hidden è un *prodotto interno* tra il vettore di input e il vettore pesi di quel neurone.

Approssimazione

- **RBF**: approssima tramite una funzione gaussiana che costruisce approssimazioni *locali*.
- **FFNN**: effettua una approssimazione *globale*.

Per concludere una rete RBF è usata per operazioni di **regressione** (stimare un'eventuale relazione funzionale esistente tra la variabile dipendente e le variabili indipendenti) e per operazioni complesse (non lineari) di **classificazione** di pattern, come accade per classificazioni di pattern circolari di punti.

Reti di Hopfield

Le reti di Hopfield sono state introdotte nel 1982 da John Hopfield. È un modello di rete non supervisionato. Queste reti sono utilizzate per il pattern completion, quindi memorizzano l'informazione e la recuperano a partire da informazioni corrotte, che rappresentano gli input della rete. Le informazioni di base sono dette memorie fondamentali. Quando presentiamo un input corrotto, la rete itera fino a quando l'input corrotto non converge ad una memoria fondamentale pulita.

Architettura

- La rete è formata da insieme di neuroni fortemente connessi tra di loro.
- I pesi sono simmetrici (ovvero vi è una connessione bidirezionale): $w_{ij} = w_{ji}$ e non connettono il neurone con se stesso.
- Gli stati di attivazione sono due: [-1, 1].

Funzione di attivazione

Per il neurone j all'iterazione n risulta:

$$y_j(n) = \varphi(v_j(n)) = \begin{cases} 1 \ se \ v_j > 0 \\ -1 \ se \ v_j < 0 \\ \varphi(v_j(n-1)) \ se \ v_j = 0 \end{cases}$$

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

Lo stato della rete con N neuroni ad una certa iterazione è una configurazione della rete con N neuroni con gli N elementi fissati a -1 e 1.

L'input è un vettore composto da 1 e -1 di dimensione N, ogni neurone rappresenta un elemento di input.

L'output è un vettore composto da 1 e -1 di dimensione N e sarà una memoria stabile più vicina possibile all'input corrotto. Una memoria è una configurazione di 1 e -1 di dimensione N.

L'attivazione dei neuroni è calcolata in modo *asincrono*: ogni volta una unità è scelta e viene calcolata l'attivazione secondo la funzione di attivazione . Se partiamo da differenti neuroni possiamo trovare stati stabili diversi, tuttavia il ***teorema di convergenza*** garantisce che dato uno stato iniziale, la rete arriverà sempre ad uno stato stabile, quindi non è possibile cambiare all'infinito gli stati della rete. Gli stati stabili agiscono come calamite: dati un qualsiasi stato, esso converge in uno stato stabile.

Apprendimento

Gli stati stabili sono quelli in cui noi vogliamo che la rete converga, quello che rende stabile uno stato sono i ***pesi***. Intuitivamente uno stato è stabile se:

- Neuroni con la stessa attivazione sono connessi da pesi positivi: (1,1) e (-1,-1).
- Neuroni con attivazioni opposte sono connessi da pesi negativi: (-1,1) e (1,-1).

L'obiettivo è quello di scegliere una configurazione dei pesi tale che le informazioni da memorizzare siano stabili. L'algoritmo di apprendimento è basato sul **principio di Hebb**:

- Connettere con pesi positivi i neuroni che devono avere la stessa attivazione;
- Connettere con pesi negativi i neuroni che devono avere attivazione diversa;

Il principio di Hebb ha una componente neurobiologica, infatti nel cervello abbiamo legami forti tra le unità che sono attive allo stesso tempo e legami deboli tra le unità che non sono attive simultaneamente.

L'apprendimento avviene in due fasi:

1. **Memorizzazione dell'informazione: regola di definizione dei pesi**

Date M memorie fondamentali $f_1,\dots,f_n$ e dei vettori di dimensione N, memorizziamo le memorie determinando i pesi nel seguente modo:

$$w_{ji} = \frac{1}{M}\sum_{k=1}^{M} f_k(i) * f_k(j) \qquad con\ i \neq j$$

Si considerano tutte le sinapsi tra i e j e per stabilire il valore del peso si fa la media del prodotto di tutte le attivazioni tra i e j. Così facendo se $f_k(i)$ e $f_k(j)$ hanno lo stesso segno, il prodotto sarà positivo e questo contribuirà a rafforzare w_{ji}, viceversa, se hanno segno opposto, ciò contribuirà ad indebolire il legame w_{ji}.

2. **Presentazione degli input corrotti e restituzione dell'output corrispondente**
    1. **Inizializzazione:** presentiamo alla rete una configurazione x* composta da 1 e -1 di lunghezza N. Per ogni neurone j abbiamo $y_j(0)=x^*(j)$
    2. **Iterazione fino alla convergenza:** si sceglie un neurone random e si aggiornano gli elementi della rete secondo la formula:

$$y_j(n) = \varphi(v_j(n)) = \begin{cases} 1 \ se \ v_j > 0 \\ -1 \ se \ v_j < 0 \\ \varphi(v_j(n-1)) \ se \ v_j = 0 \end{cases}$$

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

    3. **Convergenza:** si ripete il passo 2 per ogni neurone fino a quando non troviamo uno stato stabile, cioè fino a quando per ogni unità non risulta $y_j(n+1) = y_j(n)$.
    4. **Output:** lo stato stabile trovato è l'output della rete.

Teorema di convergenza

Dato un qualsiasi stato iniziale della rete, applicando la regola di aggiornamento, la rete converge ad uno stato stabile.

Le idee su cui si basa la prova solo le seguenti:

- Data una rete con N neuroni, abbiamo 2^N stati possibili per la rete.
  - Ad ognuno di questi stati è associato un valore di energia.
  - Si prova che ad ogni cambiamento di stato della rete, l'energia diminuisce.
  - Ci sarà un momento in cui non ci saranno più cambiamenti e quindi si arriva ad uno stato stabile.

L'energia è definita come segue:

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

Ogni prodotto $w_{ij}y_iy_j$ è positivo se:

- y_i e y_j hanno lo stesso segno e w_{ij} è positivo
- y_i e y_j hanno segno opposto e w_{ij} è negativo

L'energia misura il livello di disarmonia della rete, quindi il numero di situazioni instabili. Più è alto il valore dell'energia, più disarmonia ci sarà e di conseguenza la rete è più propensa a cambiamenti. Si prova che E diminuisce ad ogni cambiamento di stato.

Consideriamo come E cambia (ΔE) con il cambiamento dell'attivazione della k-esima unità y_k (E denota l'energia prima del cambiamento ed E* denota l'energia dopo il cambiamento di y_k):

$$E = -\frac{1}{2}\left(\sum_{i\neq k}\sum_{j\neq k} w_{ij}y_iy_j + 2\sum_{j\neq k} w_{kj}y_ky_j\right) = -\frac{1}{2}\sum_{i\neq k}\sum_{j\neq k} w_{ij}y_iy_j - \sum_{j\neq k} w_{kj}y_ky_j$$

$$E^* = -\frac{1}{2}\left(\sum_{i\neq k}\sum_{j\neq k} w_{ij}y_iy_j + 2\sum_{j\neq k} w_{kj}y_k^*y_j\right) = -\frac{1}{2}\sum_{i\neq k}\sum_{j\neq k} w_{ij}y_iy_j - \sum_{j\neq k} w_{kj}y_k^*y_j$$

$$\Delta E = E - E^* = -\frac{1}{2}\sum_{i\neq k}\sum_{j\neq k} w_{ij}y_iy_j - \sum_{j\neq k} w_{kj}y_ky_j + \frac{1}{2}\sum_{i\neq k}\sum_{j\neq k} w_{ij}y_iy_j + \sum_{j\neq k} w_{kj}y_k^*y_j$$

$$= -\sum_{j\neq k} w_{kj}y_ky_j + \sum_{j\neq k} w_{kj}y_k^*y_j = -\sum_{j\neq k} w_{kj}y_j(y_k - y^* - k)$$

A questo punto è possibile distinguere due casi:

1. y_k passa da +1 a -1 quindi:
   - $yk - y^*_k > 0$
   - $\sum_j w_{kj}y_j < 0$

   Allora $-\sum_{j\neq k} w_{kj}y_j(y_k - y_k^*) > 0$ e $E > E^*$ $(\Delta E > 0)$
2. y_k passa da -1 a -1 quindi:
   - $y_k - y^*_k < 0$
   - $\sum_j w_{kj}y_j > 0$

   Allora $-\sum_{j\neq k} w_{kj}y_j(y_k - y_k^*) > 0$ e $E > E^*$ $(\Delta E > 0)$

In sostanza ad ogni cambiamento di stato, E diminuisce. Ad un certo punto si raggiungerà uno stato stabile.

Pregi delle HN

- **Buona capacità di generalizzazione:** dato un input simile ad una memoria fondamentale la rete è in grado di recuperare la corrispondente informazione;
- **Sono tolleranti ai malfunzionamenti:** se alcune sinapsi si rompono, la rete è in grado di fornire risultati in output;
- L'apprendimento è basato sulla regola di Hebb, che è biologicamente plausibile e ci sono evidenze che esista anche nel cervello;
- **È in grado di generalizzare:** dato un input simile a ciò che è stato memorizzato è in grado di recuperare l'informazione corrispondente
- Sono in grado di completare pattern parziali;

Difetti delle HN

- Non tutti gli stati stabili sono memorie fondamentali, infatti ci sono anche stati spuri
    - L'opposto di uno stato stabile, è stabile;
    - Combinazioni di stati stabili sono stati stabili
    - Non tutte le memorie fondamentali sono stabili
- Capacità di immagazzinamento limitata;
- Possibili errori;
- Nel cervello non ci sono sinapsi simmetriche;
- Nel cervello non ci sono stati stabili, ma stati transitoriamente stabili che poi evolvono in stati successivi. Esistono comunque reti di Hopfield che apprendono sequenze di stati;

Per avere pochi errori, dati N neuroni, conviene avere 0.14N memorie fondamentali. Per raggiungere il tasso di errore minimo per N neuroni conviene avere $\frac{N}{2\log N}$ memorie fondamentali.

Macchine di Boltzmann

Le macchine di Boltzmann rappresentano un'evoluzione delle reti di Hopfield e nascono con lo scopo di ovviare alle problematiche delle reti di Hopfield come la presenza di stati spuri e il limite di memorizzazione di memorie fondamentali che è pari a 0.14N.

Caratteristiche

Le macchine di Boltzmann si differenziano dalle reti di Hopfield per due caratteristiche:

1. Per ovviare al problema dei minimi locali si utilizzano neuroni stocastici che possono assumere due stati: 0 o 1, la loro attivazione è probabilistica e non deterministica.
2. Sono composte da due unità:
    a. **Unità visibili:** sono le unità a cui presentiamo gli input e ci restituiscono l'output.
    b. **Unità hidden:** forzano la rete a convergere ad un determinato stato. Servono a fornire interpretazione degli input.

Minimi spuri

La limitata capacità di memorizzazione è dovuta all'interazione di stati stabili che portano ad un'ulteriore diminuzione dell'energia della rete generando stati spuri. Per risolvere il problema degli stati spuri Hopfield proponeva una fase di apprendimento e una fase di disapprendimento, nella speranza che la fase di disapprendimento agisse sugli stati spuri. Il problema legato a questa soluzione è che non si sa quanto bisogna disapprendere per avere risultati soddisfacenti. Un'altra soluzione proposta è quella di allenare la rete con la regola del percettrone.

Reti di Hopfield con neuroni stocastici

Un altro problema delle reti di Hopfield è rappresentato dai minimi locali, ossia il fatto che la rete cercando sempre di ridurre l'energia, giunga ad un minimo che è locale. Lo stato che si raggiunge dipende inoltre dal neurone di partenza e non abbiamo la certezza che il minimo trovato sia anche un minimo globale.

Per risolvere questo problema si utilizza una probabilità per decidere il neurone di partenza. Questo può portare ad attraversare zone di energia più alte per arrivare ad un minimo migliore. In particolare, guardiamo se l'energia globale della rete migliorerebbe o peggiorerebbe cambiando lo stato del neurone.

Unità binarie stocastiche

L'attivazione delle unità delle BM è stocastica, mentre nelle HN è deterministica.
Data una combinazione a livello visivo, per decidere se l'unità hidden deve assumere valore 0 o 1 utilizziamo la seguente formula:

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

Dove T è la temperatura e ΔE_i è l'energy gap.

Energy gap

L'energy gap misura il cambiamento di energia al cambiare dello stato dell'unità hidden:

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

Se $p(s_i=1)>0.5$ l'energia migliora, quindi cambiamo i pesi (in pratica l'energia diminuisce)

Se $p(s_i=1)<0.5$ l'energia peggiora, quindi non cambiamo i pesi (l'energia è già al minimo)

Temperatura (simulated annealing)

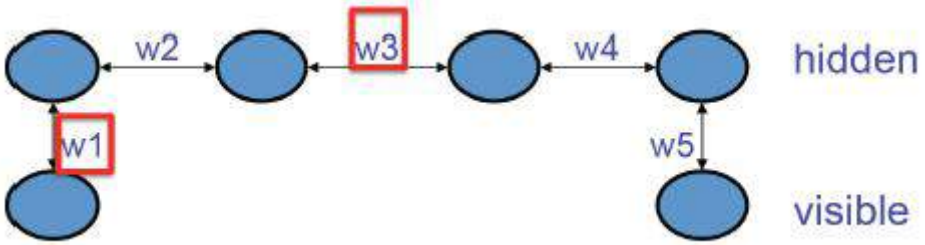
La temperatura controlla la quantità di disarmonia della rete e indica quanto la rete è propensa a cambiamenti. Con una temperatura alta le probabilità sono vicine allo 0.5, quindi si avranno cambiamenti nella rete. Con una temperatura bassa le probabilità saranno lontane da 0.5, quindi si avranno pochi cambiamenti o nessuno.

Inizialmente la temperatura è alta, quindi la rete è propensa a cambiamenti. Successivamente diminuisce fino ad arrivare a cambiamenti di stato minimi. Assumeremo che T=1 e quindi l'energia risulta:

$$E = -\sum_k s_k b_k - \sum_{k<j} s_k s_j w_{kj}$$

Apprendimento

L'obiettivo dell'apprendimento è quello di massimizzare le probabilità che la BM assegna ai vettori binari, in quanto maggiore sarà la probabilità e più migliorerà l'energia. In generale l'apprendimento può essere difficoltoso perché supponendo di avere una catena di unità dove le unità dove le unità visibili sono alla fine, per cambiare efficientemente i pesi dei neuroni visibili dobbiamo conoscere anche i pesi nascosti. Esempio: per cambiare w1 o w5 è necessario conoscere il valore di w3.



Tuttavia tutto quello che dobbiamo sapere degli altri pesi è contenuto nell'equazione che segue:

$$\frac{d \log p(v)}{dw_{ij}} = \langle s_i s_j \rangle_v - \langle s_i s_j \rangle_{model}$$

Derivando il logaritmo della probabilità associata ad un vettore v rispetto al peso otteniamo la differenza tra il prodotto dei valori attesi degli stati all'equilibrio termico dove v è fissato sulle unità visibili e il prodotto del valore atteso degli stati all'equilibrio termico del resto del modello. Quindi la differenza energetica è definita come la differenza tra il prodotto dei valori attesi degli stati all'equilibrio termico fissando un esempio e il prodotto dei valori attesi all'equilibrio termico del resto del modello.

$$\Delta w_{ij} \propto \langle s_i s_j \rangle_{data} - \langle s_i s_j \rangle_{model}$$

Per semplificare l'apprendimento sono state introdotte le RBM.

Restricted Boltzmann Machine (RBM)

Rappresentano una variante delle BM che rendono l'apprendimento più semplice. In particolare hanno due vincoli:

- Un insieme di unità visibili NON connesse tra di loro;
- Un insieme di unità nascoste NON connesse tra di loro;

Unità visibili e unità nascoste sono fortemente connesse tra di loro. II layer nascosto è unico.

Questi vincoli permettono di raggiungere l'equilibrio termico in un solo passo quando le unità visibili sono fissate. È possibile ottenere velocemente il valore $(v_i h_j)_v$.

La probabilità è data da:

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

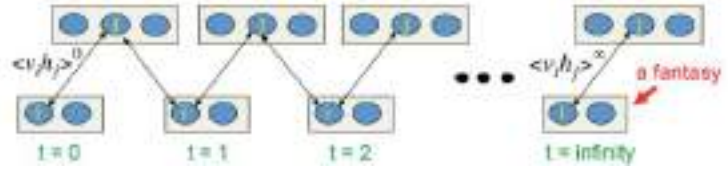
Apprendimento

Dato un input alle unità visibili, si vuole massimizzare la probabilità che la BM lo rappresenti, ossia massimizzare il prodotto delle probabilità che la BM assegna ai vettori del training set. L'idea è quella di studiare lo scarto tra l'input che vogliamo memorizzare e l'output che la macchina preferisce produrre. Per contrastare la tendenza della rete a produrre output che le piacciono si fa in questo modo:

1. Presentiamo l'input alle unità visibili;
2. Si aggiornano alternativamente le unità nascoste e le unità visibili in parallelo;

Questo metodo è altamente inefficiente perché i due passaggi vengono ripetuti all'infinito, secondo la seguente regola:

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

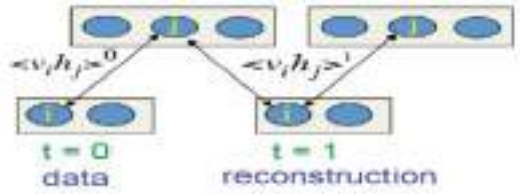


In realtà la rete può essere allenata in due passi:

1. Si presenta l'input alle unità visibili;
2. Si aggiornano tutte le unità nascoste in parallelo;
3. Si aggiornano tutte le unità visibili per ottenere una ricostruzione;
4. Si aggiornano ancora una volta le unità nascoste;

L'equazione di aggiornamento dei pesi è la seguente: $\Delta w_{ij} = \varepsilon (< v_i h_j >^0 - < v_i h_j >^1)$

In questo modo se c'è uno scarto tra i due aggiornamenti dei livelli nascosti è necessario correggere i pesi, in quanto le due ricostruzioni sono differenti; altrimenti non è necessario in quanto le ricostruzioni sono identiche.



Algoritmo contrastive divergence

L'algoritmo si basa su due passaggi:

1) Si presenta il pattern che vogliamo rappresentare a livello visibile;
2) Si calcola l'attivazione del livello nascosto;
3) Si ricalcola l'attivazione a livello visivo a partire dalle unità nascoste;
4) Si ricalcola l'attivazione del livello nascosto;

A questo punto si vede se per ogni coppia visibile-nascosta l'attivazione a livello visibile-hidden corrisponde. Se corrisponde la ricostruzione è la medesima e quindi non dobbiamo aggiornare i pesi.

L'algoritmo permette di osservare la direzione che prende la rete in pochi step verificando la discrepanza tra la prima attivazione del livello nascosto e la seconda. Se la direzione è errata si procede con la diminuzione della probabilità in modo da cambiare strada.

I neuroni nascosti si comportano da features detector.

Deep learning ed RBM

Nell reti deep abbiamo diversi livelli, ognuno dei quali serve a detectare delle features; i livelli successivi combinano features a livello precedente cercando features più complesse. È possibile fare ciò utilizzando delle RMB concatenate tra di loro, in modo che ogni BM catturi combinazioni di features detectate in precedenza.

Prima che ci fossero le reti convoluzionali si volevano delle reti profonde, ma non si sapeva come allenarle in quanto c'era il problema dei minimi locali. Le BM risolvono il problema adottando una fase di pre-training e una fase di fine-tuning.

Pre-training e fine-tuning

Nella fase di p**re-training** addestriamo i livelli a coppie; nella fase di **fine-tuning** si aggiustano leggermente i pesi ottenuti nella fase di pre-training in modo da migliorare le prestazioni della rete.

Abbiamo due modi per fare il pre-training di una rete:

1) **Autoassociatori**
2) **RMB:** considerando le coppie di livelli nascosti come RBM

La tecnica del pre-training permette di ottenere risultati migliori rispetto ad allenare la rete per intero; inoltre gran parte dell'apprendimento è non supervisionato. Solo la parte finale è supervisionata e richiede pochi dati.

Auto-associatori

Queste reti sono state pensate per ridurre la dimensionalità. La nuova dimensionalità non sarà necessariamente inferiore a quella di input.

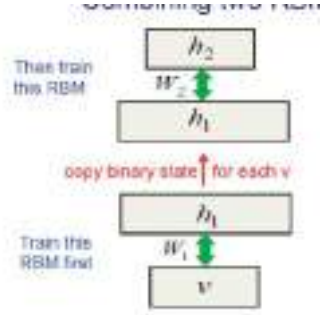
Si allena una porzione della rete profonda in modo che questa impari a ricostruire a livello di uscita lo stesso input ricevuto a livello di entrata. Si considera il livello nascosto come parte di un'architettura più piccola. È importante evitare che la rete impari una funzione identità, cioè che trovi dei pesi che copiano il vettore di input nel layer nascosto e successivamente in output. L'addestramento viene effettuato con la backpropagation standard e viene considerato non supervisionato in quanto l'output atteso e l'input coincidono, quindi non ci sarà un supervisore esterno che ci dirà quale output associare all'input presentato. Ogni livello nascosto cattura diverse rappresentazioni dell'input, sempre più astratte. La speranza è che la fase di pre-training setti i pesi di tutti i livelli in modo tale che si possa raggiungere un buon minimo.

Funzionamento

1. Si allena il primo livello nascosto come un auto-associatore in modo da minimizzare gli errori di ricostruzione dell'input. L'apprendimento è non supervisionato.
2. Le unità di output sono utilizzate come input per un altro livello nascosto, utilizzato come auto-associatore. Anche in questo caso gli esempi non sono etichettati.
3. Si itera lo step (2) per un numero arbitrario di livelli.
4. L'output dell'ultimo livello nascosto sarà l'input di un livello supervisionato e si inizializzano i suoi parametri (in maniera random o supervisionata)
5. Si fa il fine-tuning dei parametri della rete rispetto ai criteri del livello supervisionato.

Stacked RBM

Il pre-training avviene allenando coppie di livelli considerandoli come RBM.



I primi due livelli possono essere visti come una RBM. Si fa in modo che questa rete riconosca tutti gli esempi del training set. Le unità nascoste si specializzano nel riconoscimento di pattern presenti nel livello visibile.

La procedura viene ripetuta per la coppia di livelli successivi. Le attivazioni del livello nascosto della prima RBM vengono dati come input alla seconda RBM. Il livello nascosto della seconda RBM imparerà a riprodurre a livello visivo gli stati di attivazione del livello nascosto della RBM precedente, quindi imparerà pattern e caratteristiche ricorrenti del livello precedente, che sono a loro volta correlate alle caratteristiche del livello precedente (che è quello di input).

Il procedimento viene ripetuto per tutti i livelli. Man mano che si va avanti si trovano combinazioni di features sempre più complesse. Ogni volta che aggiungiamo un livello di features la probabilità di riprodurre gli esempi aumenta.

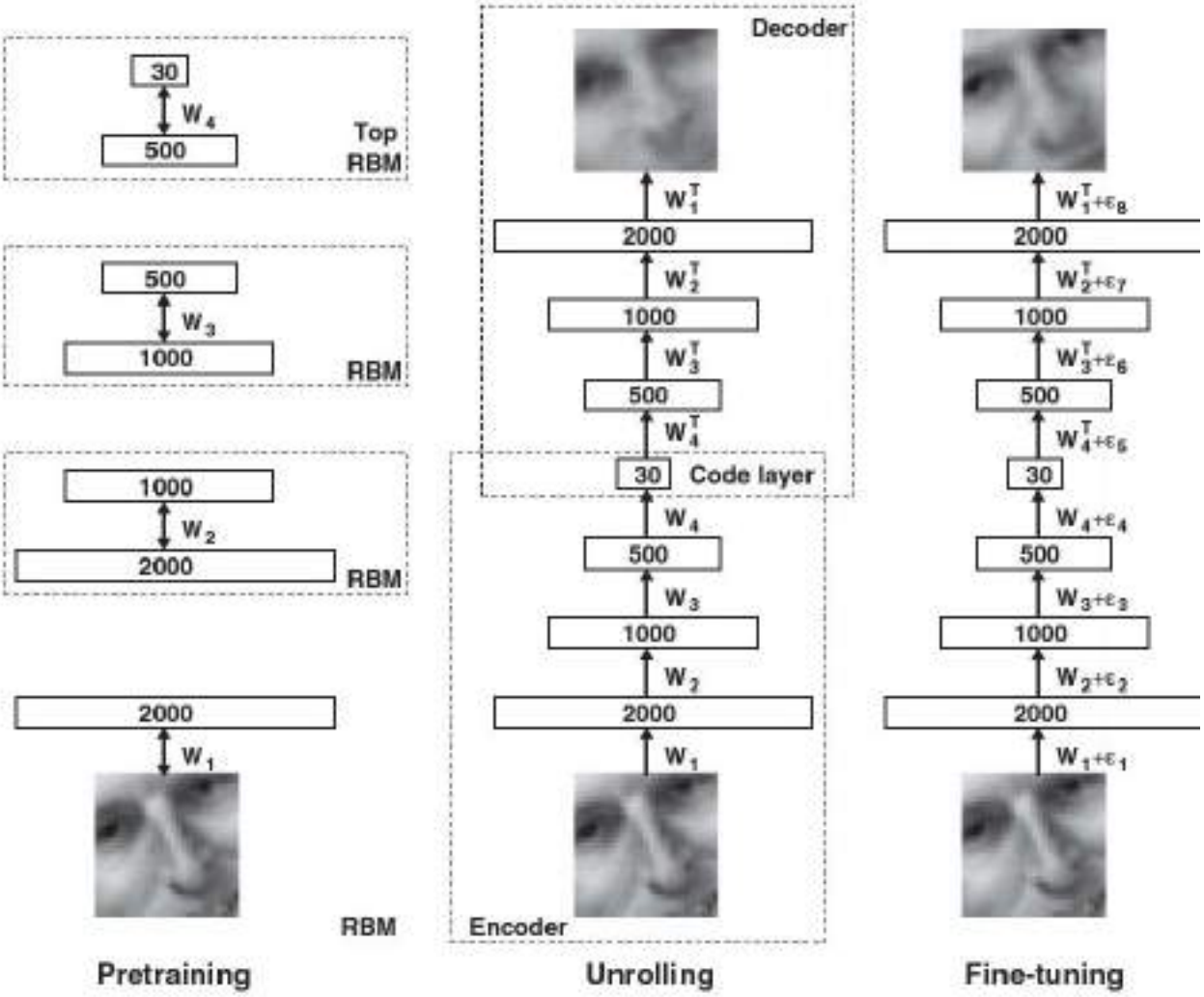
Fine-tuning

A seconda delle applicazioni della rete è possibile adottare diverse tecniche di fine-tuning:

- **Classificazione:** si aggiunge un livello di classificazione utilizzando pesi random. Il fine-tune è effettuato con backpropagation. I pesi sono già settati, vengono leggermente aggiustati per ottenere l'output desiderato.

- **Deep Belief Net (generativo):** per generare immagini a livello di input. L'apprendimento, a differenza delle reti convoluzionali, è non supervisionato. Si parte dalle RBM a livello più alto, si propaga l'attivazione verso il basso e verifichiamo se i dati generati corrispondono ai dati che vogliamo generare.

- **Autoencoders:** utilizzati per cercare una rappresentazione più compattad di un insieme di input.
    - **Pre-training:** l'autoencoder è un insieme di RBM in cui il livello nascosto diventa sempre più piccolo. Le RBM vengono allenate a coppie in modo da specializzarle per determinati task.
    - **Unrolling:** prima si codifica l'immagine utilizzando delle RBM con livelli nascosti sempre pià piccoli e poi si ricostruisce l'immagine utilizzando le RBM al contrario, quindi aumentando man mano il numero di neuroni del livello nascosto.
    - **Fine-tuning:** si aggiustano i pesi in modo da migliorare ulteriormente la rappresentazione dell'immagine.


30
W4
500
Top RBM
500
W3
1000
RBM
1000
W2
2000
RBM
2000
W1
RBM
Pretraining
Decoder
W1T
2000
W2T
1000
W3T
500
W4T
30
Code layer
W4
500
W3
1000
W2
2000
W1
Encoder
Unrolling
W1T+ε8
2000
W2T+ε7
1000
W3T+ε6
500
W4T+ε5
30
W4+ε4
500
W3+ε3
1000
W2+ε2
2000
W1+ε1
Fine-tuning

Deep learning e CNN

Il problema delle reti profonde è legato al fatto che non è possibile passare da un'immagine, intesa come insieme di pixel, alla sua descrizione in un solo passo, in quanto immagini di uno stesso oggetto possono essere molto diverse tra di loro. È quindi necessario passare attraverso più livelli di elaborazione dell'immagine. Questi livelli cercano man mano caratteristiche dell'immagine sempre più astratte tali da poter descrivere l'immagine. In particolare, nei livelli più bassi si cercano features più semplici e man mano che si va avanti nella gerarchia di livelli si cercano features più complesse fino alla classificazione dell'immagine. Le reti vengono allenate con la tecnica del pre-training e successivamente si fa fine-tuning dell'intera rete.

Le reti profonde imparano da sole le features rilevanti per classificare correttamente un'immagine. Precedentemente questo lavoro veniva fatto a mano.

Deep RBM vs CNN

Le reti convoluzionali a differenza degli auto-encoders e delle deep belief network sono supervisionate e hanno bisogno di molti dati. Ciò si avvicina di più agli aspetti cognitivi.

Hubel e Wiesel hanno scoperto a quali stimoli visivi reagivano i neuroni dei gatti, in particolare hanno osservato che questi reagivano a bordi che si muovevano in una certa direzione. Ciò significa che:

- L'immagine viene scomposta in features di base che vengono ricombinate a livelli successivi fino al riconoscimento.
- Neuroni vicini processano zone vicine del campo visivo.

Queste scoperte sono alla base delle reti profonde.

Architetture Deep

1. **Neurocognitron (Fukushima, 1980):** è un'architettura a sandwich inspirata al modello proposto a Hubel e Wiesel in cui c'è un'alternanza tra cellule semplici, che riconoscono features di base; e cellule complesse, che ricombinano le features di base trovate. I pesi dovevano essere disegnati a mano.
2. **LeNet-5 (LeCun, 1998):** è un'architettura simile al Neurocognitron, ma è in grado di imparare i pesi da sola grazie alla backpropagation. È stata introdotta da LeCun ed è utilizzata per il riconoscimento di caratteri scritti a mano.
3. **AlexNet (Hinton, 2012):** è l'architettura che ha decretato il successo delle reti profonde. Sostanzialmente fa le stesse cose della rete di LeCun, ma con immagini più complesse. La rete è stata allenata su *imageNet,* un dataset di immagini etichettare e raggruppate in categorie. Le performance sono valutare su 5 categorie: se la predizione appartiene alla categoria, l'immagine è classificata correttamente.

Architettura AlexNet

L'architettura è organizzata in maniera gerarchica:

- **Livello convoluzionale:** è il livello in cui vengono estratte le features dall'immagine utilizzando un filtro di dimensioni minori rispetto all'immagine (ad esempio 5x5x3). Il filtro è un insieme di pesi e rappresenta una determinata caratteristica. Ogni filtro determina una activation map, ossia un insieme di neuroni che si attivano alla presentazione di una determinata features. I pesi all'interno della features map sono condivisi, quindi i neuroni all'interno della stessa features map cercano le stesse features in diverse parti dell'immagini. Avremo diverse activation map, ognuna delle quali rileverà una determinata feature. Alla fine del primo livello l'immagine non esiste più, in quanto è stata scomposta in un insieme di features.
    - **Funzione di attivazione:** la funzione di attivazione utilizzata nei livelli convoluzionali è la funzione ReLU, che per input negativi o nulli restituisce 0 e per input positivi si attiva in maniera lineare. L'uso della ReLU permette di risolvere il problema del gradiente che sparisce.
- **Livello di pooling:** è il livello in cui si identificano features rilevanti indipendentemente dalla loro posizione. Questo permette di avere una rappresentazione della features più piccola e gestibile. Il pooling è fatto su ciascuna activation map in maniera indipendente.
- Gli **ultimi livelli** sono completamente connessi e fungono da classificatore. La descrizione dell'immagine non avrà niente a che fare con l'immagine di partenza, ma è abbastanza buona da consentire al classificatore di classificare correttamente l'immagine. Non si sa quali features vengono estratte in questo livello per consentire la corretta classificazione dell'immagine. Si utilizza una funzione di trasferimento SoftMax composta da un numero s di neuroni, uno per ogni classe, completamente connessi al layer precedente.

L'intera rete è allenata con backpropagation.

Difficoltà, critiche e direzioni della ricerca nelle reti deep

Il problema principale delle reti profonde è legato al fatto che il loro funzionamento è opaco, cioè la rete funziona, ma non si sa cosa succede all'interno della rete. I ricercatori cercano di capire cosa succede all'interno dei vari livelli attraverso vari esperimenti, ad esempio:

- Oscurano parte dell'immagine per capire quali unità e quali livelli si attivano. L'obiettivo è capire come si specializzano le varie unità della rete;
- Effettuano una deconvoluzione per capire quali stimoli fanno attivare maggiormente le unità, quindi si presentano degli stimoli ai vari livelli e si cerca di capire quali di questi fanno attivare maggiormente le unità;

Inoltre non si sa quale sia la dimensione ottimale di queste reti, quindi si parte dalla rete di partenza (tenendo conto delle performance) e vengono eliminati dei livelli in modo da vedere come cambiano le performance. Se le performance sono uguali, allora quella parte della rete è ridondante; se peggiorano significa che quella parte della rete è importante.

Oltre ad essere opache, le reti profonde possono essere facilmente ingannate, cioè è possibile costruire immagini senza significato per ingannare la rete. Questo ci dice che il modo in cui le reti deep processano l'informazione è diverso dal modo in cui le processa il cervello. In particolare, Ulmann nel 2016 ha dimostrato che le reti profonde non processano le immagini come le processiamo noi umani. Si è visto infatti che gli umani riescono a riconoscere un'immagine a partire da piccole porzioni dell'immagine stessa; le reti profonde invece non ci riescono.

Inoltre le reti deep richiedono una grande quantità di dati e di tempo per essere allenate. Ciò si contrappone all'apprendimento degli umani, che apprendono in pochissimo tempo.

Per questo motivo c'è un filone della ricerca che cerca di creare reti in grado di classificare immagini dopo una sola esposizione (o addirittura nessuna) al nuovo oggetto o immagine che sia, come ad esempio Google Deep Mind.

Combinazione di reti

- **Hopfiled + reti profonde:** visto che le reti deep fanno fatica a fare pattern completion si utilizza una reti di Hopfield prima della rete deep in modo tale da riconoscer e ricostruire immagini corrotte.
- **SOM + reti profonde:** i bambini apprendono una parola dopo una singola esposizione, ma come fanno a capire l'oggetto a cui si riferisce?
    - **Whole Object Constraint:** una parola è associata a tutto l'oggetto anziché alle sue parti.
    - **Taxonomic Constraint:** dopo una singola esposizione di oggetto-parola, il modello genera le associazioni tra tutti gli oggetti correlati all'oggetto di partenza e le parole correlate alla parola di partenza.

Reti ricorrenti

Una rete ricorrente è un modello neurale per il processing di sequenze di dati, cioè dati che hanno un legame temporale tra di loro, quindi testi, parlato, video, finanza ecc…

In queste reti viene introdotto il concetto di ciclo e di tempo. Inoltre una rete ricorrente è in grado di riconoscere la posizione di un parametro all'interno di una sequenza.

Architettura

Le reti ricorrenti possono essere viste come reti feed-forward in cui ad un certo punto ci sarà un punto di ritorno. L'obiettivo è ereditare l'istanza al tempo t per predire l'istanza al tempo t+1. Le istanze al tempo t sono legate alle istanze al tempo t+1, non solo in input ma anche in output.

Queste reti sono allenate con un variabile della back-propagation che tiene conto del tempo ed è detta back-propagation through time.

Grafo computazionale e unfolding

Il grafo computazionale è un modo di astrarre una rete. Il processo di unfolding consiste nello srotolare la rete lungo il tempo. In questo modo il significato dell'input dipende dal contesto in cui si trova.

Backpropagation through time

È un'estensione della backpropagation standard. Esegue la discesa del gradiente su una rete unfolded. Data una sequenza di training su un intervatto t0-t1, la funzione di loss consiste nella somma di tutte le loss comprese nell'intervallo t0-t1. La loss migliore è la cross entropy.

Sparizione del gradiente

Per sequenze di training molto lunghe il gradiente esplode e può sparire o non essere realistico. Quindi per sequenze lunghe non c'è aggiornamento. Il problema è risolto troncando la sequenza in input ed eseguire la backpropagation sulle sottosequenze.

Un altro modo per risolvere il problema è fare il clipping del gradiente. In pratica il gradiente è normalizzato e ridotto ad un certo valore ϵ. Il problema di questa soluzione è legato a fatto che il gradiente non è quello reale perché è stato normalizzato, tuttavia è un buon compromesso.

Long short term memory (LSTM)

È un modello complesso di rete ricorrente arginare il problema della sparizione del gradiente. Si introduce il concetto di memoria all'interno dei parametri. Ad un certo punto la rete inizierà a dimenticare parte di ciò che è successo in passato e questo permette di evitare la sparizione del gradiente che dipende proprio dalla lunghezza della sequenza.