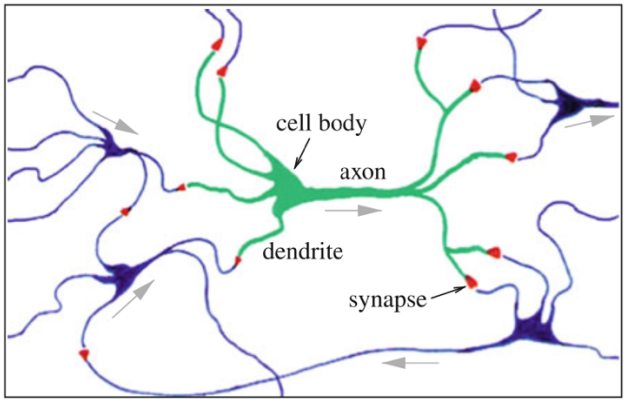
Intelligenza Artificiale – 29 maggio 2023

Reti Neurali

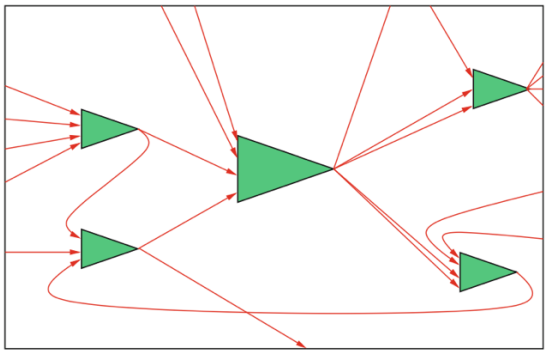
Le reti neurali sono reti di cellule nervose nel cervello di esseri umani e animali. Il cervello umano ha circa 100 miliardi di cellule nervose. Noi esseri umani dobbiamo la nostra intelligenza e la nostra capacità di apprendere varie capacità motorie e intellettuali ai complessi relè e all'adattabilità del cervello. Per molti secoli biologi, psicologi e medici hanno cercato di capire come funziona il cervello. Intorno al 1900 arrivò la rivoluzionaria consapevolezza che questi minuscoli mattoni fisici del cervello, le cellule nervose e le loro connessioni, sono responsabili della consapevolezza, delle associazioni, dei pensieri, della coscienza e della capacità di apprendere.
Il primo grande passo verso le reti neurali nell'IA fu compiuto nel 1943 da McCulloch e Pitts, in un articolo intitolato "*A logic calculus of the ideas immanent in nervous activity*". Sono stati i primi a presentare un modello matematico del neurone come elemento di commutazione di base del cervello. Questo articolo ha gettato le basi per la costruzione di reti neurali artificiali e quindi per questo ramo molto importante dell'IA.
Potremmo considerare il campo della modellazione e simulazione delle reti neurali come il ramo *bionico* all'interno dell'AI. (Bionica: disciplina che mira a convertire le "scoperte della natura vivente" in tecnologia innovativa.) Quasi tutte le aree dell'IA tentano di ricreare processi cognitivi, come nella logica o nel ragionamento probabilistico. Tuttavia, gli strumenti utilizzati per la modellazione, vale a dire matematica, linguaggi di programmazione e computer digitali, hanno ben poco in comune con il cervello umano. Con le reti neurali artificiali, l'approccio è diverso. Partendo dalla conoscenza della funzione delle reti neurali naturali, tentiamo di modellarle, simularle e persino ricostruirle sotto forma di hardware. Ogni ricercatore in questo settore affronta l'affascinante ed entusiasmante sfida di confrontare i risultati con le prestazioni degli esseri umani.

Ciascuno dei circa 100 miliardi di neuroni in un cervello umano ha, come mostrato in una rappresentazione semplificata nella figura sotto, la seguente struttura e funzione.



Oltre al corpo cellulare, il neurone ha un assone, che può stabilire connessioni locali con altri neuroni sui dendriti. L'assone può, tuttavia, crescere fino a un metro di lunghezza sotto forma di fibra nervosa attraverso il corpo.
Il corpo cellulare del neurone può immagazzinare piccole cariche elettriche, in modo simile a un condensatore o una batteria. Questa memoria viene caricata da impulsi elettrici in arrivo da altri neuroni. Maggiore è l'impulso elettrico, maggiore è la tensione. Se la tensione supera

una certa soglia, il neurone si attiverà. Ciò significa che scarica il suo deposito, in quanto invia un picco sull'assone e sulle sinapsi. La corrente elettrica si divide e raggiunge molti altri neuroni sulle sinapsi, in cui avviene lo stesso processo.
Ora sorge la questione della struttura della rete neurale. Ciascuno dei circa 10^{11} neuroni nel cervello è collegato a circa 1000-10000 altri neuroni, il che produce un totale di oltre 10^{14} connessioni. Se consideriamo inoltre che questo numero gigantesco di connessioni estremamente sottili è costituito da tessuto morbido e tridimensionale e che gli esperimenti sul cervello umano non sono facili da eseguire, allora diventa chiaro il motivo per cui non abbiamo uno schema circuitale dettagliato del cervello. Presumibilmente non saremo mai in grado di comprendere completamente lo schema circuitale del nostro cervello, basandoci unicamente sulle sue immense dimensioni.



Dalla prospettiva attuale, non vale più nemmeno la pena di provare a fare uno schema circuitale completo del cervello, perché la struttura del cervello è adattativa. Cambia e si adatta in base alle attività dell'individuo e alle influenze ambientali. Il ruolo centrale qui è svolto dalle sinapsi, che creano la connessione tra i neuroni. Nel punto di connessione tra due neuroni, è come se due cavi si incontrassero. Tuttavia, i due conduttori non sono perfettamente connessi, piuttosto c'è un piccolo spazio, che gli elettroni non possono scavalcare direttamente. Questo vuoto è riempito con sostanze chimiche, i cosiddetti neurotrasmettitori. Questi possono essere ionizzati da una tensione applicata e quindi trasportare una carica sullo spazio vuoto. La conducibilità di questo gap dipende da molti parametri, ad esempio la concentrazione e la composizione chimica del neurotrasmettitore. È illuminante sapere che la funzione del cervello reagisca in modo molto sensibile ai cambiamenti di questa connessione sinaptica, ad esempio attraverso l'influenza di alcol o altre droghe. Come funziona l'apprendimento in una rete neurale di questo tipo? La cosa sorprendente qui è che non sono le unità attive effettive, vale a dire i neuroni, ad essere adattive, ma piuttosto le connessioni tra di loro, cioè le sinapsi. In particolare, questo può modificare la loro conduttività. Sappiamo che una sinapsi è resa più forte da quanta più corrente elettrica deve trasportare. Più forte qui significa che la sinapsi ha una conduttività maggiore. Le sinapsi utilizzate spesso acquistano un peso sempre maggiore. Per le sinapsi che vengono utilizzate di rado o non sono affatto attive, la conduttività continua a diminuire. Questo può persino portarle a morire.
Tutti i neuroni nel cervello funzionano in modo asincrono e in parallelo, ma, rispetto a un computer, a velocità molto bassa. Il tempo per un impulso neurale dura circa un millisecondo, esattamente lo stesso del tempo per il trasporto degli ioni attraverso il gap sinaptico. La frequenza di clock del neurone è quindi inferiore a un kilohertz ed è quindi inferiore a quella dei computer moderni di un fattore 10^6. Questo svantaggio, tuttavia, è più che compensato in molti compiti cognitivi complessi, come il riconoscimento delle immagini, dall'alto grado di elaborazione parallela nella rete delle cellule nervose.
La connessione con il mondo esterno avviene tramite neuroni sensori, ad esempio sulla retina degli occhi, oppure tramite cellule nervose con assoni molto lunghi che dal cervello arrivano ai muscoli e quindi possono compiere azioni come il movimento di una gamba.

Tuttavia, non è ancora chiaro come i principi discussi rendano possibile un comportamento intelligente. Proprio come molti ricercatori nel campo delle neuroscienze, tenteremo di spiegare utilizzando simulazioni di un semplice modello matematico come diventano possibili compiti cognitivi, ad esempio il riconoscimento di schemi (pattern).

Il modello matematico

Per prima cosa sostituiamo l'asse temporale continuo con una scala temporale discreta. Il neurone *i* esegue il seguente calcolo in una fase temporale. Il "caricamento" del potenziale di attivazione si ottiene semplicemente sommando i valori di uscita ponderati x_1, ..., x_n di tutte le connessioni in ingresso nella formula

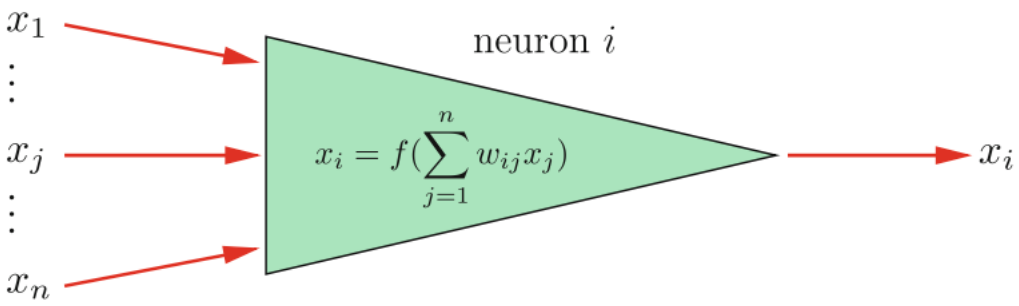
$$\sum_{j=1}^{n} w_{ij} x_j .$$

Quindi viene applicata una funzione di attivazione *f* e il risultato

$$x_i = f\left(\sum_{j=1}^{n} w_{ij} x_j\right)$$

viene trasmesso ai neuroni vicini come output.

Nella seguente figura è mostrato questo tipo di modello di neurone modellato.



La struttura di un neurone formale, che applica la funzione di attivazione f alla somma ponderata di tutti gli input.

Per la funzione di attivazione ci sono una serie di possibilità.
La più semplice è l'identità: f(x) = x. Il neurone calcola quindi solo la somma ponderata dei valori di input e la trasmette. Tuttavia, questo spesso porta a problemi di convergenza con la dinamica neurale perché la funzione f(x) = x è illimitata e i valori della funzione possono crescere nel tempo a dismisura.
Ben limitata, al contrario, è la funzione di soglia (Heaviside step function, da Olivier Heaviside, 1850-1925, matematico, fisico e ingegnere elettrico inglese):

$$H_\Theta(x) = \begin{cases} 0 & \text{if } x < \Theta, \\ 1 & \text{else.} \end{cases}$$

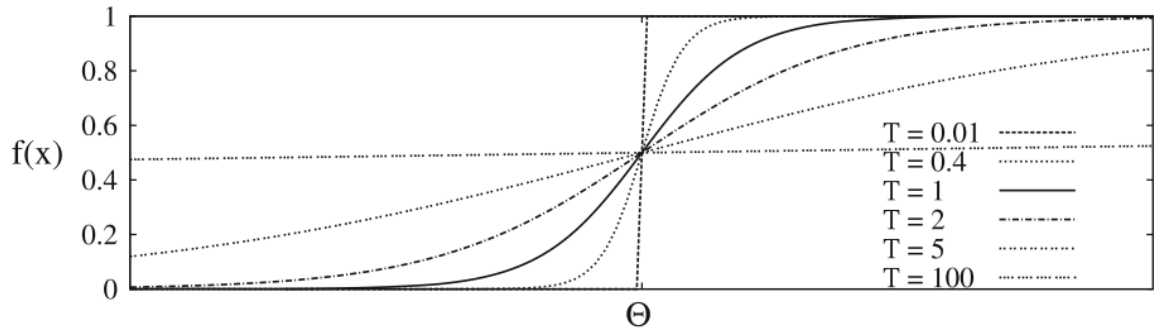
L'intero neurone allora calcola il suo output così:

$$x_i = \begin{cases} 0 & \text{if } \sum_{j=1}^{n} w_{ij} x_j < \Theta, \\ 1 & \text{else.} \end{cases}$$

che è identico a come si comporta un percettrone con soglia Θ.
La funzione soglia ha senso per i neuroni binari perché l'attivazione di un neurone può assumere comunque solo i valori zero o uno. Al contrario, per neuroni continui con attivazioni comprese tra 0 e 1, la funzione soglia crea una discontinuità. Tuttavia, questo può essere attenuato da una funzione sigmoide, come questa:

$$f(x) = \frac{1}{1 + e^{-\frac{x-\Theta}{T}}}$$

In prossimità dell'area critica intorno alla soglia Θ, questa funzione si comporta quasi in modo lineare e ha un limite asintotico. La "smoothness" può essere variata dal parametro T, come mostrato nella seguente figura.



La funzione sigmoide per vari valori del parametro T. Possiamo vedere che nel limite per T che tende a 0 la forma sempre più simile a quella della funzione step.

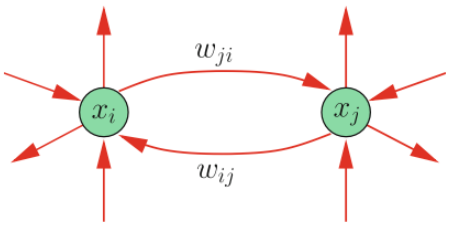
La modellazione dell'apprendimento è fondamentale per la teoria delle reti neurali. Come accennato in precedenza, una possibilità di apprendimento consiste nel rafforzare una sinapsi in base a quanti impulsi elettrici deve trasmettere. Questo principio è stato postulato da Donald Hebb (1904 – 1985, neuropsicologo canadese) nel 1949 ed è noto come la regola di Hebb:
Se esiste una connessione w_{ij} tra il neurone j e il neurone i e vengono inviati segnali ripetuti dal neurone j al neurone i, il che si traduce in entrambi i neuroni attivi contemporaneamente, il peso w_{ij} viene rinforzato. Una possibile formula per la variazione di peso Δw_{ij} è:

$$\Delta w_{ij} = \eta x_i x_j$$

con la costante η (learning rate, tasso di apprendimento) che determina la dimensione delle singole fasi di apprendimento.
Ci sono molte varianti di questa regola, che si traducono in diversi tipi di reti o algoritmi di apprendimento.

Reti di Hopfield

Guardando la regola di Hebb, vediamo che per i neuroni con valori compresi tra zero e uno, i pesi possono solo crescere con il tempo. Non è possibile che un neurone si indebolisca o addirittura muoia secondo questa regola. Questo può essere modellato, ad esempio, da una costante di decadimento che indebolisce un peso inutilizzato di un fattore costante per passo temporale, come 0,99.
Questo problema, invece, è risolto in modo diverso dal modello presentato da Hopfield nel 1982. Utilizza neuroni binari, ma con i due valori -1 per inattivo e 1 per attivo. Utilizzando la regola di Hebb si ottiene un contributo positivo al peso ogni volta che due neuroni sono attivi contemporaneamente. Se, tuttavia, solo uno dei due neuroni è attivo, Δw_{ij} è negativo.
Le reti di Hopfield, che sono un esempio di memoria auto-associativa, si basano su questa idea. I modelli possono essere archiviati nella memoria auto-associativa. Per richiamare un pattern salvato, è sufficiente fornire un pattern simile. Il magazzino di pattern quindi trova il modello salvato più simile. Un'applicazione classica di questo modo di operare è il riconoscimento della grafia.
Nella fase di apprendimento di una rete Hopfield, *N* pattern codificati binari, salvati nei vettori $\mathbf{q}^1$, ..., $\mathbf{q}^N$, devono essere appresi. Ciascun componente $q_i^j \in \{-1; 1\}$ del vettore $\mathbf{q}^j$ rappresenta un pixel di un pattern. Per i vettori costituiti da *n* pixel, viene utilizzata una rete neurale con *n* neuroni, uno per ogni pixel. I neuroni sono completamente connessi con la restrizione che la matrice dei pesi sia simmetrica e che tutti gli elementi diagonali w_{jj} siano zero. Cioè, non c'è connessione tra un neurone e se stesso.
La rete completamente connessa include complessi cicli di feedback, le cosiddette ricorrenze, come mostrato nella seguente figura:



Connessioni ricorrenti tra due neuroni in una rete di Hopfield

È possibile apprendere *N* pattern calcolando semplicemente tutti i pesi con la formula

$$w_{ij} = \frac{1}{N} \sum_{k=1}^{N} q_i^k q_j^k .$$

Questa formula mostra una relazione interessante con la regola di Hebb. Ogni pattern in cui i pixel *i* e *j* hanno lo stesso valore contribuisce positivamente al peso w_{ij}. Ogni altro pattern dà un contributo negativo. Poiché ogni pixel corrisponde a un neurone, qui vengono rinforzati i pesi tra neuroni che hanno lo stesso valore contemporaneamente. Si prega di notare questa piccola differenza con la regola Hebb.

Una volta che tutti i modelli sono stati memorizzati, la rete può essere utilizzata per il riconoscimento dei modelli. Diamo alla rete un nuovo pattern $\boldsymbol{x}$ e aggiorniamo le attivazioni di tutti i neuroni in un processo asincrono secondo la regola

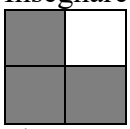
$$x_i = \begin{cases} -1 & \text{if } \sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij} x_j < 0, \\ 1 & \text{else} \end{cases}$$

fino a quando la rete non diventa stabile, cioè fino a quando le attivazioni non cambiano più. Questo modo di operare è espresso dal seguente algoritmo:

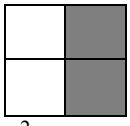
HOPFIELDASSOCIATOR($\boldsymbol{q}$)
Initialize all neurons: $\boldsymbol{x} = \boldsymbol{q}$
Repeat
  i = Random$(1, n)$
  Update neuron i according to
Until $\boldsymbol{x}$ converges
Return $(\boldsymbol{x})$

Esercizio

Insegnare a una rete Hopfield i seguenti pattern:



q^1



q^2

a ogni pixel corrisponde un neurone della rete, quindi avremo una rete con 4 neuroni, ciascuno connesso con tutti gli altri ma non a se stesso, quindi avremo $4^2 - 4 = 12$ connessioni (sinapsi) a cui associamo 12 pesi.
Rappresentiamo i pixel "attivi" (in grigio) con il valore +1 e quelli "inattivi" (in bianco) con -1 e rappresentiamo i 2 pattern in forma vettoriale, leggendone i valori riga per riga, dall'alto verso il basso, da sinistra verso destra:
$q^1 = [\ 1, -1, 1, 1]$
$q^2 = [\ -1, 1, -1, 1]$.

Usiamo la formula per il calcolo dei pesi delle connessioni. L'idea di base è che le connessioni tra neuroni con lo stesso valore pesino positivamente, mentre quella tra neuroni con valori diversi pesino negativamente.

$$w_{ij} = \frac{1}{N} \sum_{k=1}^{N} q_i^k q_j^k .$$

Creiamo la matrice W dei pesi, ricordandoci che le reti di Hopfield non prevedono neuroni connessi a se stessi ($w_{ii}= 0$) e che le connessioni sono simmetriche ($w_{ij} = w_{ji}$)
Ricordando che
$q^1 = [\ 1, -1, 1, 1]$
$q^2 = [\ -1, 1, -1, 1]$
abbiamo che
$w_{11} = 0$
$w_{12} = w_{21} = (q^1{}_1 * q^1{}_2 + q^2{}_1 * q^2{}_2)/2 = (-1\ -1)/2 = -1$
$w_{13} = w_{31} = (q^1{}_1 * q^1{}_3 + q^2{}_1 * q^2{}_3)/2 = (1\ +1)/2 = 1$
$w_{14} = w_{41} = (q^1{}_1 * q^1{}_4 + q^2{}_1 * q^2{}_4)/2 = (1\ -1)/2 = 0$
$w_{22} = 0$
$w_{23} = w_{32} = (q^1{}_2 * q^1{}_3 + q^2{}_2 * q^2{}_3)/2 = (-1\ -1)/2 = -1$
$w_{24} = w_{42} = (q^1{}_2 * q^1{}_4 + q^2{}_4 * q^2{}_2)/2 = (-1\ +1)/2 = 0$
$w_{33} = 0$
$w_{34} = w_{43} = (q^1{}_3 * q^1{}_4 + q^2{}_4 * q^2{}_3)/2 = (1\ -1)/2 = 0$
$w_{44} = 0$

0	-1	1	0
-1	0	-1	0
1	-1	0	0
0	0	0	0

Ora la rete ha "imparato" i 2 pattern q^1 e q^2. L'informazione acquisita non è nel valore assunto dai neuroni, ma nei pesi. I neuroni possono essere settati a nuovi valori in input. La rete risponde a tali valori e li modifica con la regola

$$x_i = \begin{cases} -1 & \text{if } \sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij} x_j < 0, \\ 1 & \text{else} \end{cases}$$

Vediamo che cosa succede quando in input mandiamo un valore x imparato in precedenza dalla rete ($x = q^1$).

$x = [\ 1, -1, 1, 1]$

$x_{1new} = \text{sgn}(w_{12} * x_2 + w_{13} * x_3 + w_{14} * x_4) = \text{sgn}(-1 * -1 + 1 * 1 + 0 * 1) = \text{sgn}(2) = 1$

$x_{2new} = sgn(w_{21}*x_1 + w_{23}*x_3 + w_{24}*x_4) = sgn(-1*1 + -1*1 + 0*1) = sgn(-2) = -1$
$x_{3new} = sgn(w_{31}*x_1 + w_{32}*x_2 + w_{34}*x_4) = sgn(1*1 + -1*-1 + 0*1) = sgn(2) = 1$
$x_{4new} = sgn(w_{41}*x_1 + w_{42}*x_2 + w_{43}*x_3) = sgn(0*1 + 0*-1 + 0*1) = sgn(0) = 1$

$x_{new} = [\ 1, -1, 1, 1]$

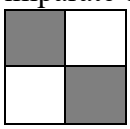
I nuovi valori dei neuroni sono identici a quelli dati in input: la configurazione è stabile.
Metaforicamente parlando, la rete ha "riconosciuto" un pattern che conosce.
Lo stesso vale per $x = q^2$.

$x = [\ -1, 1, -1, 1]$

$x_{1new} = sgn(w_{12}*x_2 + w_{13}*x_3 + w_{14}*x_4) = sgn(-1*1 + 1*-1 + 0*1) = sgn(-2) = -1$
$x_{2new} = sgn(w_{21}*x_1 + w_{23}*x_3 + w_{24}*x_4) = sgn(-1*-1 + -1*-1 + 0*1) = sgn(2) = 1$
$x_{3new} = sgn(w_{31}*x_1 + w_{32}*x_2 + w_{34}*x_4) = sgn(1*-1 + -1*1 + 0*1) = sgn(-2) = -1$
$x_{4new} = sgn(w_{41}*x_1 + w_{42}*x_2 + w_{43}*x_3) = sgn(0*-1 + 0*1 + 0*-1) = sgn(0) = 1$

$x_{new} = [\ -1, 1, -1, 1] = x = q^2$

Vediamo ora che cosa succede quando si dà in input una configurazione diversa da quella imparate dalla rete. Ad esempio la seguente:



x = [1, -1, -1, 1]

Calcoliamo l'output (questa volta con una notazione più visiva, con anche la matrice dei pesi):

0	-1	1	0
-1	0	-1	0
1	-1	0	0
0	0	0	0

Calcolo di x_{1new}:

sgn(0*1+ -1*-1+ 1*-1+ 0*1)=1

-1	0	-1	0
1	-1	0	0
0	0	0	0

Analogamente, calcoliamo gli altri componenti di x_{new}:

sgn(0*1+ -1*-1+ 1*-1+ 0*1)=1

sgn(-1*1+ 0*-1+ -1*-1+ 0*1)=1

sgn(1*1+ -1*-1+ 0*-1+ 0*1)=1

sgn(0*1+ 0*-1+ 0*-1+ 0*1)=1

Attenzione: x_{new} ([1,1,1,1]) è diverso da x. La rete non è stabile. Bisogna proseguire i calcoli e vedere come evolvono i neuroni.

Calcoliamo x_{new2}:

sgn(0*1+ -1*1+ 1*1+ 0*1)= 1

sgn(-1*1+ 0*1+ -1*1+ 0*1)= -1

sgn(1*1+ -1*1+ 0*1+ 0*1)= 1

sgn(0*1+ 0*1+ 0*1+ 0*1)= 1

Otteniamo x_{new2} = [1, -1, 1, 1]. Di nuovo, un output diverso, quindi dobbiamo proseguire con i calcoli.

Calcoliamo x_{new3}:

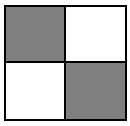
sgn(0*1+ -1*-1+ 1*1+ 0*1)= 1

sgn(-1*1+ 0*-1+ -1*1+ 0*1)= -1

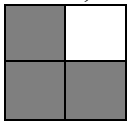
sgn(1*1+ -1*-1+ 0*1+ 0*1)= 1

sgn(0*1+ 0*-1+ 0*1+ 0*1)= 1

Questa volta $x_{new3} = x_{new2}$, ossia la rete ha raggiunto una configurazione stabile (ulteriori iterazioni darebbero gli stessi risultati).
Notiamo che $x_{new3} = x_{new2} = q^1$, ossia avviene quanto segue:
dando in input



la rete, dopo qualche calcolo, converge a:



Si potrebbe interpretare questo comportamento come una correzione da parte della rete. Arriva in input un q^1 con un bit "corrotto", e la rete riesce comunque a riconoscere la configurazione corretta.

Vediamo, però, che cosa succede se diamo in input il seguente pattern:

x = [-1,1,1,-1]

Calcoliamo x_{new}:

sgn(0*-1+ -1*1+ 1*1+ 0*-1)=1

sgn(-1*-1+ 0*1+ -1*1+ 0*-1)=1

sgn(1*-1+ -1*1+ 0*1+ 0*-1)=-1

sgn(0*-1+ 0*1+ 0*1+ 0*-1)=1

x_{new}([1,1,-1,1]) è diverso da x([-1,1,1,-1]).
La rete non è stabile, quindi ripetiamo il calcolo con x_{new} come input:

sgn(0*1+ -1*1+ 1*-1+ 0*1)=-1

sgn(-1*1+ 0*1+ -1*-1+ 0*1)=1

sgn(1*1+ -1*1+ 0*-1+ 0*1)=1

sgn(0*1+ 0*1+ 0*-1+ 0*1)=1

x_{new2} ([-1,1,1,1]) è diverso da x_{new}, quindi ripetiamo ancora:

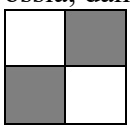
sgn(0*-1+ -1*1+ 1*1+ 0*1)=1

sgn(-1*-1+ 0*1+ -1*1+ 0*1)=1

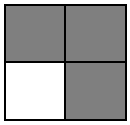
sgn(1*-1+ -1*1+ 0*1+ 0*1)=-1

sgn(0*-1+ 0*1+ 0*1+ 0*1)=1

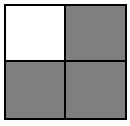
Otteniamo x_{new3}=[1,1,-1,1], ossia torniamo ad avere x_{new}.
Questo vuol dire che la rete continuerà ad oscillare tra x_{new}([1,1,-1,1]) e x_{new2}([-1,1,1,1]) ossia, dando in input



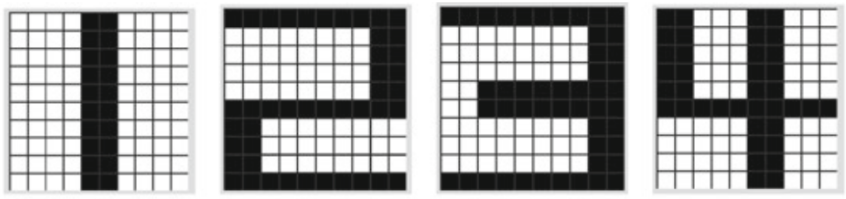
la rete oscilla all'infinito tra



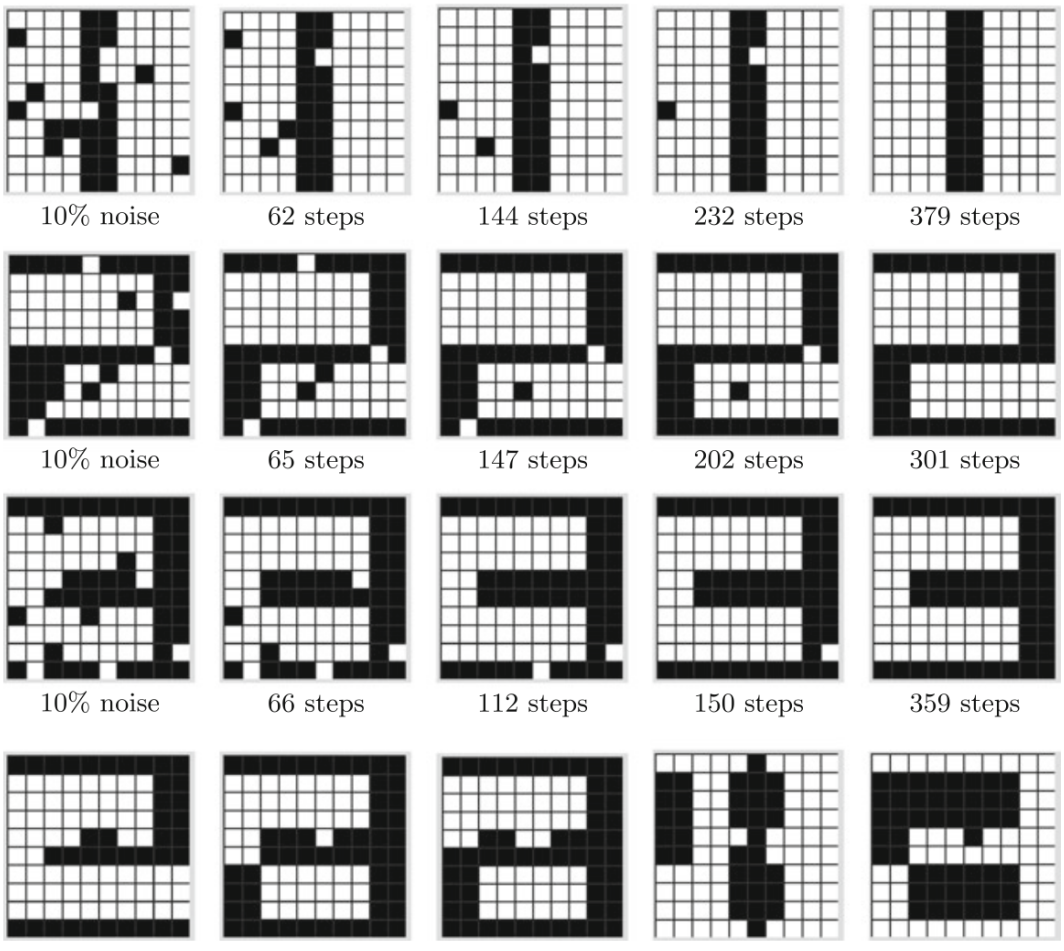
e



Esempio di applicazione a riconoscimento di caratteri.
Per prima cosa vengono addestrati i modelli delle cifre 1, 2, 3, 4. Cioè, i pesi sono calcolati. Quindi inseriamo il pattern con rumore aggiunto e lasciamo che la dinamica di Hopfield funzioni fino alla convergenza. Nelle righe da 2 a 4 della figura, durante il riconoscimento vengono mostrate cinque istantanee dello sviluppo della rete. Al 10% di rumore, tutti e quattro i modelli appresi vengono riconosciuti in modo molto affidabile. Al di sopra di circa il 20% di rumore, l'algoritmo converge spesso ad altri modelli appresi o anche a modelli che non sono stati appresi.



The four training examples learned by the network.



Several stable states of the network which were not learned.

Memorie associative neurali

Una memoria a lista tradizionale può nel caso più semplice essere un file di testo in cui vengono salvate stringhe di cifre riga per riga. Se il file è ordinato per riga, la ricerca di un elemento può essere eseguita molto rapidamente in tempo logaritmico, anche per file molto grandi.

Tuttavia, la memoria a lista può essere utilizzata anche per creare mappature. Ad esempio, una rubrica è una mappatura dalla serie di tutti i nomi immessi alla serie di tutti i numeri di telefono. Questa mappatura è implementata come una semplice tabella, generalmente salvata in un database.
Il controllo dell'accesso a un edificio utilizzando il riconoscimento facciale è un'attività simile. Qui potremmo anche utilizzare un database in cui viene salvata una foto di ogni persona insieme al nome della persona e possibilmente altri dati. La telecamera all'ingresso scatta quindi una foto della persona e cerca nel database una foto identica. Se la foto viene trovata, la persona viene identificata e ottiene l'accesso all'edificio. Tuttavia, un edificio con un tale sistema di controllo non attirerebbe molti visitatori perché la probabilità che la foto corrente corrisponda esattamente alla foto salvata è molto piccola.
In questo caso non è sufficiente salvare la foto in una tabella. Piuttosto, quello che vogliamo è una memoria associativa, che è in grado non solo di assegnare il nome giusto alla foto, ma anche a qualsiasi insieme potenzialmente infinito di foto "simili". Una funzione per trovare la somiglianza dovrebbe essere generata da un insieme finito di dati di addestramento, vale a dire le foto salvate etichettate con i nomi. Un approccio semplice per questo è il metodo del vicino prossimo introdotto in precedenza. Durante l'apprendimento, tutte le foto vengono semplicemente salvate. Per applicare questa funzione è necessario che nel database si trovi la foto più simile a quella attuale. Per un database con molte foto ad alta risoluzione, questo processo, a seconda della metrica della distanza utilizzata, può richiedere tempi di calcolo molto lunghi e quindi non può essere implementato in questa semplice forma. Pertanto, invece di un algoritmo così pigro, preferiremo uno che trasferisca i dati in una funzione che quindi crea un'associazione molto veloce quando viene applicata.
Trovare una metrica di distanza adeguata presenta un ulteriore problema. Vorremmo che una persona fosse riconosciuta anche se il suo volto appare in un altro punto della foto (traslazione), o se è più piccola, più grande o addirittura ruotata. Anche l'angolo di visione e l'illuminazione potrebbero variare.
È qui che le reti neurali mostrano i loro punti di forza. Senza richiedere allo sviluppatore di pensare a una metrica di somiglianza adeguata, forniscono comunque buoni risultati.
Introdurremo due dei più semplici modelli di memoria associativa e inizieremo con un modello di Teuvo Kohonen, uno dei pionieri in questo settore.
Il modello Hopfield presentato nel capitolo precedente sarebbe troppo difficile da usare per due ragioni. Primo, è solo una memoria auto-associativa, cioè una mappatura approssimativamente identica che mappa oggetti simili all'originale appreso. In secondo luogo, le complesse dinamiche ricorrenti sono spesso difficili da gestire nella pratica.
Pertanto ora esamineremo semplici reti feedforward a due strati.

Memoria con matrice di correlazione

Kohonen ha introdotto un modello di memoria associativa basato sull'algebra lineare elementare. Questo mappa i vettori di input $\mathbf{q} \in R^n$ ai vettori di output $\mathbf{t} \in R^m$. Stiamo cercando una matrice **W** che mappi correttamente, basandosi su un insieme di dati di addestramento

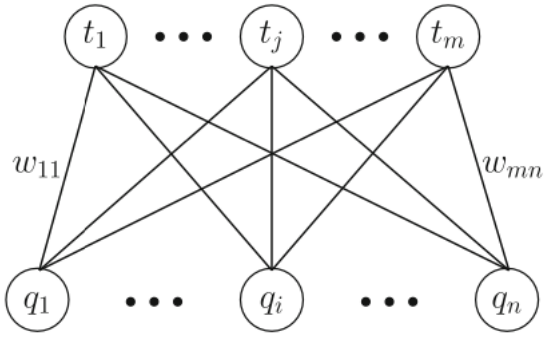
$$T = \{(\boldsymbol{q}^1, \boldsymbol{t}^1), \ldots, (\boldsymbol{q}^N, \boldsymbol{t}^N)\}$$

con N coppie input (query, q) – output (target, t), tutti i vettori di query alle rispettive risposte. Cioè, per p = 1, ..., N deve essere:

$$\boldsymbol{t}^p = \boldsymbol{W} \cdot \boldsymbol{q}^p$$

o, scritto in maniera diversa,

$$t_i^p = \sum_{j=1}^{n} w_{ij} q_j^p .$$



Rappresentazione della memoria associativa Kohonen come una rete neurale a due livelli (layers).

Per calcolare i valori w_{ij} della matrice **W**, usiamo la regola:

$$w_{ij} = \sum_{p=1}^{N} q_j^p t_i^p$$

Queste due equazioni lineari possono essere semplicemente intese come una rete neurale se definiamo, come mostrato nella figura sopra, una rete a due livelli con **q** come livello di input e **t** come livello di output. I neuroni del livello di output hanno una funzione di attivazione lineare e la regola di apprendimento utilizzata, quella per stabilire i pesi delle connessioni, corrisponde esattamente alla regola di Hebb.

Teorema
Se tutti gli N vettori di query $\mathbf{q^p}$ nei dati di addestramento sono ortonormali, ogni vettore $\mathbf{q^p}$ viene mappato al vettore di destinazione $\mathbf{t^p}$ tramite moltiplicazione con la matrice **W** definita con il calcolo dei valori w_{ij} mostrato sopra.

I vettori $\mathbf{q^p}$ sono ortonormali quando per ogni i e per ogni j≠i si ha che:
$\mathbf{q^i q^j} = 0$ e $|\mathbf{q^i}| = |\mathbf{q^j}| = 1$.

Pertanto, se i vettori di query sono ortonormali, tutti gli input verranno mappati correttamente sui rispettivi output. Tuttavia, l'ortonormalità è una restrizione molto forte.
Poiché le mappature lineari sono continue e iniettive, sappiamo che la mappatura dai vettori di query ai vettori di destinazione preserva la somiglianza. Le query simili vengono quindi mappate a obiettivi simili a causa della continuità. Allo stesso tempo sappiamo, tuttavia, che query diverse vengono mappate a target diversi. Se la rete è stata addestrata per mappare i volti ai nomi e se il nome Henry è assegnato a un volto, allora siamo sicuri che per l'input di un volto simile, verrà prodotto un output simile a "Henry", ma la stringa esatta "Henry" non verrà fuori come output perché la mappatura è iniettiva e anche una minima differenza nella

foto in input rispetto alla foto “imparata” deve produrre un output diverso.
Se l’output viene interpretato come una stringa, allora, ad esempio, potrebbe essere “Genry” o “Gfnry”. Per arrivare al caso appreso più simile (“Henry”), Kohonen utilizza una codifica binaria per il neurone di output, e il risultato calcolato di una query viene arrotondato se il suo valore non è 0 oppure 1. Anche in questo caso non abbiamo alcuna garanzia di raggiungere il vettore di destinazione. In alternativa, potremmo aggiungere una successiva mappatura della risposta calcolata al vettore target appreso con la distanza più piccola.

Reti lineari con errori minimi

La regola Hebb utilizzata nei modelli neurali presentati finora funziona con associazioni tra neuroni vicini. Nella memoria associativa, questo viene sfruttato per apprendere una mappatura dai vettori di query ai target. Questo funziona molto bene in molti casi, specialmente quando i vettori di query sono linearmente indipendenti. Se questa condizione non è soddisfatta, ad esempio quando sono disponibili troppi dati di allenamento, sorge la domanda: come troviamo la matrice del peso ottimale? Ottimale significa che riduce al minimo l’errore medio.
Noi umani siamo in grado di imparare dagli errori. La regola di Hebb non offre questa possibilità. L’algoritmo di *backpropagation*, descritto di seguito, utilizza un’elegante soluzione ispirata all’approssimazione di funzioni per *modificare* i pesi in modo tale da ridurre al minimo l’errore sui dati di addestramento.
Sia dato un insieme di N coppie di vettori di allenamento

$$T = \{(\boldsymbol{q}^1, \boldsymbol{t}^1), \ldots, (\boldsymbol{q}^N, \boldsymbol{t}^N)\}$$

con $q^p \in [0, 1]^n$ e $t^p \in [0, 1]^m$. Stiamo cercando una funzione $f:[0, 1]^n \rightarrow [0, 1]^m$ che minimizza l’errore quadratico sui dati:

$$\sum_{p=1}^{N} (f(\boldsymbol{q}^p) - \boldsymbol{t}^p)^2$$

Supponiamo innanzitutto che i dati non contengano contraddizioni. Ovvero, non è presente alcun vettore di query nei dati di addestramento che dovrebbe essere mappato su due obiettivi diversi. In questo caso non è difficile trovare una funzione che minimizzi l’errore quadratico. In effetti, esistono infinite funzioni che rendono l’errore nullo. Definiamo la funzione così:

$$f(\boldsymbol{q}) = 0, \quad \text{if } \boldsymbol{q} \notin \{\boldsymbol{q}^1, \ldots, \boldsymbol{q}^N\}$$

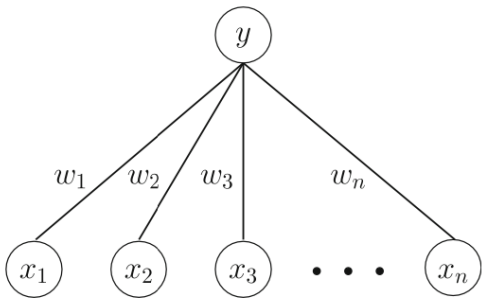
$$f(\boldsymbol{q}^p) = \boldsymbol{t}^p \quad \forall p \in \{1, \ldots, N\}.$$

Questa è una funzione che azzera anche l’errore sui dati di addestramento. Cosa si può volere di più? Perché non siamo soddisfatti di questa funzione?
La risposta è: perché vogliamo costruire un sistema intelligente! “Intelligente” significa, tra le altre cose, che la funzione appresa può generalizzare bene dai dati di addestramento a dati nuovi e sconosciuti dallo stesso set di dati rappresentativo. In altre parole: non vogliamo un overfitting dei dati tramite memorizzazione. Cos’è allora che vogliamo veramente?
Vogliamo una funzione che sia smooth e che “uniformi” lo spazio tra i punti. La continuità e la capacità di poter essere derivata più volte sarebbero requisiti sensati. Poiché anche con

queste condizioni ci sono ancora infinite funzioni che rendono zero l'errore, dobbiamo restringere ulteriormente questa classe di funzioni.

Metodo dei minimi quadrati

La scelta più semplice è una mappatura lineare. Iniziamo con una rete a due strati, come nella seguente figura:



in cui il singolo neurone **y** del secondo strato calcola la sua attivazione utilizzando

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right)$$

dove *f(x) = x*. Il fatto che qui stiamo considerando un solo neurone di output non rappresenta una vera restrizione perché una rete a due strati con due o più neuroni di output può sempre essere separata in reti indipendenti con neuroni di input identici per ciascuno dei neuroni di output originali. I pesi delle sottoreti sono tutti indipendenti. L'utilizzo di una funzione sigmoide al posto dell'attivazione lineare non offre alcun vantaggio in questo caso perché la funzione sigmoide è strettamente monotonicamente crescente e non cambia la relazione d'ordine tra i vari valori di uscita.
Stiamo cercando un vettore **w** di pesi che minimizza l'errore quadratico:

$$E(\boldsymbol{w}) = \sum_{p=1}^{N} (\boldsymbol{w}\,\boldsymbol{q}^p - t^p)^2 = \sum_{p=1}^{N} \left(\sum_{i=1}^{n} w_i {q_i}^p - t^p\right)^2$$

Come condizione necessaria per un minimo di questo errore, tutte le derivate parziali devono essere zero. Quindi richiediamo che per j = 1, ..., n:

$$\frac{\partial E}{\partial w_j} = 2\sum_{p=1}^{N}\left(\sum_{i=1}^{n} w_i q_i^p - t^p\right) q_j^p = 0.$$

Svolgendo le moltiplicazioni otteniamo:

$$\sum_{p=1}^{N}\left(\sum_{i=1}^{n} w_i q_i^p q_j^p - t^p q_j^p\right) = 0,$$

e scambiando le somme si ottiene il sistema lineare di equazioni

$$\sum_{i=1}^{n} w_i \sum_{p=1}^{N} q_i^p q_j^p = \sum_{p=1}^{N} t^p q_j^p,$$

che, con le seguenti definizioni

$$A_{ij} = \sum_{p=1}^{N} q_i^p q_j^p \quad \textit{and} \quad b_j = \sum_{p=1}^{N} t^p q_j^p$$

può essere riscritto nella forma di equazione matriciale **Aw** = **b**.
Queste cosiddette equazioni normali hanno sempre almeno una soluzione e, quando A è invertibile, esattamente una. Inoltre, la matrice A è definita positiva, il che implica che la soluzione scoperta nel caso unico è un minimo globale. Una matrice reale simmetrica A (di forma n×n) è definita positiva quando, per ogni vettore colonna $\mathbf{x} \in R^n$ con $\mathbf{x} \neq 0$, si ha $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$.
Questo algoritmo è noto come *metodo dei minimi quadrati*.

La regola del Delta

I minimi quadrati sono, come il percettrone e l'apprendimento dell'albero decisionale, un cosiddetto algoritmo di apprendimento in batch, al contrario dell'apprendimento incrementale. Nell'apprendimento "*batch*": tutti i dati di addestramento devono essere appresi in un'unica esecuzione. Se vengono aggiunti nuovi dati di allenamento, non possono essere semplicemente appresi in aggiunta a ciò che è già presente. L'intero processo di apprendimento deve essere ripetuto con il set ingrandito. Questo problema viene risolto da algoritmi di apprendimento *incrementale*, che possono adattare il modello appreso a ogni nuovo esempio aggiuntivo. In quanto segue, aggiorneremo in modo additivo i pesi per ogni nuovo esempio di allenamento secondo la regola:

$$w_j = w_j + \Delta w_j.$$

Per derivare una variante incrementale del metodo dei minimi quadrati, riconsideriamo le n derivate parziali della funzione di errore calcolate prima:

$$\frac{\partial E}{\partial w_j} = 2\sum_{p=1}^{N}\left(\sum_{i=1}^{n} w_i q_i^p - t^p\right) q_j^p$$

Il gradiente

$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right)$$

sotto forma di vettore di tutte le derivate parziali della funzione di errore punta nella direzione del più forte aumento della funzione di errore nello spazio n-dimensionale dei pesi. Nella ricerca del minimo, seguiremo quindi la direzione del gradiente negativo. Come formula per modificare i pesi otteniamo allora

$$\Delta w_j = -\frac{\eta}{2}\frac{\partial E}{\partial w_j} = -\eta\sum_{p=1}^{N}\left(\sum_{i=1}^{n} w_i q_i^p - t^p\right) q_j^p$$

dove il tasso di apprendimento η è una costante positiva liberamente selezionabile. Un η maggiore accelera la convergenza ma allo stesso tempo aumenta il rischio di oscillazione attorno ai minimi o alle valli piane. Pertanto, la scelta ottimale di η non è un compito semplice. Un grande η, ad esempio η = 1, è spesso usato per iniziare, e poi si diminuisce gradualmente.
Considerando che

$$y^p = \sum_{i=1}^{n} w_i q_i^p$$

cioè il p-esimo output y^p è calcolato moltiplicando per i pesi w_i l'input di addestramento q^p, la formula è semplificata e otteniamo la *regola del delta*

$$\Delta w_j = \eta\sum_{p=1}^{N}(t^p - y^p) q_j^p .$$

Ossia, per ogni esempio di addestramento, la differenza tra l'obiettivo t^p e l'output effettivo y^p della rete viene calcolata per il dato input q^p. Dopo aver sommato i risultati di tutti gli N pattern, i pesi vengono quindi modificati proporzionalmente a tale somma. L'algoritmo è riassunto nella seguente figura:

DELTALEARNING(*TrainingExamples*, η)
Initialize all weights w_j randomly
Repeat
  $\Delta \boldsymbol{w} = \mathbf{0}$
  For all $(\boldsymbol{q}^p, t^p) \in$ *TrainingExamples*
    Calculate network output $y^p = \boldsymbol{w}^p \boldsymbol{q}^p$
    $\Delta \boldsymbol{w} = \Delta \boldsymbol{w} + \eta(t^p - y^p)\boldsymbol{q}^p$
  $\boldsymbol{w} = \boldsymbol{w} + \Delta \boldsymbol{w}$
Until $\boldsymbol{w}$ converges

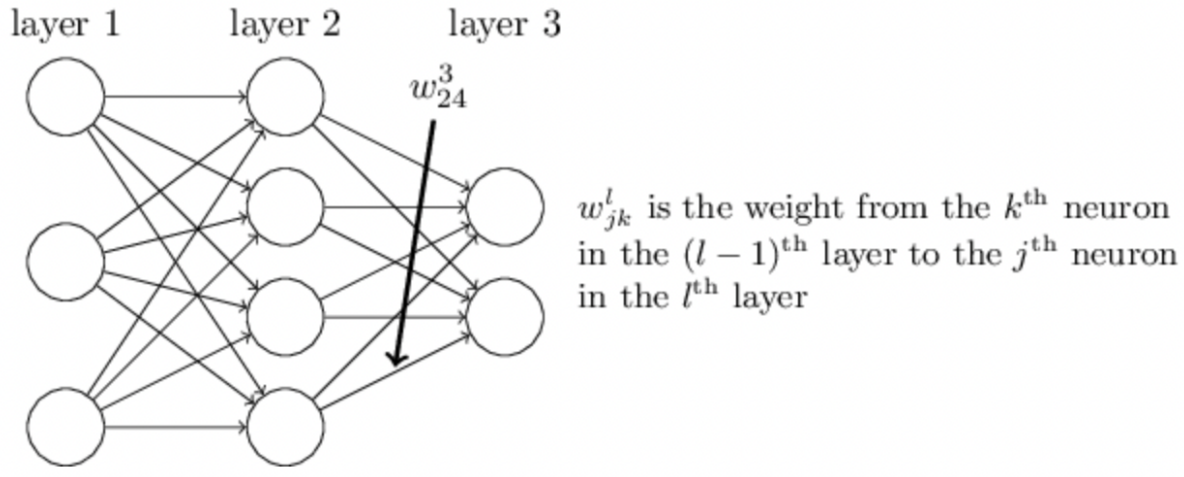
A voler essere precisi, notiamo che l'algoritmo non è ancora realmente incrementale perché i cambiamenti di peso si verificano solo dopo che tutti gli esempi di allenamento sono stati applicati una volta. Possiamo correggere questa carenza modificando direttamente i pesi (discesa gradiente incrementale) dopo ogni esempio di allenamento, il che, in senso stretto, non è più una corretta implementazione della regola del delta.

DELTALEARNINGINCREMENTAL(*TrainingExamples*, η)
Initialize all weights w_j randomly
Repeat
  For all $(\boldsymbol{q}^p, t^p) \in$ *TrainingExamples*
    Calculate network output $y^p = \boldsymbol{w}^p \boldsymbol{q}^p$
    $\boldsymbol{w} = \boldsymbol{w} + \eta(t^p - y^p)\boldsymbol{q}^p$
Until $\boldsymbol{w}$ converges

Reti neurali e backpropagation

Introduciamo ora il modello neurale più utilizzato, quello della rete di backpropagation (retro-propagazione), a più livelli di neuroni. La ragione del suo uso diffuso è la sua versatilità universale per compiti di approssimazione arbitraria. L'algoritmo ha origine direttamente dalla regola del delta incrementale. Contrariamente alla delta rule, applica una funzione sigmoide non lineare sulla somma ponderata degli ingressi come funzione di attivazione. L'algoritmo, concepito negli anni '70, è diventato noto attraverso l'articolo di Rumelhart et al. nel 1986.

Cominciamo con una notazione che ci consente di fare riferimento ai pesi nella rete in modo univoco. Useremo w^l_{jk} per denotare il peso per la connessione dal k-esimo neurone nel (l – 1)-esimo livello al j-esimo neurone nel livello l-esimo. Quindi, ad esempio, il diagramma seguente mostra il peso su una connessione dal quarto neurone nel secondo livello al secondo neurone nel terzo livello di una rete:



Nel modello generale di un neurone artificiale, oltre agli stimoli in input provenienti dai neuroni in ingresso, si aggiunge anche un *bias*, ossia una costante additiva caratteristica del neurone stesso. Il neurone *j* sarà caratterizzato da un bias b_j che va a sommarsi agli stimoli in ingresso per formare l'input per la funzione di attivazione. Se specifichiamo il livello *l* a cui si trova il neurone, diremo che il neurone *j* del livello *l* ha bias b^l_j.

L'equazione di base è quindi la seguente: l'attivazione del neurone *j* al livello *l*, ossia il suo output a^l_j, è dato dalla funzione di attivazione σ (sigma, perché tipicamente e tradizionalmente si tratta di una funzione sigmoidale) applicata alla sommatoria degli stimoli $a^{l-1}{}_k$ provenienti dal livello precedente (*l-1*), ciascuno pesato secondo il peso w^l_{jk} associato alla connessione tra il *k*-esimo neurone del livello (*l-1*) e il *j*-esimo neurone del livello *l*, a cui si aggiunge il bias b^l_j.

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

Per riscrivere questa espressione in una forma di matrici definiamo una matrice di peso w_l per ogni livello *l*. Le voci della matrice dei pesi w_l sono solo i pesi che si collegano all'*l*-esimo livello di neuroni, ovvero la voce nella *j*-esima riga e nella *k*-esima colonna è w^l_{jk}. Allo stesso modo, per ogni livello *l* definiamo un vettore di bias, b^l. I componenti del vettore di bias sono i valori b^l_j, un componente per ogni neurone nell'*l*-esimo livello. Infine, definiamo un vettore di attivazione a^l i cui componenti sono le attivazioni a^l_j. L'ultimo ingrediente di cui abbiamo bisogno per riscrivere l'equazione in una forma matriciale è l'idea di vettorializzare una funzione come σ. L'idea è che vogliamo applicare una funzione come σ a ogni elemento in un vettore *v*. Usiamo l'ovvia notazione $\sigma(v)$ per denotare questo tipo di applicazione elemento per elemento di una funzione. Cioè, le componenti del vettore $\sigma(v)$ sono $\sigma(v)_j = \sigma(v_j)$. L'equazione viene quindi riscritta nella seguente forma matriciale più compatta.

$$a^l = \sigma(w^l a^{l-1} + b^l).$$

Quando si usa l'equazione per calcolare a^l, calcoliamo la quantità intermedia

$$z^l = w^l a^{l-1} + b^l.$$

Risulterà utile dare un nome a questa quantità: chiamiamo z^l l'*input pesato* dei neuroni[1] nello strato *l* e possiamo riscrivere l'equazione in termini di input pesato, come

$$a^l = \sigma(z^l).$$

Vale anche la pena notare che z^l ha componenti

$$z^l_j = \Sigma_k w^l_{jk} a^{l-1}{}_k + b^l_j,$$

ovvero z^l_j è l'input pesato per la funzione di attivazione del neurone *j* nello strato *l*.

La funzione costo

L'obiettivo della backpropagation è calcolare le derivate parziali $\frac{\partial C}{\partial w}$ e $\frac{\partial C}{\partial b}$ della funzione di costo C rispetto a qualsiasi peso *w* o bias *b* nella rete. Affinché la backpropagation funzioni, dobbiamo formulare due ipotesi principali sulla forma della funzione di costo. Prima di affermare queste ipotesi, tuttavia, è utile avere in mente una funzione di costo di esempio. Useremo la funzione di costo quadratico nella seguente forma:

[1] l'input non è per i neuroni, ma per la loro funzione σ

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2,$$

dove n è il numero totale di esempi di training; la sommatoria è sugli esempi di training x; $y = y(x)$ è l'output desiderato corrispondente; L indica il numero di livelli nella rete; e $a^L = a^L(x)$ è il vettore delle attivazioni in uscita dalla rete quando x è l'input.

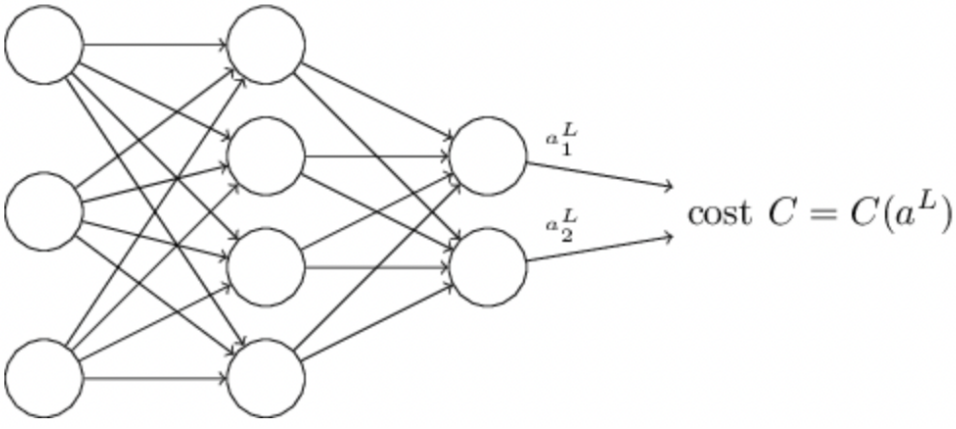
Facciamo due ipotesi sulla funzione di costo. La prima ipotesi di cui abbiamo bisogno è che la funzione di costo possa essere scritta come una media

$$C = \frac{1}{n} \sum_x C_x$$

rispetto alle funzioni di costo C_x per esempi di addestramento individuali x.
Questo è il caso della funzione di costo quadratico, dove il costo per un singolo esempio di addestramento è

$$C_x = \frac{1}{2}(y - a^L)^2.$$

Il motivo per cui abbiamo bisogno di questa ipotesi è perché ciò che la backpropagation ci consente effettivamente di fare è calcolare le derivate parziali $\frac{\partial C_x}{\partial w}$ e $\frac{\partial C_x}{\partial b}$ per un singolo esempio di addestramento. Quindi recuperiamo $\frac{\partial C}{\partial w}$ e $\frac{\partial C}{\partial b}$ calcolando la media sugli esempi di allenamento. Infatti, con questo presupposto in mente, supponiamo che l'esempio di addestramento x sia fissato e eliminiamo il pedice x, scrivendo il costo C_x come C.
Alla fine rimetteremo la x, ma per ora è un fastidio notazionale che è meglio lasciare implicito.
La seconda ipotesi che facciamo sul costo è che può essere scritto in funzione degli output dalla rete neurale:



Ad esempio, la funzione di costo quadratico soddisfa questo requisito, poiché il costo quadratico per un singolo esempio di training x può essere scritto come

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2,$$

e quindi è una funzione delle attivazioni delle uscite. Naturalmente, questa funzione di costo dipende anche dall'output *y* desiderato, e ci si potrebbe chiedere perché non consideriamo il costo anche in funzione di *y*. Ricordiamo, tuttavia, che l'esempio di addestramento dell'input *x* è fisso, quindi l'output *y* è anch'esso un parametro fisso. In particolare, non è qualcosa che possiamo modificare cambiando i pesi e i bias, cioè non è qualcosa che la rete neurale impara. E quindi ha senso considerare *C* come una funzione delle sole attivazioni dell'uscita a^L, con *y* semplicemente un parametro che aiuta a definire quella funzione.

Il prodotto di Hadamard

L'algoritmo di backpropagation si basa su operazioni algebriche lineari comuni: cose come l'addizione di vettori, la moltiplicazione di un vettore per una matrice e così via. Ma una delle operazioni è un po' meno comunemente usata. In particolare, supponiamo che *s* e *t* siano due vettori della stessa dimensione. Quindi usiamo $s \odot t$ per denotare il prodotto elemento per elemento dei due vettori.
Quindi le componenti di $s \odot t$ sono semplicemente

$$(s \odot t)_j = s_j t_j.$$

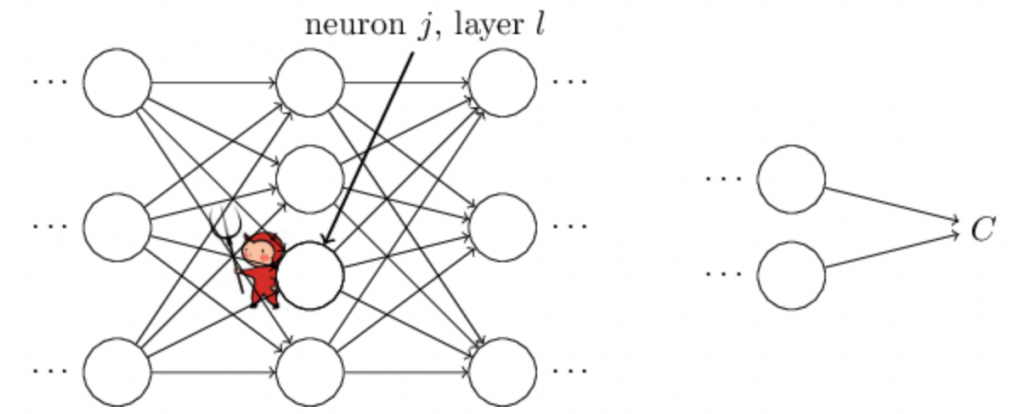
Ad esempio,

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}.$$

Questo tipo di moltiplicazione per elementi è talvolta chiamato prodotto di Hadamard o prodotto di Schur. Lo chiameremo il prodotto di Hadamard. Buone librerie di matrici di solito forniscono implementazioni veloci del prodotto di Hadamard e ciò è utile quando si implementa la backpropagation.

Le 4 equazioni fondamentali della backpropagation

La backpropagation riguarda la comprensione di come la modifica dei pesi e dei bias in una rete modifichi la funzione di costo. In definitiva, questo significa calcolare le derivate parziali $\frac{\partial C}{\partial w_{jk}^l}$ e $\frac{\partial C}{\partial b_j^l}$. Per calcolarli, introduciamo prima una quantità intermedia, δ_j^l, che chiamiamo errore nel *j*-esimo neurone dell'*l*-esimo livello. La backpropagation ci fornirà una procedura per calcolare l'errore δ_j^l e quindi metteremo in relazione δ_j^l con $\frac{\partial C}{\partial w_{jk}^l}$ e $\frac{\partial C}{\partial b_j^l}$.
Per capire come viene definito l'errore, immaginiamo che ci sia un demone nella nostra rete neurale:



Il demone siede al *j*-esimo neurone nel livello *l*. Quando arriva l'input per il neurone, il demone interferisce con l'attività del neurone. Aggiunge un piccolo cambiamento Δz^l_j all'input pesato del neurone, in modo che invece di emettere $\sigma(z^l_j)$, il neurone invece emette $\sigma(z^l_j + \Delta z^l_j)$. Questo cambiamento si propaga attraverso gli strati successivi della rete, provocando infine una variazione del costo complessivo di una quantità $\frac{\partial C}{\partial z^l_j}\Delta z^l_j$.

Ora, questo demone è un buon demone e sta cercando di aiutarci a migliorare il costo, cioè sta cercando di trovare un Δz^l_j che riduca il costo. Supponiamo che $\frac{\partial C}{\partial z^l_j}$ abbia un valore elevato (positivo o negativo). Quindi il demone può abbassare un po' il costo scegliendo Δz^l_j con un segno opposto a $\frac{\partial C}{\partial z^l_j}$. Al contrario, se $\frac{\partial C}{\partial z^l_j}$ è vicino a zero, allora il demone non può migliorare molto il costo perturbando l'input pesato z^l_j. Per quanto ne sa il demone, il neurone è già abbastanza vicino all'ottimale (questo è solo il caso di piccoli cambiamenti Δz^l_j; ipotizziamo che il demone possa solo fare cambiamenti piccoli). Quindi c'è un senso euristico in cui $\frac{\partial C}{\partial z^l_j}$ è una misura dell'errore nel neurone.

Motivati da questa storia, definiamo l'errore δ^l_j del neurone *j* nel livello *l* come

$$\delta^l_j \equiv \frac{\partial C}{\partial z^l_j}.$$

Secondo le nostre convenzioni usuali, usiamo δ^l per denotare il vettore di errori associati al layer *l*. La backpropagation ci darà un modo per calcolare δ^l per ogni livello, e quindi correlare quegli errori alle quantità di interesse reale, $\frac{\partial C}{\partial w^l_{jk}}$ e $\frac{\partial C}{\partial b^l_j}$. Ci si potrebbe chiedere perché il demone stia cambiando l'input pesato z^l_j: sicuramente sarebbe più naturale immaginare il demone che cambia l'attivazione dell'output a^l_j, con il risultato che useremmo $\frac{\partial C}{\partial a^l_j}$ come misura dell'errore. In effetti, se si esegue questa operazione le cose funzionano in modo abbastanza simile, ma si scopre che la presentazione della backpropagation è un po' più complicata algebricamente. Quindi ci atteniamo a $\delta^l_j = \frac{\partial C}{\partial z^l_j}$ come misura dell'errore.

La backpropagation si basa su quattro equazioni fondamentali. Insieme, queste equazioni ci danno un modo per calcolare sia l'errore δ^l sia il gradiente della funzione di costo. Le quattro equazioni sono indicate di seguito.

L'equazione per l'errore nello strato di output, δ^L

Le componenti di δ^L sono date dalla seguente equazione (BP1):

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j).$$

Il primo termine a destra $\frac{\partial C}{\partial a^L_j}$ misura quanto velocemente il costo sta cambiando in funzione della j-esima attivazione dell'output. Se, ad esempio, C non dipende molto da un particolare neurone di output j allora δ^L_j sarà piccolo, che è quello che ci aspetteremmo. Il secondo termine a destra $\sigma'(z^L_j)$ misura la velocità con cui la funzione di attivazione σ cambia in z^L_j. Si noti che tutto in questa equazione viene calcolato facilmente. In particolare, calcoliamo z^L_j durante il calcolo del comportamento della rete, e c'è solo un piccolo sovraccarico aggiuntivo per calcolare $\sigma'(z^L_j)$. La forma esatta di $\frac{\partial C}{\partial a^L_j}$ dipenderà, ovviamente, dalla forma della funzione di costo. Tuttavia, a condizione che la funzione di costo sia nota, dovrebbero esserci pochi problemi nel calcolare $\frac{\partial C}{\partial a^L_j}$. Ad esempio, se stiamo usando la funzione di costo quadratico allora

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a^L_j)^2,$$

e quindi $\frac{\partial C}{\partial a^L_j} = (a^L_j - y_j)$, che ovviamente è facilmente calcolabile.

L'equazione (BP1) è un'espressione per componenti per δ^L. È un'espressione perfettamente buona, ma non ha la forma basata su matrice che vogliamo per la backpropagation. Tuttavia, è facile riscrivere l'equazione in una forma matriciale, come segue:

$$\delta^L = \nabla_a C \odot \sigma'(z^L).$$

Qui, $\nabla_a C$ è definito come un gradiente, ossia un vettore le cui componenti sono le derivate parziali $\frac{\partial C}{\partial a^L_j}$. Si può pensare a $\nabla_a C$ come espressione della velocità di variazione di C rispetto alle attivazioni dell'output. È facile vedere che le due equazioni sono equivalenti, e per questo motivo d'ora in poi useremo il nome (BP1) in modo intercambiabile per fare riferimento a entrambe le equazioni. Ad esempio, nel caso del costo quadratico abbiamo $\nabla_a C = (a^L - y)$, e quindi la forma completamente matriciale di (BP1) diventa

$$\delta^L = (a^L - y) \odot \sigma'(z^L).$$

L'equazione per l'errore δ^l in termini di errore nel livello successivo, δ^{l+1}

Ecco l'equazione, che chiamiamo (BP2):

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l),$$

dove $(w^{l+1})^T$ è la trasposta della matrice dei pesi w^{l+1} per l'$(l+1)$-esimo livello. Questa equazione sembra complicata, ma ogni elemento ha una interpretazione chiara. Supponiamo di conoscere l'errore δ^{l+1} al livello $(l+1)$-esimo. Quando applichiamo la matrice trasposta del peso $(w^{l+1})^T$ possiamo pensare intuitivamente a questo come a spostare l'errore all'indietro attraverso la rete, dandoci una sorta di misura dell'errore all'uscita dell'l-esimo livello. Quindi prendiamo il prodotto Hadamard $\odot\sigma'(z^l)$. Questo sposta l'errore indietro attraverso la funzione di attivazione nel livello l, dandoci l'errore δ^l in ingresso al livello l.

Combinando (BP2) con (BP1) possiamo calcolare l'errore δ^l per ogni livello della rete. Iniziamo usando (BP1) per calcolare δ^L, quindi applichiamo l'equazione (BP2) per calcolare δ^{L-1}, quindi di nuovo l'equazione (BP2) per calcolare δ^{L-2} e così via, per tutto il percorso attraverso la rete.

L'equazione per il tasso di variazione del costo rispetto a qualsiasi bias nella rete

Ecco l'equazione, che chiamiamo (BP3):

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j.$$

Cioè, l'errore δ^l_j è esattamente uguale alla velocità di variazione $\frac{\partial C}{\partial b^l_j}$. Questa è un'ottima notizia, poiché (BP1) e (BP2) ci hanno già detto come calcolare δ^l_j. Possiamo riscrivere l'equazione in maniera più breve come

$$\frac{\partial C}{\partial b} = \delta,$$

dove si sottintende che δ viene valutato nello stesso neurone del bias b.

L'equazione per il tasso di variazione del costo rispetto a qualsiasi peso nella rete

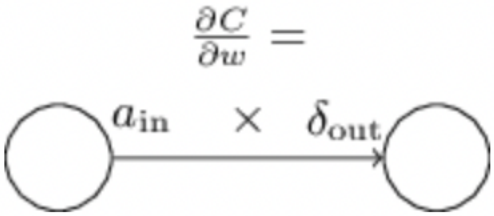
Ecco l'equazione, che chiamiamo (BP4):

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j.$$

Questo ci dice come calcolare le derivate parziali $\frac{\partial C}{\partial w^l_{jk}}$ in termini delle quantità δ^l e a^{l-1}, che sappiamo già come calcolare. L'equazione può essere riscritta in una notazione meno pesante come

$$\frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}},$$

dove si intende che a_{in} è l'attivazione del neurone in input alla connessione con peso w, e δ_{out} è l'errore del neurone in output alla connessione con peso w. Ingrandendo per guardare solo il peso w e i due neuroni collegati da quel peso, possiamo raffigurarlo come:



Una conseguenza di questa equazione è che quando l'attivazione a_{in} è piccola ($a_{\text{in}} \approx 0$), anche il termine gradiente $\frac{\partial C}{\partial w}$ tenderà ad essere piccolo. In questo caso, diremo che il peso impara lentamente, il che significa che non cambia molto durante la discesa del gradiente. In altre parole, una conseguenza di (BP4) è che i pesi che si trovano a valle di neuroni a bassa attivazione apprendono lentamente.

Ci sono altre intuizioni di questo tipo che possono essere ottenute da (BP1)-(BP4). Cominciamo guardando il livello di output. Consideriamo il termine $\sigma'(z^L_j)$ in (BP1). Ricordiamo dal grafico della funzione sigmoide che la funzione σ diventa molto piatta quando $\sigma(z^L_j)$ è approssimativamente 0 o 1. Quando ciò si verifica avremo $\sigma'(z^L_j) \approx 0$. Quindi la lezione è che un peso nel livello finale imparerà lentamente se il neurone di output è a bassa attivazione (≈ 0) o alta attivazione (≈ 1). In quest'ultimo caso è normale dire che il neurone di output si è saturato e, di conseguenza, il peso ha smesso di apprendere (o sta imparando lentamente). Osservazioni simili valgono anche per i bias dei neuroni di output. Possiamo ottenere informazioni simili per i livelli precedenti. In particolare, si noti il termine $\sigma'(z^l)$ in (BP2). Ciò significa che è probabile che δ^l_j diventi piccolo se il neurone è vicino alla saturazione. E questo, a sua volta, significa che qualsiasi peso in input a un neurone saturo imparerà lentamente.

Nessuna di queste osservazioni è troppo sorprendente. Tuttavia, aiutano a migliorare il nostro modello mentale di ciò che sta accadendo mentre una rete neurale apprende. Inoltre, possiamo "ribaltare" questo tipo di ragionamento. Le quattro equazioni fondamentali risultano essere valide per qualsiasi funzione di attivazione, non solo per la funzione sigmoide standard (questo perché, come vedremo tra poco, le dimostrazioni non utilizzano proprietà speciali di σ). Così possiamo usare queste equazioni per progettare funzioni di attivazione che hanno particolari proprietà di apprendimento desiderate. Come esempio, supponiamo di scegliere una funzione di attivazione (non sigmoidea) σ in modo che σ' sia sempre positiva e non si avvicini mai allo zero. Ciò impedirebbe il rallentamento dell'apprendimento che si verifica quando i neuroni sigmoidi ordinari si saturano.

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{j\,k}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

Dimostrazioni

Tutte e quattro le equazioni sono conseguenze della regola della catena (chain rule) del calcolo multivariabile.

Cominciamo con l'equazione (BP1), che fornisce un'espressione per l'errore di output, δ^L. Per dimostrare questa equazione, ricordiamo che per definizione

$$\delta^L_j = \frac{\partial C}{\partial z^L_j}.$$

Applicando la chain rule, possiamo riesprimere la derivata parziale di cui sopra in termini di derivate parziali rispetto alle attivazioni di output,

$$\delta^L_j = \sum_k \frac{\partial C}{\partial a^L_k} \frac{\partial a^L_k}{\partial z^L_j},$$

dove la somma è su tutti i neuroni k nel livello di output. Naturalmente, l'attivazione dell'output a^L_k del k-esimo neurone dipende solo dall'input pesato z^L_j per il j-esimo neurone quando k = j. E così $\frac{\partial a^L_k}{\partial z^L_j}$ scompare quando k $\neq$ j. Di conseguenza possiamo semplificare l'equazione precedente a

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}.$$

Ricordando che $a^L_j = \sigma(z^L_j)$ il secondo termine a destra può essere scritto come $\sigma'(z^L_j)$, e l'equazione diventa

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L),$$

che è proprio (BP1) con i componenti scritti in maniera esplicita.

Ora dimostriamo (BP2), che fornisce un'equazione per l'errore δ^l in termini di errore nel livello successivo δ^{l+1}. Per fare questo, vogliamo riscrivere $\delta^l_j = \frac{\partial C}{\partial z_j^l}$ in termini di $\delta^{l+1}_k = \frac{\partial C}{\partial z_k^{l+1}}$.

Possiamo farlo usando la chain rule:

$$\begin{aligned} \delta_j^l &= \frac{\partial C}{\partial z_j^l} \\ &= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1}, \end{aligned}$$

dove nell'ultima riga abbiamo scambiato i due termini a destra, sostituendo la definizione di δ_k^{l+1}. Per valutare il primo termine sull'ultima riga, ricordate che

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1}.$$

Differenziando, otteniamo:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l).$$

Riscrivendo l'equazione di $\delta^l{}_j$ di cui sopra, otteniamo quindi:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l).$$

Questa è (BP2) con i componenti scritti in maniera esplicita.

Queste dimostrazioni possono sembrare complicate, ma in realtà sono solo il risultato dell'applicazione attenta della regola della catena. Un po' meno succintamente, possiamo pensare alla backpropagation come un modo per calcolare il gradiente della funzione di costo applicando sistematicamente la regola della catena dal calcolo multivariabile. Questo è tutto ciò che c'è davvero dietro alla backpropagation. Il resto sono dettagli.

L'algoritmo di backpropagation

Le equazioni di backpropagation ci forniscono un modo per calcolare il gradiente della funzione di costo. Scriviamolo esplicitamente sotto forma di algoritmo:

Input *x*: imposta l'attivazione corrispondente a^l per il livello di input.

Feedforward: per ogni $l = 2,3,\ldots,L$ calcola $z^l = w^l a^{l-1} + b^l$ e $a^l = \sigma(z^l)$.

Errore di output δ^L: Calcola il vettore $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

Backpropagation dell'errore:
per ogni $l = L-1, L-2,\ldots, 2$ calcola $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^L)$.

Risultato: il gradiente della funzione di costo è dato da

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$$

e

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l.$$

Esaminando l'algoritmo si capisce perché si chiama "backpropagation". Calcoliamo i vettori di errore δ^l all'indietro, partendo dal livello finale. Può sembrare strano che stiamo attraversando la rete a ritroso, ma il movimento all'indietro è una conseguenza del fatto che il costo è una funzione degli output dalla rete. Per capire come varia il costo con i pesi e i bias precedenti, dobbiamo applicare ripetutamente la regola della catena, lavorando a ritroso attraverso i livelli per ottenere espressioni utilizzabili.

L'algoritmo di backpropagation calcola il gradiente della funzione di costo per un singolo esempio di addestramento, $C = C_x$. In pratica, è comune combinare la backpropagation con un algoritmo di apprendimento come la discesa del gradiente stocastico, in cui calcoliamo il gradiente per molti esempi di addestramento. In particolare, dato un mini-batch di *m* esempi di addestramento, l'algoritmo agisce come illustrato prima, con la sola modifica che i pesi e i bias vengono modificati non sulla base di un singolo gradiente, ma sulla base di una media tra tutte le modifiche calcolate sulla base di ciascuno degli *m* esempi.

Riferimenti bibliografici

D.E. Rumelhart, G.E. Hinton, and R.J.Williams, "Learning Internal Representations by Error Propagation" in D. Rumelhart and J. McClelland (eds) *Parallel Distributed Processing*, volume 1. MIT Press, 1986

Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015