U.I.C

Università Italiana Cracking – Italian University of Cracking

Neural Networks

(Neural Networks & Artificial Intelligence)

versione 1.0

-=Quequero=-

Email: quequero (at) bitchx (dot) it

Paper #3 – January 2004
Developed as an **UIC**'s project
http://quequero.org E-mail: quequero@bitchx.it

NEURAL NETWORKS ... 1
CENNI STORICI ... 3
IL NEURONE... ... 3
IL CERVELLO NEL COMPLESSO ... 4
IL NEURONE INFORMATICO ... 6
LA FUNZIONE DI ATTIVAZIONE ... 7
FUNZIONE BINARIA E GRADINO DI HEAVYSIDE ... 8
FUNZIONE LINEARE ... 9
FUNZIONE A SATURAZIONE LINEARE ... 9
FUNZIONE LOGISTICA (O SIGMOIDE) ... 10
SIMILSIGMOID ... 10
TANGETE IPERBOLICA ... 11
RADIAL BASIS FUNCTIONS ... 12
LA RETE NEURALE ... 13
REGOLE DI APPRENDIMENTO ... 16
REGOLA DI WIDROW-HOFF (O DELTA RULE) ... 16
DISCESA DELL'ERRORE ... 18
IL PROBLEMA DELLA SEPARAZIONE LINEARE ... 22
ALGORITMO DI ERROR BACK-PROPAGATION ... 25
Calcolo dell'errore dello strato di output ... 25
Evitiamo le oscillazioni ... 27
ALGORITMI GENETICI ... 28
SISTEMI EMBODIED ... 34
OSSERVAZIONI ... 40
CONCLUSIONI ... 42
RINGRAZIAMENTI ... 42

http://quequero.org E-mail: quequero@bitchx.it

Cenni Storici

L'intelligenza Artificiale vide la sua data di nascita nel 1956 durante una conferenza nel Dartmouth College alla quale parteciparono coloro che diventarono i pionieri dell'IA: Minsky, McCarthy, Simon, Newell.
Lo scopo era quello di trovare un modo per risolvere problemi complessi sui quali non e' possibile applicare una ricerca esaustiva, nel corso degli anni si inizio' anche a cercare di far risolvere ai computer quei problemi che per noi sono banali ma che risultano molto difficili per una macchina: un esempio classico e' quello del riconoscimento dei caratteri.
A questo proposito nacque, grazie a McCulloch e Pitts, il primo esempio di rete neurale artificiale e nel 1949 Donald Hebb propose, e successivamente realizzo', alcuni modelli di apprendimento.

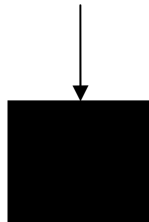
Il Neurone...

Unita' di base del nostro cervello, ne possediamo in media 100 miliardi, ogni neurone e' formato da un corpo detto ***soma***, da numerosi ingressi (mediamente 20.000 per neurone) detti ***dendriti*** e un'unica uscita detta ***assone*** (lunga fino a 2-3 metri).
E' estremamente importante ricordare che l'uscita del neurone e' soltanto una, mentre gli ingressi sono migliaia... Nonostante questo ogni assone puo' diramarsi e finire su altri neuroni.
I neuroni sono connessi tra loro attraverso delle giunzioni dette ***sinapsi*** che collegano l'assone ai relativi dendriti di un altro neurone, i segnali elettrici che escono dall'assone vengono trasmessi tramite dei neurotrasmettitori, quindi il segnale elettrico viene trasmesso da un neurone all'altro in maniera chimica e puo' essere di tipo eccitatorio oppure inibitorio.
Ma un neurone non passa il suo tempo a spedire segnali, anche perche' non avrebbe senso, e invia neurotrasmettitori soltanto quando viene attivato. La soglia di attivazione varia da neurone a neurone ed e' rappresentata dalla differenza di potenziale tra i capi dei suoi dendriti e l'interno della cellula, superato un certo livello il neurone viene attivato e viene rilasciato un *burst*.
Questo tipo di struttura rende il cervello un calcolatore parallelo estremamente potente nonostante la lentezza dei neuroni (che sono all'incirca 6 ordini di grandezza piu' lenti di un processore, ricordate infatti che un neurone ha un

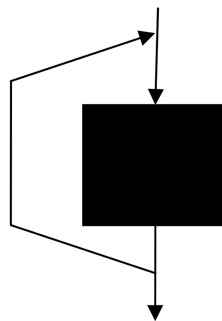
tempo di processing nell'ordine del millisecondo, cioe' 10^{-3} secondi, mentre un processore lavora nell'ordine di microsecondi, ovvero 10^{-9} secondi), nonostante questo siamo in grado di riconoscescere in pochissimi millisecondi un volto, una voce, una scritta.... Compito in cui un computer riesce con molta difficolta' e con gran dispendio di tempo. Ma un pc sa moltiplicare in qualche microsecondo numeri lunghi svariate decine di cifre... Mentre a noi servirebbero svariati minuti.
A questo punto e' chiaro che noi siamo una macchina non deterministica, un computer invece e' una macchina deterministica, ognuna ha i suoi pregi e i suoi difetti, e' per questo che noi esseri non deterministici abbiamo inventato i pc, per porre rimedio a quelle lacune che fanno parte della nostra struttura.
Il nostro cervello oltretutto e' una struttura estremamente plastica, puo' variare enormemente col tempo ma soprattutto... Se si rompe un elemento, l'insieme continua a funzionare in maniera pressocche' invariata, il cervello e' dunque *fault-tolerant* proprio grazie al suo parallelismo e alla sua plasticita', si e' stimato inoltre che il degrado delle prestazioni e' lineare nel caso di lesioni non troppo estese.

Il cervello nel complesso

Ora sappiamo com'e' formata l'unita' di base del nostro centro comandi, vediamo pero' come funziona in generale.
La memoria e' un fattore fondamentale, ci consente di interagire col mondo esterno, grazie ad essa ricordiamo i volti, le parole, la posizione degli oggetti e tutto il resto... Ma come funziona questo meccanismo?
Ovviamente non c'e' nessun harddisk dentro la nostra testa e capire dove finisce quella mole immensa di dati che ci passa davanti durante la nostra vita puo' non essere facile. Tutto questo e' reso possibile dalla struttura plastica del cervello, ad esempio quando studiamo per preparare un esame, una relazione o qualunque altra cosa, non facciamo altro che modificare le connessioni tra i vari neuroni creando meccanismi ricorsivi sul neurone stesso, mi spiego meglio, immaginate la classica scatola nera che ha un ingresso ed un'uscita, ad ogni ciclo inseriamo un input e la scatola ci da un output... cosi:



Inseriamo ora una connessione ricorsiva che dall'output torna verso l'input, in questa maniera:



In questo caso al ciclo *t* avremo un output che al ciclo *t+1* tornera' ad essere l'input della nostra scatola, in questo modo abbiamo creato una memoria perche' siamo in grado di ricordare lo stato della macchina ciclo per ciclo. All'incirca anche la nostra memoria e' cosi, i neuroni hanno una miriade di connessioni ricorsive (ovvero quando l'output di un neurone torna ad essere l'input del neurone stesso) e quanto piu' studiamo una cosa, tanto piu' vengono creati nuovi circuiti ricorsivi. E' lecito pensare che una persona con buona memoria sara' in grado di creare e mantenere a lungo dei circuiti

ricorsivi all'interno del proprio cervello, mentre una persona particolarmente "veloce" sara' in grado di creare queste connessioni in maniera piu' rapida di quanto faccia una persona piu' "lenta". Ma attenzione... Le simulazioni al computer ci hanno insegnato che piu' memoria implica anche meno capacita' di generalizzare, questo significa che siamo capaci di riconoscere determinati "pattern" anche quando non ne abbiamo mai visti di simili, esempio: se mostriamo ad un bambino un quadrato, lui imparera' che quello e' un quadrato, se poi prendiamo un altro quadrato piu' grande, piu' piccolo, colorato o ruotato, lui comunque rispondera' che quello e' un quadrato pur non avendone mai visti di tal colore, dimensione etc... Questa capacita' viene detta generalizzazione ed e' una tra le doti piu' grandi che abbiamo. Ma cosa succede se creiamo una rete con doti di memorizzazione troppo elevate? Succedera' che la rete memorizzera' con gran precisione tutti i pattern incontrati, diventando quindi eccessivamente selettiva e perdendo al tempo stesso la capacita' di generalizzare, perche' non piu' in grado di classificare tra i pattern validi quei pattern che si discostano, anche se poco, da quelli gia' incontrati.

Il neurone informatico

Ora che abbiamo una visione complessiva possiamo esaminare la struttura di un neurone informatico. La versione digitale della nostra struttura cerebrale sara' un'unita' composta da svariati ingressi (i dendriti), un'uscita con varie diramazioni (l'assone) e un'unita' interna di calcolo (il soma), ovviamente al termine di ogni dendrite e assone ci sara' una sinapsi (che chiameremo *peso*).
Un neurone informatico svolge un compito estremamente semplice, prende il valore in input che gli arriva dal dendrite d_0, lo moltiplica per il peso w_0 e ottiene un risultato, questo risultato sara' sommato con quello ottenuto dalla moltiplicazione dell'input del dendrite d_1 per il peso w_1 e cosi via per tutti i dendriti che arrivano sul nostro neurone, quest'operazione viene detta "somma pesata" (o *net*) e, indicando con d_i il dendrite i-esimo e con w_i il peso associato a tale dendrite, ecco matematicamente cosa vuol dire:

$$\sum_{i=0}^{n} w_i d_i$$

Per comodita a volte si inserisce un valore di soglia (*threshold*), in pratica:

$$\sum_{i=0}^{n} w_i d_i - \varepsilon$$

Cioe': ogni volta che viene moltiplicato l'input sul dendrite per il rispettivo peso, viene sottratto questo valore, tale pratica aiuta ad abbassare i valori in ingresso che possono anche diventare molto alti. Ben piu' comune e' invece la pratica di considerare il valore opposto alla soglia: il *bias*.
Il bias viene considerato come un neurone virtuale, in pratica si inserisce un singolo neurone che ha un assone che finisce su tutti gli altri neuroni (ad eccezione dei neuroni di input), l'uscita di questo neurone viene sempre considerata a 1, mentre i pesi vengono trattati come tutti gli altri. Sebbene il bias sia un neurone fittizio e' estremamente importante per il funzionamento della rete, senza di esso infatti si avrebbero comportamenti anomali.

A questo punto abbiamo la somma pesata, nel neurone reale il valore in input fa si che il neurone emetta il neurotrasmettitore se questo input e' sopra un valore "di soglia", la quantita' di neurotrasmettitore sara' variabile a seconda dell'intensita' dell'input, o nullo se l'input non e' alto abbastanza per "eccitare" il neurone.
A livello informatico simuliamo questo comportamento in maniera piuttosto semplice: prendiamo la somma pesata e la facciamo passare attraverso una funzione di attivazione che ci restituira' l'uscita del neurone.

La funzione di attivazione

Le funzioni di soglia sono il cuore del neurone, esse infatti calcolano l'uscita del neurone a seconda della somma pesata che giunge in ingresso e sono normalisissime funzioni matematiche con caratteristiche stabilite a seconda dei casi, ecco una panoramica completa su tutte le funzioni di soglia comunemente utilizzate, ricordate che un valore di uscita positivo e' detto eccitatorio, mentre uno negativo e' detto inibitorio:

Funzione binaria e gradino di Heavyside

$$\text{Output} = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

Questa funzione viene utilizzata molto spesso ed ha uscita nulla o eccitatoria, se abbiamo bisogno di valori di uscita sia eccitatori che inibitori allora dobbiamo ricorrere al gradino di Heavyside che graficamente e' identico alla funzione binaria tranne per il fatto che i valori di uscita sono 1,-1 invece che 0,1:

$$\text{Output} = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

Graficamente si presenta cosi:

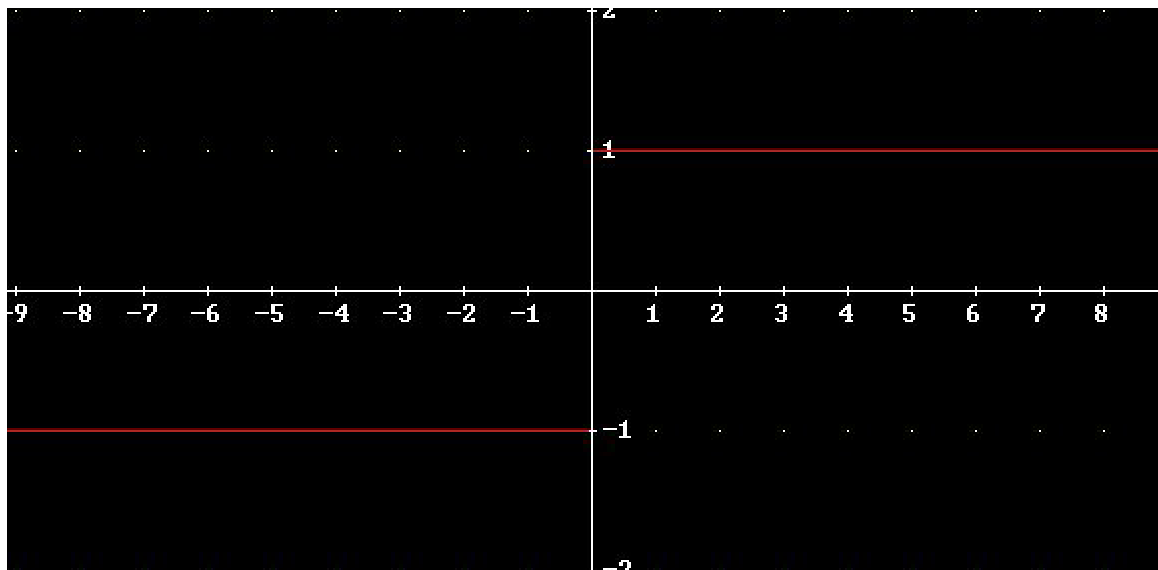


Fig. 1

Funzione Lineare

Un'altra funzione di utilizzo comune e' la funzione lineare o identita' (la bisettrice del primo e terzo quadrante):

$$\text{Output} = \{ y = x$$

Funzione a Saturazione Lineare

Per valori di uscita crescenti possiamo ricorrere alla funzione a saturazione lineare:

$$\text{Output} = \begin{cases} 0 & x \leq 0 \\ x & 0 < x < 1 \\ 1 & x \geq 1 \end{cases}$$

Graficamente e' cosi:

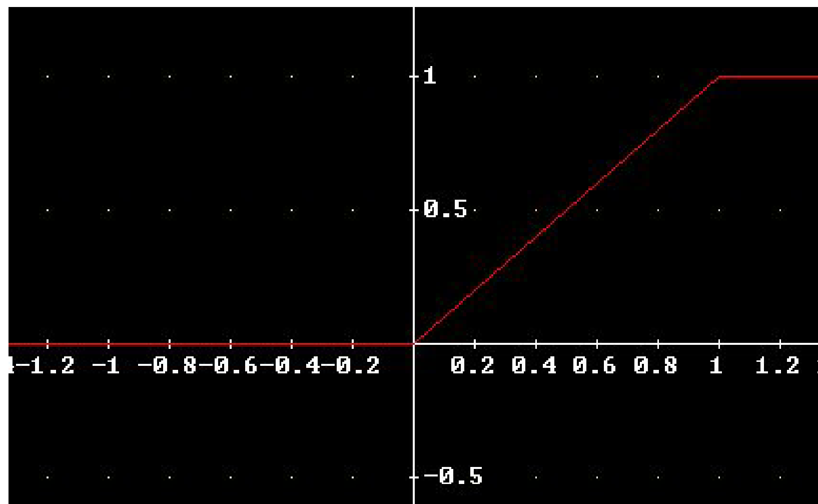


Fig. 2

Funzione Logistica (o Sigmoide)

Questa invece e' la funzione piu' importante tra tutte, specie nelle reti dove viene utilizzata la backpropagation (ne parleremo piu' tardi), viene chiamata: funzione sigmoide o logistica, l'equazione e':

$$\text{Output} = \frac{1}{1+e^{-kx}}$$

Graficamente si presenta cosi:

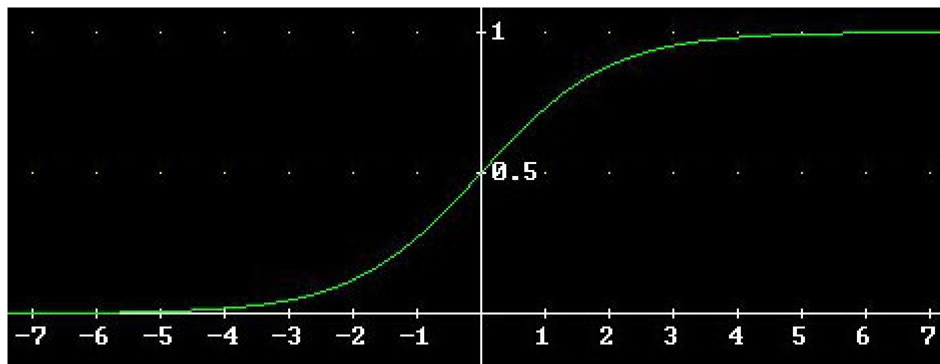


Fig. 3

E presenta alcune caratteristiche molto importanti: e' sempre crescente, e' continua su tutto l'asse dei reali ed e' derivabile, vale 1 a +∞ e 0 a -∞, questa funzione e' la piu' utilizzata nelle reti multistrato dove e' importante avere risultati nel continuo, pero' presenta un "problema", essendoci un esponenziale risulta piuttosto gravosa in termini di calcoli (considerate che ogni neurone ha una funzione di attivazione e le uscite vanno calcolate molte milioni di volte) ma per fortuna c'e' un rimedio, una persona che si trova del gruppo di ricerca in cui lavoro ha trovato un'interessante sostituto alla funzione logistica, vi presento la:

SimilSigmoid

$$\text{Output} = 0.5 + 0.5 * \left(\frac{x}{1+|x|} \right)$$

Graficamente e' fatta cosi:

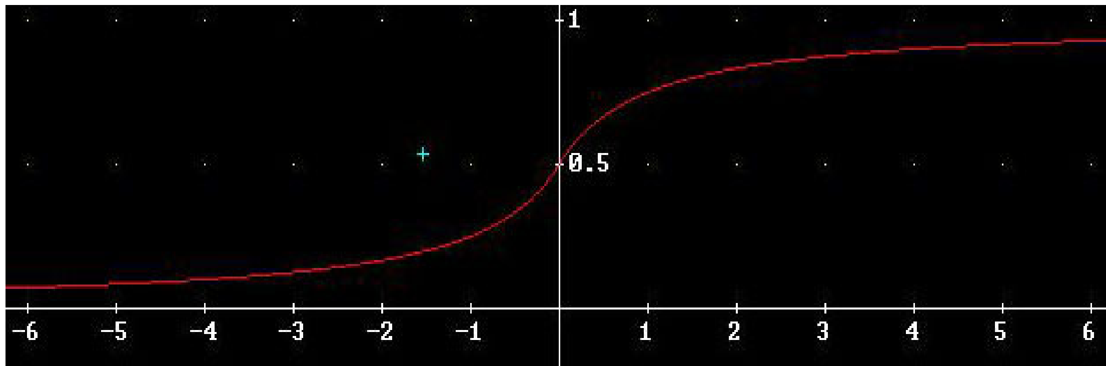


Fig. 4

E' del tutto simile alla sigmoide classica, sempre crescente, derivabile e continua su tutto l'asse R, compresa tra 0 e 1 e con il vantaggio di non avere nessun esponenziale, questo piccolo espediente ha ridotto i tempi delle mie simulazione da 3 a 2 ore.... E quando c'e' bisogno di fare molte simulazioni al giorno diventa un risparmio notevole.

Tangete Iperbolica

Un'altra funzione spesso utilizzata se si ha bisogno di valori di uscita inibitori e' la tangente iperbolica:

$$\text{Output} = \left(\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \right)$$

Che e' fatta cosi:

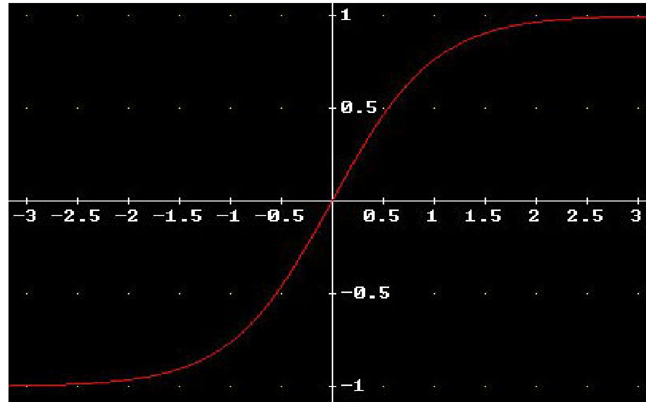


Fig. 5

Solo che qui gli esponenziali sono ben quattro.....

Radial Basis Functions

Ed infine... Per certi tipi di rete abbiamo bisogno delle *radial basis functions* (RBF) ovvero funzioni simili alla campana di Gauss:

$$\text{Output} = \frac{1}{1+x^2}$$

Che e' questa:

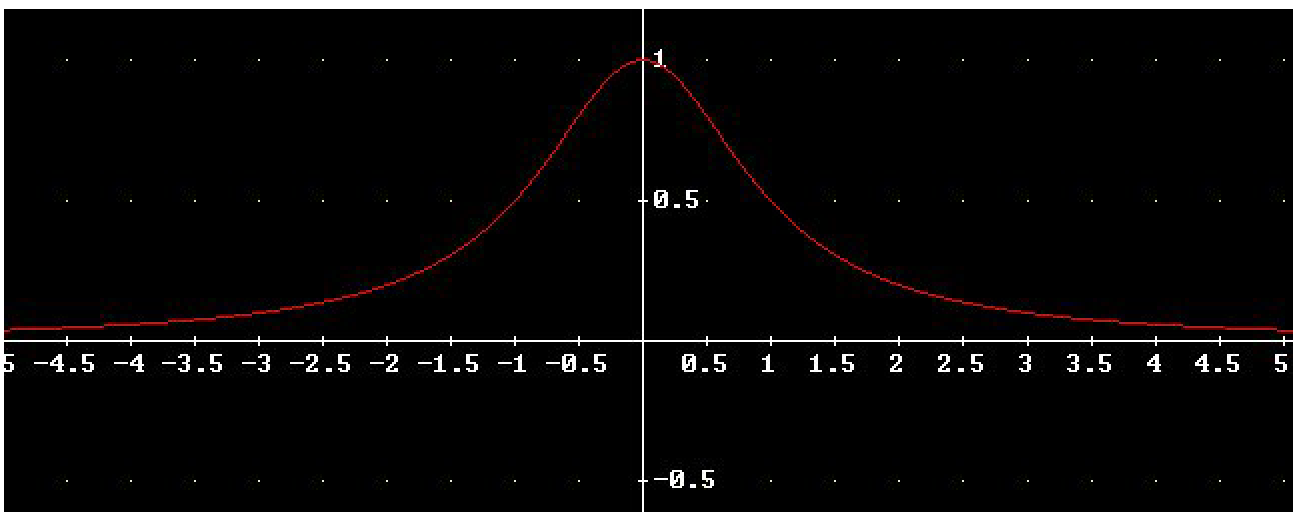


Fig. 6

Si noti che i valori di uscita della funzione sono uguali sia in caso di ingresso inibitorio che eccitatorio (ed e' l'unica funzione di quelle viste che presenta questa particolarita').
Possiamo ora disegnare il neurone informatico per averne una migliore visione d'insieme:

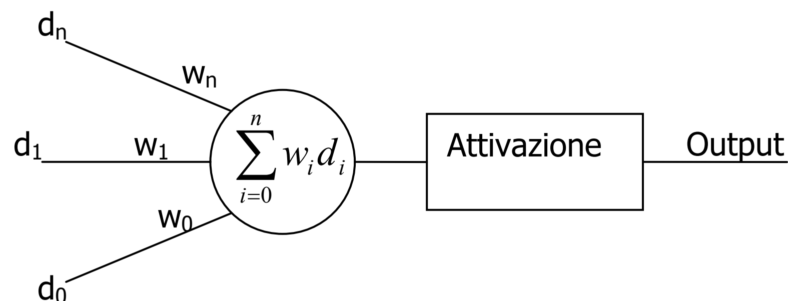


Fig. 7

Si prendono i valori che vengono dai dendriti, si moltiplicano per il rispettivo peso sinattico, si sommano i risultati ottenuti e si invia questo numero alla funzione di attivazione che ci restituisce l'output del neurone.

La rete neurale

Ora che sappiamo come e' fatto un neurone, possiamo definire la rete neurale: una rete neurale e' una serie di neuroni interconnessi tra di loro, se al neurone e' associata una funzione di trasferimento a soglia allora la rete viene detta a *perceptron*.
Le reti neurali sono composte almeno da due strati: uno strato di input che serve a prendere i dati in ingresso e uno strato di output dal quale esce il risultato (questo tipo di rete viene detta a strato singolo o single layer perche', per convenzione, gli strati di input e di output vengono considerati come uno solo), e' ovviamente possibile inserire tra questi due strati un ulteriore strato detto hidden layer, o anche piu' di uno (ma e' possibile risolvere la quasi totalita' dei compiti con al massimo due strati).

Le tipologie sono molto differenti, e diverse architetture possono essere utilizzate per svolgere lo stesso compito anche se alcune saranno ottimali e altre meno.
Una rete classica e' quella di tipo *feed-forward*, nella quale ogni neurone di uno strato e' connesso con i neuroni dello strato successivo, nessun neurone ha collegamenti all'indietro, nessun neurone e' collegato con altri neuroni dello stesso strato e non ci sono circuiti ricorsivi.
Eccone un disegno, normalmente le reti si disegnano dal basso verso l'alto, quindi lo strato piu' in basso e' l'input, quello piu' in alto e' l'output:

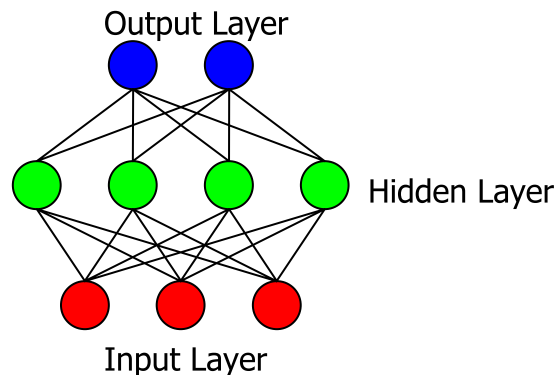


Fig. 8

Possiamo inserire dei circuiti ricorsivi sui neuroni, questa e' una maniera di implementare una sorta di memoria, normalmente i circuiti ricorsivi sono inseriti sullo strato nascosto e non sull'input o sull'output.
Ad esempio questo tipo di rete e' detta "rete caotica" o "rete di Hopfield":

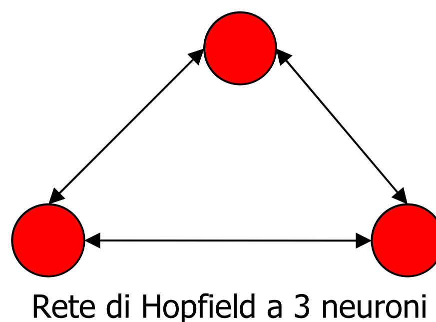


Fig. 9

Questo tipo di rete viene utilizzata come memoria auto-associativa, oppure per risolvere problemi di ottimizzazione, classico e' il problema: *N* lavori per *N* persone: ogni persona lavora, esegue un solo lavoro e tutti i lavori sono eseguiti, va quindi ottimizzato il rendimento globale. Altro problema famoso e' il TSP (Traveling Salesman Problem), cioe: ci sono N citta', conosciamo la distanza tra due qualunque di queste citta', dobbiamo visitare tutte le citta' solo una volta e tornare alla citta' di partenza senza visitare due volte lo stesso posto, questo tipo di reti si usa per risolvere il TSP attenendosi ai vincoli e minimizzando il percorso globale... E' facile intuire in questo caso la potenza delle reti, risolvere matematicamente il TSP con N = 10.000 richiederebbe tempi di calcolo piuttosto lunghi, mentre utilizzando una rete neurale di Hopfield si e' risolto il TSP con N=500.000 (utilizzando il metodo *Branch and Cut*) in tempi ristrettissimi.
Un'altra architettura estremamente potente e' la tipologia di Kohonen, queste reti vengono anche chiamate SOM (Self-Organizing Maps) ed hanno destato un grande interesse perche' molto simili alla struttura cerebrale.
Nelle SOM ogni neurone e' connesso con tutti gli altri, incluso se stesso, questa tipologia consente alla rete di auto-organizzarsi, in pratica per l'addestramento non e' necessario alcun supervisore in quanto la rete riesce a capire da sola quali legami ci sono con i vari input, i neuroni infatti si organizzano in zone, ognuna adibita alla risoluzione di un dato problema, cosi come avviene nel cervello umano dove ci sono zone dedicate all'apparato motorio, zone dedicate all'apparato vocale, alla memoria, al senso del tatto e cosi via... Questo tipo di organizzazione consente ai neuroni attivati di eccitare i suoi limitrofi e inibire quelli piu' distanti, creando dei veri e propri "reparti specializzati" all'interno della rete stessa, tali reparti vengono anche chiamati *bolle di attivazione*.
Le SOM risultano particolarmente adattive perche' in grado di scoprire autonomamente le proprieta' dell'input senza una supervisione esterna, sono quindi molto comode per studiare la capacita' di adattamento degli organismi nell'ambiente o fenomeni simili.

Regole di Apprendimento

Abbiamo detto che una rete e’ in grado di apprendere, cio’ vuol dire che una volta creata e compilata non sara’ in grado di fare nulla, dovremo percio’ insegnarle il compito per il quale e’ stata creata.
Un metodo di apprendimento e’ quello supervisionato, ovvero alla rete vengono presentati degli esempi di un *training set*, volta per volta viene calcolato l’errore commesso dalla rete e quindi i pesi vengono modificati di conseguenza. Le formule per la variazione dei pesi sono numerose e noi esamineremo le piu’ conosciute.

Regola di Widrow-Hoff (o Delta Rule)

Questa formula puo’ essere applicata solo alle reti a singolo strato (input-output) perche’ con questa regola non e’ possibile calcolare l’errore sullo strato nascosto. L’errore viene calcolato semplicemente sottraendo all’output desiderato l’output restituito dalla rete, quindi chiamando *o* l’uscita della rete e *r* l’uscita desiderata, possiamo calcolare l’errore δ in questo modo:

$$\delta = r_i - o_i$$

Supponiamo di voler creare, a scopo didattico, una rete che impari a fare le addizioni, se forniamo in input *2+2* e la rete torna *3* allora:

$$o = 3$$
$$r = 4$$

Quindi l’errore delta sara’: $\delta = 4 - 3 = 1$
La variazione delta del peso w_{ij} si calcolera’ in questo modo:

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

Con w_{ij} indichiamo un generico peso, i e j sono gli indici della matrice in cui quest’ultimo e’ contenuto, η e’ un numero compreso tra 0 e 1, tale numero viene detto *learning rate* e normalmente un valore piccolo significa piu’ cicli di addestramento ma risultati con errore molto piccolo, numeri piu’ grandi

significano meno cicli ma errore medio leggermente piu' alto, quindi e' un numero che rappresenta la velocita' di apprendimento del neurone. *δ* e' ovviamente l'errore ottenuto calcolato con la formula di sopra e x_i e' l'input che e' stato dato al neurone *i*-esimo. Quindi i pesi vengono aggiornati in questa maniera:

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

Nella rete rappresentata in Fig. 10 la variazione dei pesi viene calcolata in questo modo (ponendo il learning rate a 0.6):

$$w_{00} = w_{00} + [0.6 * (r_0 - o_0) * x_0]$$
$$w_{01} = w_{01} + [0.6 * (r_0 - o_0) * x_1]$$

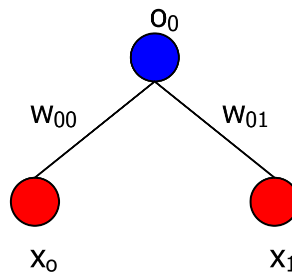


Fig. 10

Le modalita' di aggiornamento dei pesi sono due:
Modalita' on-line: ad ogni esempio viene calcolato l'errore e vengono variati i pesi.
Modalita' batch: ogni *N* esempi viene calcolato l'errore medio e vengono aggiornati i pesi di conseguenza.
La prima modalita' e' matematicamente meno corretta ma in questo modo si evita di incappare nei minimi locali (di cui parleremo dopo).

Discesa dell’errore

Consideriamo la rete disegnata sopra con uscita binaria e addestriamola per risolvere l’operazione di OR, cosi definita:

OR		
x_0	x_1	o_0
0	0	0
1	0	1
0	1	1
1	1	1

Adesso rappresentiamo su un mondo bidimensionale tutte le possibili coppie di 0,1:

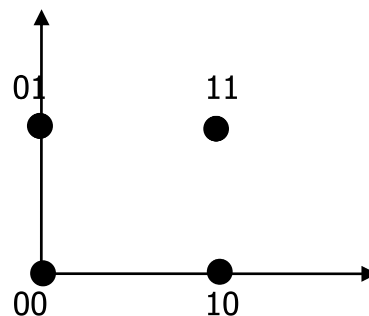


Fig. 11

Fornendo alla rete due input binari (0 o 1) dobbiamo far in modo che la rete ci restituisca 1 se la relazione e’ vera, 0 altrimenti, percio’ inserendo:

$$x_0 = 1, x_1 = 0$$

vogliamo ottenere come risultato: $o_0 = 1$ (perche’ l’OR e’ vero se uno dei due input risulta vero).
La rete in esame e’ a perceptron bidimensionale (2 input), tutto cio’ che la rete fara’ durante la fase di apprendimento sara’ approssimare una retta che divida le soluzioni vere, da quelle false, vale a dire:

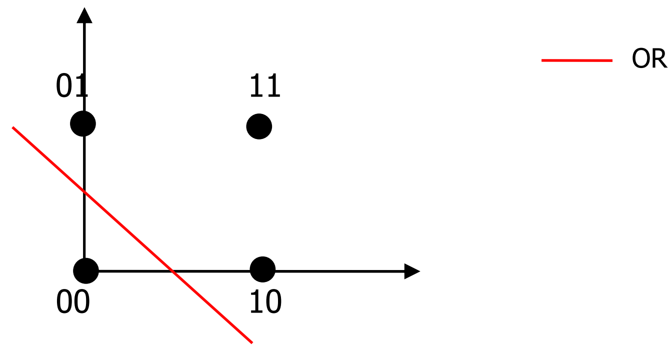


Fig. 12

In questa maniera i punti che cadono sotto la retta rossa daranno come output della rete: 0, quello che cadono al di sopra della retta daranno come output 1. Addestrare la rete e' piuttosto semplice:

1. Inizializziamo i pesi con valori casuali (ad esempio tra -1 e 1).
2. Inseriamo sui due neuroni di input due valori (*0,1* o *1,1* o *1,0* o *0,0*).
3. Calcoliamo l'errore ottenuto dall'output della rete.
4. Aggiorniamo i pesi con la delta rule.
5. Torniamo al punto 2. finche' la rete non risponde correttamente.

A questo punto la rete, come gia' detto, approssimera' una retta (perche' l'input e' bidimensionale) e man mano che i pesi verranno aggiornati, la retta cambiera' soltanto di pendenza. Dal momento che i pesi sono stati inizializzati casualmente, all'inizio (con molta probabilita') gli output saranno errati perche' la retta non dividera' bene le soluzioni. Nella figura in basso vediamo infatti una retta che divide le nostre soluzioni in maniera errata, il punto (1,0) dovrebbe infatti stare al di sopra di tale retta.

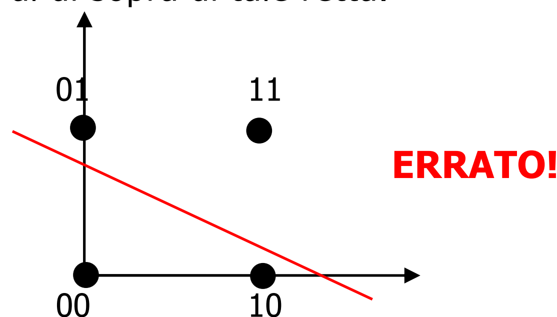


Fig. 13

La delta rule ci aiutera' a cambiare la pendenza della retta' finche' non otterremo risultati accettabili, a quel punto potremo inserire in input anche valori diversi da 1 o 0 (ad esempio 0.90) ed otterremo comunque un risultato esatto, inserendo x_1=0.90, x_2=0.95, la rete ci fornira' in output 1, che e' un risultato esatto.
Cio' dimostra come una rete neurale sia in grado di dare risultati esatti anche in presenza di *rumore* sull'input.
Il file Or.c contiene un esempio pratico di rete che impara l'OR (And.c e' in grado di imparare l'And, rispetto all'altro sorgente cambia soltanto una riga, quella che calcola il valore *expected*), come vedremo in seguito, giocando con il learning rate e' possibile diminuire notevolmente i tempi di addestramento (con un learning rate di 0.7 sono necessari appena 3 esempi, con learning rate piu' bassi ne sono necessari molti di piu', ma la rete diventa in grado di calcolare con esattezza anche input rumorosi).
Il *perceptron* semplice e' in grado di approssimare rette nel caso di input bidimensionale, piani nel caso di input tridimensionali e iperpiani nel caso di input multidimensionali.
Se al posto di funzioni binarie utilizziamo funzioni di trasferimento lineari (ad esempio la funzione identita') e rappresentiamo l'errore quadratico medio ottenuto in funzione dei pesi, allora otteniamo un iper-paraboloide che diventa un paraboloide nel caso di input bidimensionali:

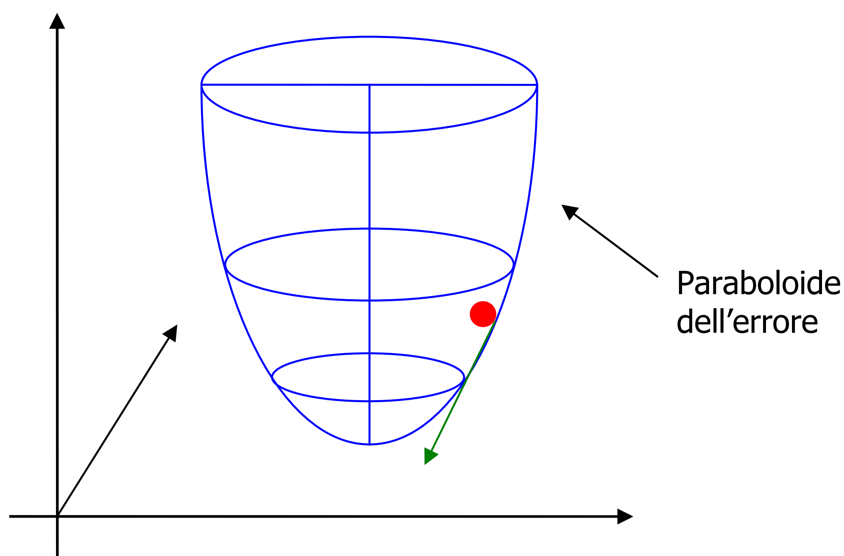


Fig. 14

La pallina rossa rappresenta la discesa dell'errore, i pesi sono inizializzati casualmente, quindi la pallina si trovera' in un punto qualunque del paraboloide. Ogni volta che i pesi vengono corretti la pallina scende verso il minimo globale, quando quella posizione sara' raggiunta l'errore sara' minimo e la rete avra' terminato il suo addestramento.
La scelta del learning rate e' particolarmente importante, un valore basso significa avere tempi di apprendimento estremamente lunghi, al contrario un valore alto consente una discesa rapida dell'errore ma lo svantaggio e' che ci sara', quasi certamente, un certo grado di oscillazione intorno all'errore minimo (locale o globale), cio' vuol dire che la pallina va da una parte all'altra del paraboloide senza riuscire a raggiungere il livello minimo.
Nel caso di paraboloidi a *n* dimensioni si possono presentare delle irregolarita' sulla superficie che formano dei minimi locali, tali minimi sono estremamente pericolosi perche' e' possibile che ci si resti intrappolati, in tal caso l'errore potrebbe anche essere accettabile ma di certo il risultato non sarebbe ottimale.
Ecco una figura che meglio spiega il fenomeno del minimo globale:

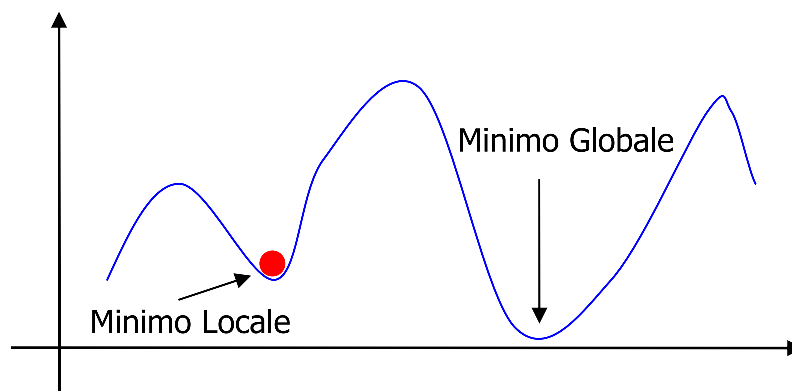


Fig. 15

Il minimo locale e' uno dei maggiori problemi quando si parla di reti neurali, ma i modi per evitarli, o per ridurre la possibilita' di restarne intrappolati, sono molti, uno di questi e' stato gia' spiegato ed e' la modalita' di addestramento *on-line*.

Il problema della separazione lineare

Abbiamo visto che un perceptron n-dimensionale e' in grado di suddividere lo spazio in rette, piani e iper-piani, ma proviamo ad insegnare alla nostra rete l'operazione di xor (OR-Esclusivo) che e' cosi' definita:

XOR		
x_0	x_1	o_0
0	0	0
1	0	1
0	1	1
1	1	0

Rappresentiamo sul nostro mondo bidimensionale i 4 punti binari e proviamo a dividere le soluzioni vere da quelle false cosi come abbiamo fatto per l'OR:

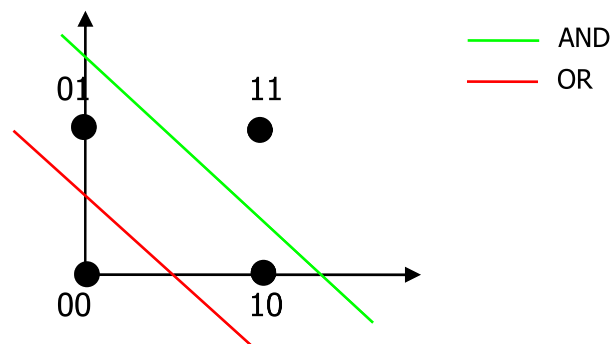


Fig. 16

Come si vede chiaramente, lo XOR e' separabile soltanto non-linearmente:

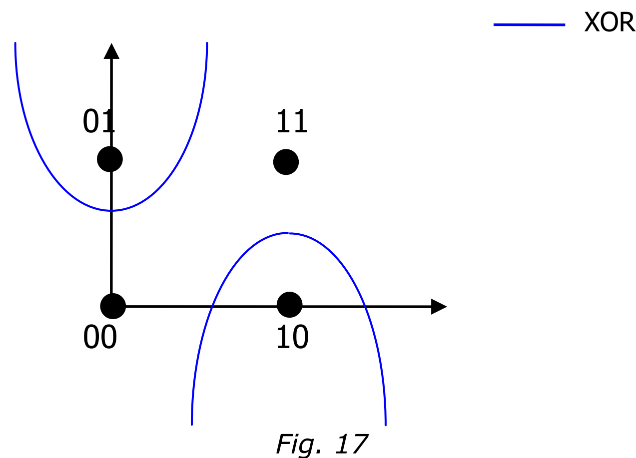


Fig. 17

E' quindi impossibile col nostro perceptron a strato singolo riuscire a risolvere il problema dello XOR, come si risolve questo problema?
Se il perceptron a strato singolo e' in grado "soltanto" di suddividere lo spazio in iperpiani, allora dobbiamo trovare una soluzione a questo problema.
Una possibile alternativa e' data dall'inserimento di input fittizi, ma non analizzeremo questa soluzione perche' al crescere dei neuroni di input, crescono in maniera esponenziale le combinazioni degli input fittizi, non e' pertanto una buona soluzione e quindi dobbiamo trovare una seconda alternativa: bastera' aggiungere uno strato nascosto di neuroni (o anche piu' di uno) per riuscire a suddividere lo spazio in partizioni piu' complesse. Gia' con un singolo strato nascosto e' possibile partizionare lo spazio in regioni estremamente complesse come ad esempio:

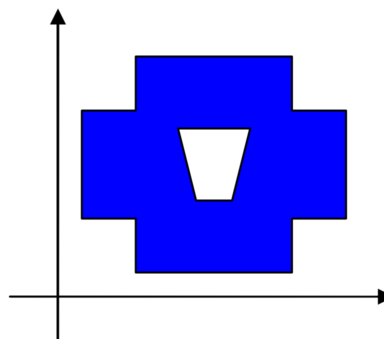


Fig. 18

In questo caso i punti *veri* si trovano nell'area blu, mentre i punti *falsi* si trovano nell'area bianca che, come e' chiaro, possono trovarsi sia all'interno che all'esterno del nostro spazio che e' stato partizionato in maniera molto piu' complessa rispetto a quanto si sarebbe potuto fare con delle rette o dei piani.
Allo scopo di risolvere il problema della separazione lineare, modifichiamo la nostra rete per rispettare la tipologia che stiamo per sperimentare aggiungendo uno strato nascosto di due neuroni:

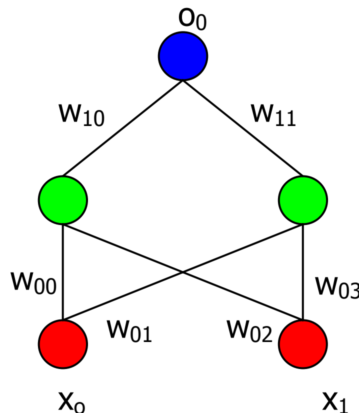


Fig. 19

Ma sorge immediatamente un problema, utilizzando la delta-rule possiamo addestrare la nostra rete? La risposta e': in parte.
Siamo infatti in grado di calcolare la variazione delta dei pesi w_{10} e w_{11}, ma non possiamo calcolare nessun'altra variazione perche' non siamo in grado di calcolare l'errore sullo strato nascosto. Sappiamo infatti qual'e' il valore atteso sul neurone o_0, ma non sappiamo quale dovra' essere questo valore sui due neuroni nascosti e quindi la regola di Widrow-Hoff ci potra' aiutare a modificare solo gli ultimi due pesi, ma di certo non sara' abbastanza per ottenere dei risultati corretti.
Questo problema complico' la vita ai ricercatori per moltissimi anni (quasi 30) e col tempo si perse interesse alle reti MultiLayer Perceptron (MLP) finche' nel 1986 non fu inventato un metodo per ovviare a questo problema, la soluzione venne chiamata Error Back-Propagation (EBP).

Algoritmo di Error Back-Propagation

Questo algoritmo consente, tramite apprendimento supervisionato, di trovare pian piano i pesi corretti.
L'EBP e' una generalizzazione della regola di Widrow-Hoff e consta di due fasi: nella prima fase si inserisce un input e si trova l'output della rete, nella seconda si calcola l'errore sull'output e poi si calcola l'errore sui neuroni dello strato nascosto. Una volta calcolato l'errore di uno strato, questo stesso errore viene propagato sullo strato superiore e questa procedura viene eseguita su ogni strato (nel nostro caso lo strato di output propaga l'errore sullo strato nascosto e lo strato nascosto propaga l'errore sullo strato di input).
Questo algoritmo richiede una funzione di trasferimento derivabile (utilizzeremo la funzione logistica, cioe' la sigmoide) e si puo' applicare a reti multistrato con ogni numero di strati nascosti, ovviamente l'errore calcolato sullo strato nascosto e' proporzionale in base all'output del neurone e ai pesi ad esso associati.
Per applicare l'EBP e' necessario applicare i seguenti step:

1. Inserire un input.
2. Memorizzare le uscite dei neuroni nascosti e dello strato di output.
3. Verificare l'errore commesso dallo strato di output.
4. Modificare i pesi che arrivano allo strato di output.
5. Verificare l'errore commesso dai neuroni nascosti.
6. Modificare i pesi che arrivano allo strato nascosto.

Calcolo dell'errore dello strato di output

L'errore d_k sullo strato di output viene calcolato in questa maniera:

$$d_k = (r_k - o_k) f^{'}(net_k)$$

Come al solito r_k rappresenta l'uscita attesa sul k-esimo neurone, o_k e' il valore effettivo di uscita del k-esimo neurone e $f'(net_k)$ e' la derivata prima della

funzione di attivazione calcolata sulla somma pesata (net) del k-esimo neurone. Se utilizziamo la funzione logistica allora la derivata e':

$$f'(x) = o_k(1 - o_k)$$

Percio' la formula per il calcolo dell'errore d_k diventa:

$$d_k = (r_k - o_k)o_k(1 - o_k)$$

A questo punto siamo in grado di aggiornare tutti i pesi che arrivano sullo strato di output tramite una semplice generalizzazione della regola di Widrow-Hoff, ovvero:

$$\Delta w_{jk} = w_{jk} + \eta d_k o_j$$

Δw_{jk} e' la variazione del peso ke va dal *k-esimo* neurone (nascosto) al *j-esimo* neurone (di output), w_{jk} e' il peso del quale vogliamo calcolare la variazione, η e' il learning rate, d_k l'errore calcolato poco sopra e o_j l'output del *j-esimo* neurone. Calcoliamo l'errore dello strato di output:

$$d_j = o_j(1 - o_j)\sum_k d_k w_{kj}$$

d_j e' l'errore del *j-esimo* neurone dello strato nascosto. *k* indica i neuroni dello strato che sta propagando all'indietro l'errore, o_j si riferisce all'output dei neuroni nascosti.
Adesso siamo in grado di aggiornare i pesi che vanno dallo strato nascosto allo strato di input:

$$\Delta w_{ij} = w_{ij} + \eta d_j o_j$$

Δw_{ij} e' la variazione del peso che va dal j-esimo neurone nascosto all'i-esimo neurone di input. Le altre variabili sono gia' state spiegate.
Nel file Xor.c troviamo l'implementazione dell'algoritmo di BackPropagation sulla rete che abbiamo disegnato in *Figura 19*.

Il programma Xor autoaddestra la rete per tutte le epoche definite dalla macro EPOCH, appena l'addestramento e' terminato viene chiesto all'utente di inserire due valori binari e come si vede i risultati sono estremamente vicini allo xor matematico, otteniamo infatti 0.9 quando dovremmo ottenere 1.0 e 0.02 quando dovremmo ottenere 0.0, la precisione e' quindi notevole e puo' essere ulteriormente migliorata semplicemente aumentando i neuroni nascosti.
Si puo' giocare un po' col learning rate per diminuire le epoche necessarie all'addestramento, ma valori troppo alti, come gia' detto, farebbero oscillare l'errore in questa maniera:

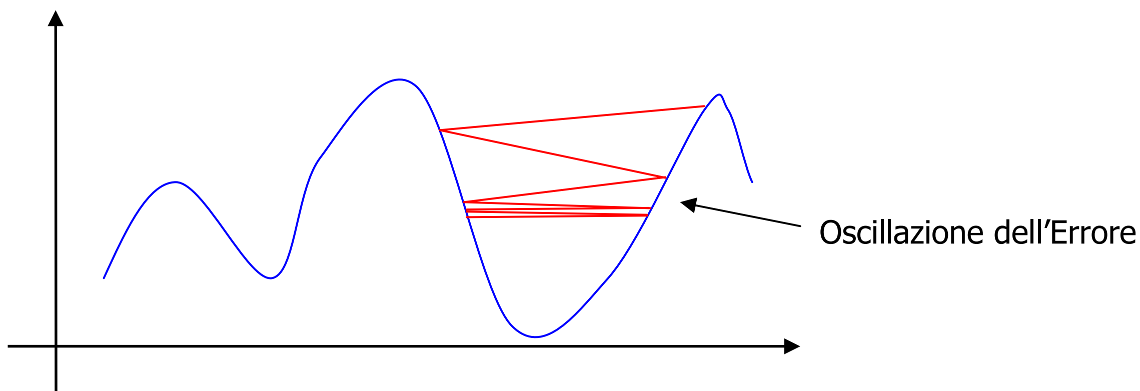


Fig. 20

Questo fenomeno e' dovuto al fatto che tramite l'EBP scendiamo lungo il minimo seguendo il gradiente, se il gradiente e' negativo si salta in avanti nella direzione del gradiente, se e' positivo si salta indietro, percio' se il learning rate e' sufficientemente basso allora si giunge in prossimita' del minimo, altrimenti si continua a "rimbalzare" lungo le pareti del minimo senza discenderle (come e' mostrato in *Figura 20*).

Evitiamo le oscillazioni

Evitare le oscillazioni dell'errore e' possibile in due modi: il primo consiste nello scegliere un learning rate molto alto (ad esempio 1.0, ma e' comunque possibile provare con valori superiori) per poi diminuirlo durante l'addestramento della rete.

La seconda tecnica consiste nell'inserire una costante alfa detta *momentum* (compresa tra 0 e 1). E' sufficiente modificare in questa maniera la regola per la variazione dei pesi:

$$\Delta w_{ij}(n+1) = \eta d_j o_i + \alpha \Delta w_{ij}(n)$$

In questo modo la variazione dei pesi all'istante *n+1* dipende dalla variazione applicata ai pesi nell'istante precedente.
Riporto qui sotto una tabella con i risultati di un benchmark effettuato nel '96 sul problema dello XOR, i risultati sono stati calcolati su 25 prove per ogni metodo, l'errore finale doveva essere di 0.04 in meno di 20.000 epoche:

Metodo	Num Medio di Epoche	Num di Successi
EBP	13184	7
EBP con momentum	5721	25
Intervento sul T.S.	3881	25

Un paio di parole vanno spese sull'ultimo metodo, ovvero l'intervento sul Training Set, in questo caso si lascia inalterato l'algoritmo di BackPropagation e si opera sui dati del Training Set eliminando gli esempi correlati tra loro, si procede in questa maniera:

1. Se l'errore e' maggiore di un valore prefissato si usa tutto il T.S.
2. Se l'errore non e' diventato molto piccolo, ma e' stato appreso circa il 50% di esempi, creare con l'altro 50% un secondo T.S.
3. Proseguire l'addestramento alternando il T.S. completo con l'altro fatto con esempi piu' difficili.

Algoritmi Genetici

Spesso non e' possibile applicare algoritmi di apprendimento come quelli gia' visti, alle volte perche' non e' possibile sapere quali sono i valori di output giusti e altre volte perche' il training set necessario non e' abbastanza esteso da permettere alla rete di apprendere il nostro problema.

E’ proprio in situazioni come queste che gli algoritmi genetici diventano indispensabili, a livello teorico sono semplicissimi da applicare sebbene la loro implementazione richieda un po’ di tempo per motivi che vedremo in seguito.
Gli algoritmi genetici sono stati ideati da J. H. Holland nel 1975 e si basano sull’evoluzione darwiniana. A livello pratico viene creata una popolazione di individui, ogni individuo possiede una proprio DNA che codifica i pesi della rete neurale e vive un tempo che puo’ essere variabile o prefissato, al termine di tale vita si calcola lo stato di salute (*fitness*) di ogni individuo, si prendono i migliori e si fanno riprodurre. Ogni volta che un indivuo si riproduce, cosi come accade in natura, alcuni cromosomi vengono modificati in maniera casuale e si inizia una nuova generazione... Questo processo andra’ avanti fino a che la nostra popolazione non avra’ una fitness media abbastanza alta, questo significa che e’ stata trovata una soluzione al nostro problema e quindi possiamo fermare l’algoritmo e prelevare il genoma dell’individuo con la fitness piu’ alta. E’ chiaro a questo punto che i GA (*Genetic Algorithms*) servono a risolvere problemi di ottimizzazione e ricerca, in realta’ e’ possibile risolvere tutti i tipi di problemi purche’ vengano riformulati in maniera opportuna.
Ecco come si procede per applicare un GA:

1. Vengono inizializzati casualmente tanti genomi quanti sono gli individui che faranno parte della nostra popolazione.
2. Il genoma viene decodificato per creare la rete neurale.
3. Viene calcolata la fitness di ogni individuo al termine della propria vita.
4. Si seleziona un numero *N* di individui presi tra i migliori e che diventeranno parte di un *Mating Pool*.
5. Gli individui del Mating Pool vengono fatti riprodurre (per *Clonazione* o per *CrossOver*) e vengono applicate le mutazioni genetiche.
6. Si torna al punto 2. finche’ la fitness non cresce.

Entriamo quindi nel dettaglio per spiegare le varie fasi.
Per prima cosa si dovra’ scegliere il tipo di simulazione che vogliamo creare, ovvero: a *N-variabile* o a *N-Fisso*.
Nel caso a *N-variabile* soltanto gli individui che sopravvivono possono riprodursi (e quindi la popolazione, specie nelle prime generazioni, tende a diminuire enormemente), nel caso a *N-Fisso* vengono fatti riprodurre solo gli individui con fitness piu’ alta ma la popolazione resta costante perche’ gli individui non possono morire, si passa quindi alla scelta della codifica del genoma.

Il genoma di un individuo altro non e' che la codifica dei pesi della sua rete neurale, quindi in fase di progettazione si sceglierа' la rete neurale degli individui ed i pesi verranno codificati in una stringa che formera' il genoma.
I principali tipi di codifica sono: codifica binaria, codifica numerica.
Nella codifica binaria ogni peso della rete viene codificato in una stringa binaria, ogni peso verra' sara' un gene e ogni gene binario avra' una lunghezza determinata dal massimo valore che potra' assumere il peso, quindi per un peso che puo' variare da -1.00 a 1.00 (con due cifre decimali) avremo bisogno di 8 bit, il primo bit sara' per il segno (se e' 1 il peso e' negativo, se e' 0 il peso e' positivo), e gli altri 7 bit serviranno a rappresentare tutti i valori tra 0 e 100, quindi in fase di decodifica non dovremo far altro che leggere il primo bit per il segno, leggere gli altri sette per comporre il numero e dividere per 100 in modo da avere due cifre decimali.
Il secondo metodo consiste nel rappresentare il genoma come una matrice di numeri *float* (o *double* se abbiamo bisogno di maggior precisione), utilizzando questo metodo non e' necessario alcun tipo di codifica/decodifica in quanto l'inizializzazione della rete consistera' in una semplice operazione di copiatura dei pesi dal genoma alla relativa matrice.
Una volta copiati tutti i pesi avremo un individuo (una rete neurale) che non sara' in grado di assolvere al proprio compito perche' i pesi sono stati inizializzati casualmente, pero' qualche individuo sara' piu' bravo della massa ad eseguire il proprio lavoro e quindi avra' un valore di fitness piu' alto, tutti gli individui piu' bravi verranno inseriti nel mating pool.
E' possibile riempire il mating pool scegliendo i migliori individui, oppure e' possibile utilizzare il metodo della roulette, vuol dire che ogni individuo ha una probabilita' di riprodursi proporzionale alla sua fitness, questa probabilita' viene identificata con un range di numeri.
Ecco un esempio, il primo individuo ha Fitness 100/100 e sulla nostra roulette gli verranno assegnati i numeri da 1 a 100, il secondo ha fitness 20/100, e quindi prendera' i numeri da 101 a 121, il terzo ha fitness 40/100 e quindi gli verranno assegnati i numeri da 121 a 161 e cosi' via per tutta la popolazione.
Si decidera' poi il numero *N* che rappresenta la quantita' di individui che finiranno nella generazione successiva, se questo numero e' N=10, allora verranno estratti 10 numeri casuali, si vedra' a chi appartengono e quindi gli individui possessori di questo numero verranno fatti riprodurre.
Durante l'atto di riproduzione verranno presi tutti gli individui che fanno parte del mating pool, verra' letto il loro genoma e verranno creati nuovi individui tramite *Clonazione* o *CrossOver*.

Clonare un individuo significa leggere il suo genoma, applicare le mutazioni e quindi trasferirlo nel nuovo individuo (simile alla riproduzione asessuata).
La riproduzione per CrossOver (simile alla riproduzione sessuale) invece consiste nel leggere parte del cromosoma del padre, parte del cromosoma della madre e combinarli (dopo aver inserito le mutazioni) all'interno del figlio, il CrossOver puo' essere di tipo *One-Point*, ovvero si genera un numero casuale x compreso tra 0 e L, dove L e' la lunghezza del genoma, quindi tutti i geni compresi tra 0 e x saranno letti da uno dei due genitori e tutti i geni compresi tra x e L saranno letti dall'altro genitore, il genoma verra' quindi riunito e si creera' il nuovo individuo.

Supponiamo $x = 3$ e $L = 10$

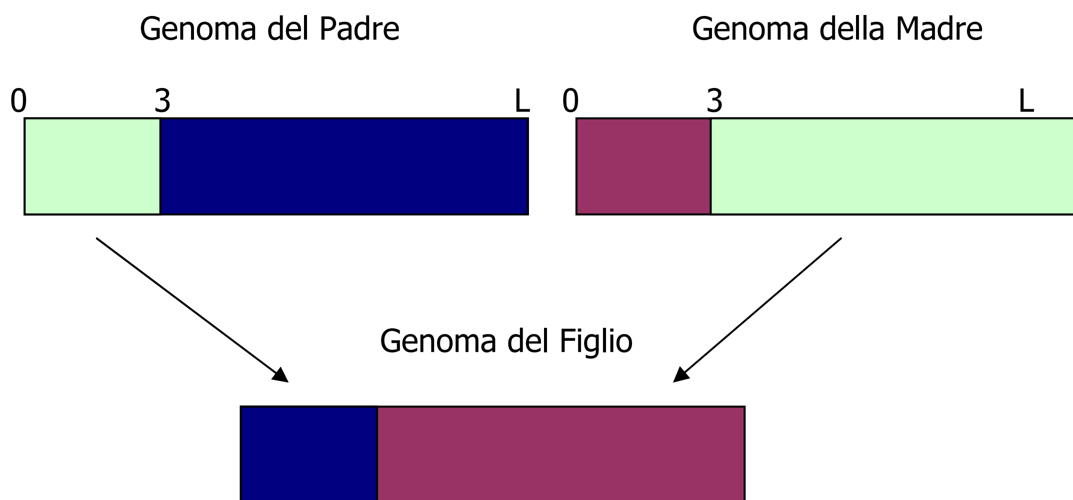


Fig. 21

Il crossover puo' esser fatto su un punto, due punti o *N* punti a seconda delle necessita'. Utilzzando la codifica numerica e' leggermente piu' semplice fare il CrossOver rispetto alla codifica binaria.
A questo punto il nuovo genoma e' pronto e dobbiamo soltanto fare le necessarie mutazioni, ad ogni gene viene assegnata una probabilita' di mutazione generalmente compresa tra [0.001 e 0.01], comunque delle osservazioni saranno fatte nella sezione finale di questo documento.

In fase di copia, per ogni gene viene generato un numero random, se questo numero e' minore o uguale alla probabilita' da noi scelta per la mutazione allora il gene muta. Utilizzando la codifica binaria, in caso di mutazione e' necessario soltanto invertire il bit in esame, diventa pero' piu' complesso nel caso della codifica numerica, infatti non possiamo agire sul singolo bit ma dobbiamo agire sul numero e dobbiamo anche scegliere entro quale range vogliamo variare tale peso.
In questo caso la codifica numerica non e' necessariamente piu' scomoda perche' possiamo anche imporre limitazioni piccole, mentre se utilizziamo la codifica binaria con pesi piu' grandi (ad esempio tra -10 e 10), e se a dover cambiare e' il bit piu' significativo, allora la mutazione sara' molto pesante.
Fino ad ora abbiamo interpretato il genoma come la codifica dei pesi, in realta' e' possibile utilizzare la codifica binaria anche per cambiare l'architettura della rete, in questo modo ogni stringa binaria indichera' soltanto la connessione con un dato neurone.
La tabella qui sotto e' composta solo di valori binari, 1 vuol dire connesso, 0 non connesso:

	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	1	0	1
4	1	0	0	0

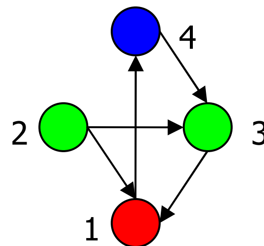


Fig. 22

In questo modo e' possibile sperimentare varie architetture in maniera assolutamente automatica, ci pensera' poi la fitness a dirci se funzionano bene o meno, i pesi possono essere gestiti alla solita maniera, o se si preferisce e'

anche possibile creare un genoma binario per le connessioni e uno numerico per i pesi.
La parte piu' importante del GA e' la *formula di fitness*, ovvero il grado di adattamento dell'invididividuo. La formula di fitness varia a seconda dei casi, ma vale la regola non scritta "piu' e' semplice piu' il risultato sara' buono", quindi vanno evitate ridondanze sui dati e tutto quanto puo' confondere la rete. Un esempio che ho visto risolto tramite GA e' stato quello della massimizzazione del diametro di un cerchio all'interno di un mondo pieno di bolle, mi spiego, la situazione iniziale era cosi:

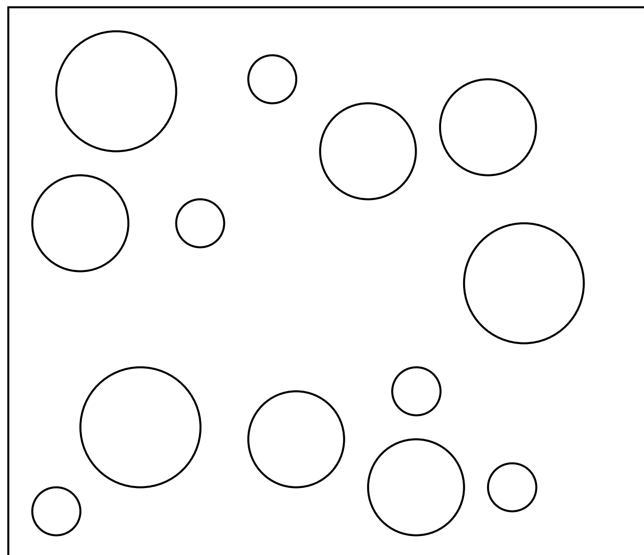


Fig. 23

Ed il problema era: dove si puo' costruire il cerchio di massimo diametro all'interno di questo Bubble-World?
In questo caso e' possibile utilizzare un GA senza alcuna rete neurale, basta infatti creare individui in posizioni casuali facendo poi accoppiare quelli che raggiungono il diametro maggiore, una possibile formula di fitness potrebbe essere: fitness = diametro.
In questo modo facendo accoppiare gli individui col diametro piu' grande (ovviamente il genoma contiene solo le coordinate del centro, perche' in questo caso la rete neurale e' assente) diventa facile trovare la posizione ottima, procedimento che a livello iterativo sarebbe invece piuttosto lungo, la soluzione per questo Bubble-World era:

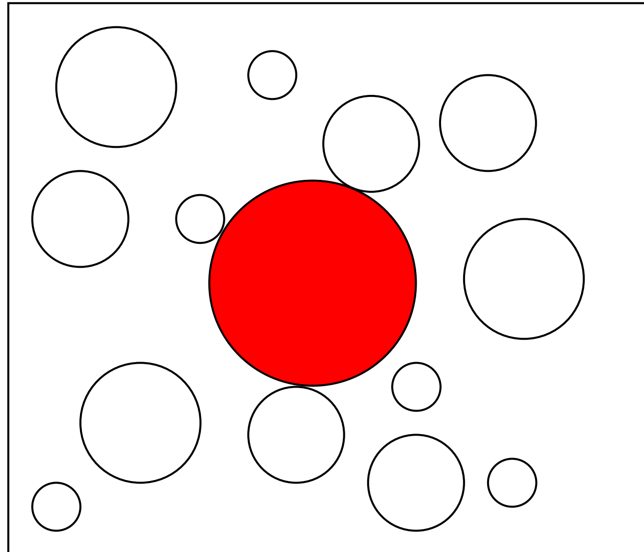


Fig. 24

E' quindi possibile applicare il GA non solo su problemi di intelligenza artificiale, ma anche su un qualunque problema di ricerca e ottimizzazione, questo esempio e' stato riportato sia per mostrare il doppio utilizzo dei GA sia per illustrare in maniera chiara il significato della formula di fitness.
L'utilizzo dei GA e' molto vasto, un'applicazione relativamente nuova e' quella dei sistemi embodied che saranno largamente illustrati nel prossimo capitolo.

Sistemi Embodied

Da un po' di tempo a questa parte i ricercatori si sono posti una domanda: cosa succederebbe se una rete neurale venisse inserita in un corpo, a sua volta inserito in un mondo, dove tutto e' governato dalle leggi dell'evoluzione darwiniana?
Verrebbe fuori quel che si dice: sistema embodied, ovvero un sistema intelligente, con un corpo che e' in grado di interagire con il mondo esterno come farebbe, all'incirca, un animale in natura. Questo e' il campo dell'IA nel quale sto lavorando ultimamente e in questo capitolo verranno presentati esempi e dati di lavori che ho svolto in passato su questo argomento.

Un esempio classico di sistema embodied e’ quello dell’ameba (con ameba indichiamo un qualunque individuo semplice che gira in un mondo) affamata, ovvero un’ameba che gira nel suo mondo alla ricerca di cibo. Sicuramente e’ il caso di iniziare con un sistema semplice, supponiamo quindi che l’ameba viva in un mondo quadrato, all’interno di questo mondo e’ presente un numero *N* di cibi, questo numero e’ mantenuto costante da un reinserimento del cibo ogni volta che ne viene mangiato uno. Per iniziare con un esempio semplice supponiamo che l’ameba non abbia alcun tipo di limitazione visiva, quindi puo’ vedere sia davanti che dietro per tutta la grandezza del mondo e il suo movimento e’ limitato a quattro azioni: rotazione di novanta gradi a destra, rotazione di novanta gradi a sinistra, avanzamento e immobilita’, verrebbe qualcosa di simile a questo:

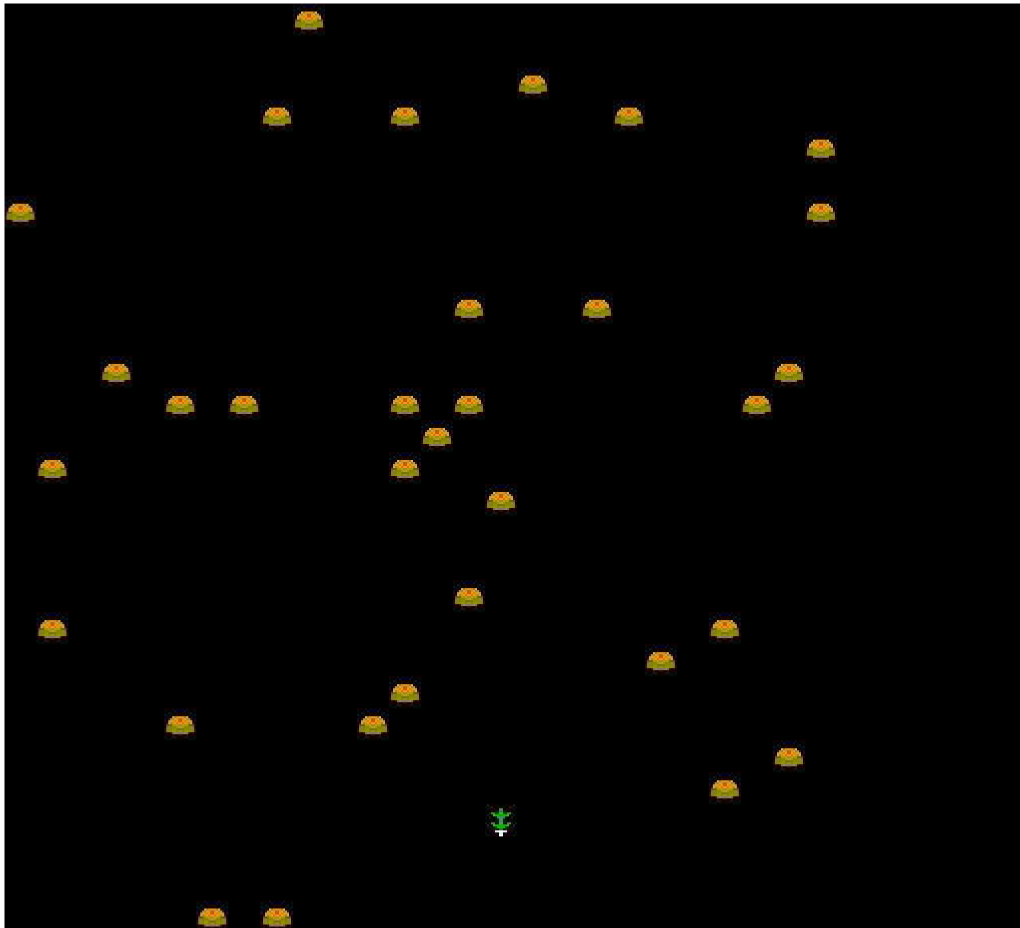


Fig. 25

Questa era la mia ameba, ma serve solo a dare un'idea. L'ameba non puo' vedere dove si trovano tutti i panini, altrimenti lo strato di input della rete sarebbe troppo grande, quindi possiamo farle vedere solo il panino piu' vicino. Dobbiamo ora stabilire un metodo di visione, il piu' semplice possibile e' quello a Manhattan, ovvero il campo visivo e' diviso in 4 parti: alto destra, alto sinistra, basso destra, basso sinistra. L'ameba non sara' quindi in grado di distinguere tra un panino che si trova davanti a lei e uno che si trova alla sua destra, ma vedrete che sviluppera' dei comportamenti adattivi al fine di sopperire a questa mancanza, notate inoltre che l'ameba non e' in grado di distinguere la distanza e nonostante questo sara' comunque in grado di vivere egregiamente nel suo mondo. Il sistema di visione che abbiamo scelto e' il piu' semplice possibile, in relazione ovviamente al tipo di movimento che l'ameba puo' fare.
A questo punto possiamo progettare la rete, gli input possibili sono 4, quindi due neuroni andranno bene, ne inseriremo tre nascosti e due di output ,visto che gli output possibili sono comunque 4 (avanti, destra, sinistra, ferma), verra' fuori qualcosa del genere:

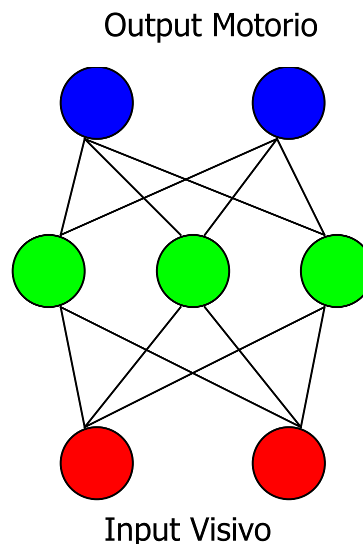


Fig. 26

Le funzioni di soglia saranno delle Simil Sigmoidi sui neuroni nascosti e binarie su quelli di output, codifichiamo ora l'input e l'output:

	Alto Destra	Alto Sinistra	Basso Destra	Basso Sinistra
Input 1	1	1	0	0
Input 2	1	0	1	0

L'input sara' questo:

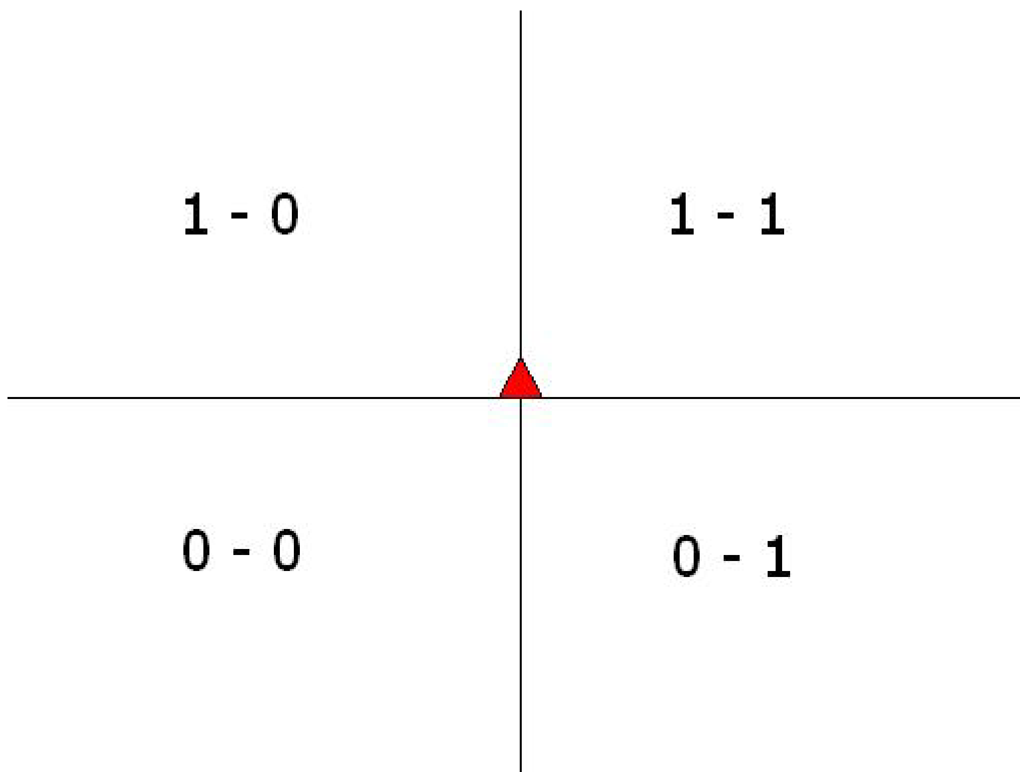


Fig. 27

	Avanti	Destra	Sinistra	Ferma
Output 1	1	1	0	0
Output 2	1	0	1	0

Non ci resta che stabilire una formula di fitness, vista la semplicita' del problema anche la fitness sara' semplice, quindi un individuo piu' mangera' piu'

avra' fitness alta (questo non e' un comportamento ecologico, ma essendo il primo esempio possiamo trascurare questo particolare).
Assegnamo quindi ad ogni panino un valore, diciamo 10, la fitness sara' cosi calcolata:

fitness = 10 * panini_mangiati

il 10 serve solo a distanziare un po' piu' le fitness tra loro ed e' stato inserito solo per comodita'. La codifica del genoma e' una scelta puramente soggettiva, io ho usato un array di float ma nessuno ci vieta di usare una codifica binaria, i pesi sono stati inizializzati tra 1 e -1, la percentuale di mutazione e' stata settata al 5% e la variazione massima di un peso e' stata settata al 30%, vuol dire che se un peso vale 1.0, una mutazione potra' farlo salire massimo a 1.3 o potra' farlo scendere minimo a 0.7, anche questa e' una scelta arbitraria e soltanto i tentativi e l'esperienza possono dirci quale e' meglio.
Non ci resta che scegliere la vita dell'individuo, poniamola per ora a 1000 cicli (vale a dire che dopo 1000 movimento l'individuo si riproduce e muore), e infine per la riproduzione verranno selezionati i 10 individui migliori dai quali verranno clonati altri 10 individui per un totale di 100 individui.
Una volta settati tutti i parametri gli individui vengono fatti evolvere (l'evoluzione non e' concorrente, vale a dire che ogni individuo ha il suo mondo con i suoi cibi), questo e' il grafico della fitness migliore e media di un test che ho fatto:

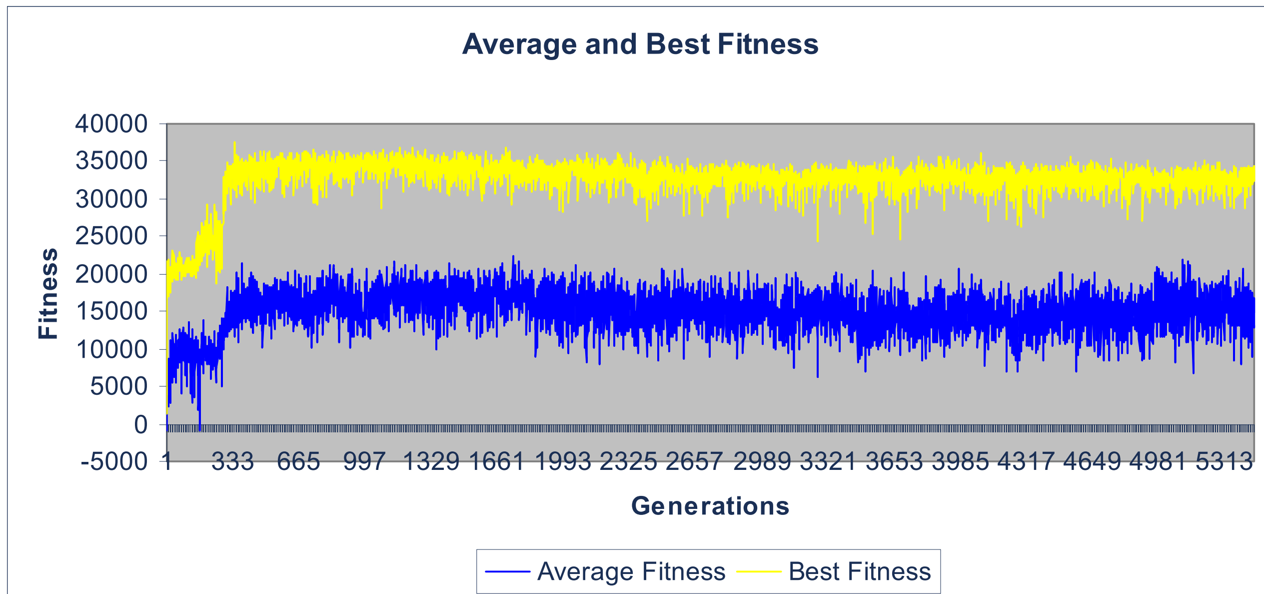


Fig. 28

Tra tutti i grafici ho scelto questo per un ben determinato motivo: l'algoritmo genetico esplora lo spazio di ricerca sia casualmente (le mutazioni) che cercando una media tra i punti migliori (la riproduzione) ma le mutazioni possono dar vita a salti evolutivi molto forti, guardate la fitness migliore, tra la generazione 333 e la 334 c'e' stato un salto molto grande, si e' passati da una fitness di 27000 a una di 34000 e questo e' anche il motivo principale per il quale si consiglia fare simulazioni molto lunghe.
L'andamento della fitness di una popolazione e' quasi sempre molto simile a quello visto poco sopra, ovvero: la fitness sale molto rapidamente nelle prime generazioni per poi stabilizzarsi, ma non siamo in grado di prevedere quando ci sara' un salto evolutivo e quindi e' buona norma allungare le nostre simulazioni, nella speranza di ottenere fitness piu' alte possibile.
Realizzare una simulazione di questo tipo e' un ottimo esercizio, il C++ e' particolarmente indicato a questo scopo perche' le stesse classi possono essere riutilizzate per qualunque altra simulazione, se avete bisogno di qualche suggerimento una possibile via e' quella di creare una classe per l'Individuo, una per la Popolazione, una per la ReteNeurale e una per il Genoma.
Adesso siete in grado di creare un sistema intelligente che puo' interagire con lo spazio circostante, e credetemi, le possibilita' sono veramente. Questo capitolo verra' esteso con materiale piu' avanzato non appena terminero' la ricerca sulla quale sto lavorando e faro' in modo di inserire anche il codice sorgente del programma.

Osservazioni

L'IA e' un campo di ricerca nuovo dove c'e' moltissimo da scoprire, questo e' il motivo per il quale ho scelto di inserire un capitolo dedicato alle osservazione che e' una raccolta di consigli, spesso non presenti sui libri, che potrebbero darvi una mano se davvero tutto e' stato creato come si deve, ma nonostante questo... Non funziona.
Una cosa che va sempre fatta e' la normalizzazione dei dati, se state insegnando ad un'ameba a mangiare il cibo piu' vicino in un mondo grande 100x100, non e' necessario inserire la distanza reale, in un mondo di queste dimensioni la diagonale e' circa di 141 unita', pertanto all'ameba verra' presentato in input un valore della distanza che puo' variare a 1 a 141... Ma e' un range troppo grande, possiamo normalizzarlo dividendo la distanza effettiva per 30 in modo da ottenere un range piu' piccolo, ricordate infatti che lavorando con i float (o meglio con i double) possiamo avere molte cifre decimali di precisione e alla rete neurale non interessa la distanza reale, quanto piuttosto il rapporto esistente tra due distanze, e' un po' come se gli cambiassimo unita' di misura, se per noi puo' sembrare strano, alla rete risulta invece comodissimo avere valori piu' piccoli.
Mantenete sempre coerenza tra gli input, se ci sono 4 neuroni che prendono in input valori tra 0 e 2, e il quinto prende valori tra 1 e 1000, fate in modo di normalizzare l'input in modo da ottenere, anche sul quinto neurone, valori tra 0 e 2, o tra 0 e 4, cercate sempre di mantenere coerenza sugli ingressi.
Se in input dovete dare un angolo, non inseritelo in gradi, ma dateglielo in radianti e normalizzate anche questo valore su un range piccolo, non intimoritevi nel sentire le parole "range piccolo", perche', come gia' detto, avendo a disposizione molte cifre decimali possiamo avere una grande precisione.
Mantenete coerenza tra input e output, questo e' fondamentale, se create un'ameba che puo' solo ruotare e andare avanti e' assolutamente inutile darle in input la distanza dal cibo e l'angolo, perche' non sarebbe in grado di sfruttare nessuno dei due valori, anzi sarebbe anche controproducente perche' la rete dovrebbe adattarsi a "semplificare" l'input, e non e' detto che ci riesca.
Ricordatevi sempre di denormalizzare gli output, se in input date un angolo in radianti compreso tra a 0-3 e se in output vi aspettate un angolo, ovviamente anche questo sara' in radianti e compreso tra 0-3 quindi denormalizzatelo prima di dire che non funziona.
Non inserite troppi neuroni nascosti, difficilmente avrete bisogno di piu' di uno strato nascosto, ma inserendo troppi neuroni leverete alla rete la capacita' di generalizzare (che e' l'unica cosa che ci interessa) e invece che generalizzare memorizzera' il problema, rispondendo male ai nuovi input. Non so se esistano regole scritte o meno, personalmente ho inventato una regoletta che fino ad ora e' sempre andata bene: se i neuroni di input

sono pochi (sei o meno) e quelli di output sono anch'essi pochi (tre o meno), allora la formula diventa:

hidden = input + (input/2)

se invece i neuroni in input sono di piu' allora sono solito inserire nello strato nascosto circa il doppio di neuroni rispetto a quello di input. Ovviamente non e' detto che tali regole funzionino, ma per me vanno bene quindi le uso, ad ogni modo col tempo imparerete, ad occhio, a creare le vostre reti, perche' anche noi siamo reti neurali e con l'esperienza siamo in grado di generalizzare un problema ☺.
Le reti neurali non sono in grado di risolvere gli algoritmi di crittografia, quindi scordatevi di crackare gli hash MD5 per il semplice fatto che non c'e' corrispondenza tra l'input e l'output, mentre il lavoro della rete e' proprio quello di cercare dei pattern noti.
Inizializzate i pesi sempre con valori molto piccoli, in genere pesi tra -1.000 e 1.000 vanno benissimo, semmai alzatene il range se vedete che nel genoma tendono a salire molto, ricordate anche che e' buona norma fare piu' simulazioni prima di dire "non funziona", i pesi infatti vanno iniziati casualmente e non sempre si e' fortunati da avere delle buone simulazioni al primo tentativo.
La percentuale di mutazione puo' essere elevata (anche il 10%) nel caso di reti piccole formate da 15 neuroni o meno, se la rete cresce il valore di mutazione va ridotto altrimenti si avra' un andamento della fitness molto oscillante.
Semplificate al massimo le formule di fitness, la rete cerca infatti di trovare il modo per ottenere valori piu' alti, ma se la formula e' troppo difficile allora ci vorra' molto piu' tempo.
Ricordate inoltre che, specie con i GA, gli individui inventeranno una varieta' incredibile di modi per "imbrogliare", vuol dire che otterranno fitness elevate pur assumendo comportamenti da imbroglione, eccone un paio di esempi: in qualche esempio mi capito' di vedere un'ameba con fitness molto alta, quest'ameba non faceva altro che girare in maniera casuale nel mondo piuttosto che dirigersi verso un determinato cibo. Un'altra persona stava lavorando ad un braccio che doveva toccare degli oggetti, capito' che il braccio migliore era quello che toccava a casaccio tutto lo spazio dove poteva arrivare, piuttosto che guardare l'oggetto e dirigersi in quella direzione.
Spesso verranno fuori comportamenti sub-ottimali, vuol dire che ci saranno casi in cui si otterranno comportamenti che portano al giusto fine ma nel modo sbagliato, un esempio e' l'ameba che puo' arrivare al cibo andando dritta ma che invece fa un sacco di curve inutili.

Conclusioni

Per il momento questo documento e' concluso nonostante manchino ancora molte cose, appena mi sara' possibile lo aggiornero' con i risultati della mia ultima ricerca e aggiungero' anche altri tipi di rete che non sono stati trattati.
Come si dice di solito: feedback is welcome, quindi se potete mandatemi impressioni, correzioni e critiche costruttive sul documento.
Spero, col nuovo anno, di avere abbastanza tempo per concludere come vorrei questo breve trattato sull'intelligenza artificiale.

Ringraziamenti

Ringrazio tutti coloro che mi hanno permesso di studiare e imparare (grazie ai loro chiari insegnamenti) un argomento che da sempre avrei voluto approfondire, in ordine sparso:

Silvio Cammarata – Reti Neuronali, dal Perceptron alle reti caotiche e neuro-fuzzy.
Prof. Parisi e tutto il GRAL che saluto.
Prof. Beatrice Lazzarini – Introduzione alle Reti Neurali e agli Algoritmi Genetici.
A colui che ha scritto Connessionismo e Reti Neurali, il paper non e' firmato.
Donald R. Tveter – BackPropagation.
Asynchro che anni fa mi mise la pulce nell'orecchio.
Un saluto a tutto il chan #crack-it su AzzurraNet, a AndreaGay, Ironspark, Ntoskrnl, fobbo, korn, s1mo, N0body88, xOANON, DJK, CrazyKiwi, Camillo, Zairon, Pincuzzo, Albe, Tmy17, deimos, brnocrist, sgrakkyu, Spider, artemis, mary, giulia, khuma e tutti quelli che inevitabilmente dimentichi di menzionare quando cerchi di fare un elenco e hai una fretta spietata. A tutti i fratellini SPP, a Catalyst, al mio cane Full che se non era per lui non so come l'avremmo spalati quei 40kg di neve, a Pantani che e' riuscito a farsi sei metri in aria col bob atterrando ancora vivo e senza riportare evidenti danni fisici e che poi ci ha riprovato, solo che e' atterrato (senza bob) 12 metri oltre il trampolino, al mio virgolette-virgolette-amico-virgolette-virgolette Franco (si chiama proprio cosi).
Se qualcuno e' stato dimenticato verra' inserito nella prossima release, promesso.
Ciao a tutti e happy new year, sperando come al solito che il prossimo sia migliore di quello passato.

-=Quequero=-