Introduzione alle reti neurali attraverso algoritmi in C

Pre-requisiti per questo tutorial:

- Conoscenza delle basi del linguaggio C
- Buone basi matematiche

Obiettivi di questo tutorial:

- Fornire le nozioni teoriche e matematiche alla base della teoria delle reti neurali e dell'intelligenza artificiale
- Fornire algoritmi di esempio in C per lo sviluppo di elementari reti neurali

Introduzione alle reti neurali

Le reti neurali sono un'applicazione dell'intelligenza artificiale relativamente vecchia (sono state teorizzate negli anni '60, per poi essere abbandonate sui primi anni '70 in seguito alla pubblicazione, da parte di M.Minsky e S.Papert, di *Perceptrons*, un libro che metteva in risalto le carenze della tecnologia), ma che ultimamente sta vivendo un periodo di rinascita in seguito al rinnovato interesse nei confronti delle tecniche di intelligenza artificiale e al perfezionamento di questa tecnologia stessa.

Il meccanismo delle reti neurali si ispira dichiaratamente a quello del sistema nervoso degli animali. Le reti neurali non sono progettate per "nascere imparate", né tantomeno per avere una grande precisione in fatto di calcolo (d'altronde la potenza di calcolo di un cervello umano verrebbe facilmente ridicolizzata da qualsiasi calcolatrice), ma sono progettate per apprendere. La fase di apprendimento di una rete neurale si basa su un campione di dati ("training set") che viene presentato alla rete stessa, spesso con i risultati che si desiderano ottenere. Ad esempio, se voglio addestrare una rete neurale a risolvere le 4 operazioni fondamentali, posso presentare in input alla rete diversi numeri, e poi i risultati che desidero ottenere con quei numeri. Sarà la rete stessa a imparare, graduatamente, i meccanismi che sono alla base dell'operazione che deve compiere. Ovviamente, più corposo sarà il training set della rete, più precisi saranno i risultati che si potranno ottenere una volta completato l'addestramento.

Le reti neurali, come accennavo prima, non sono molto usate nel campo del calcolo matematico-scientifico, proprio in base alla loro scarsa precisione da sistemi fuzzy[1], ma si rivelano utilissime (proprio in virtù delle loro caratteristiche fuzzy) nella risoluzione di problemi reali. Un cervello umano non saprà risolvere un integrale definito con il

1 I sistemi fuzzy sono sistemi che si ispirano alla logica fuzzy, una logica polivalente che si può considerare un ampliamento della logica booleana classica, che prende in esame non solo un numero discreto possibile di valori, come lo 0 e 1 nell'algebra di Boole, ma anche possibili valori "intermedi" non numerabili. La logica fuzzy si pone quindi come valida alternativa alla logica tradizionale nell'esame dei problemi reali, in cui i valori che possono assumere le variabili in gioco non sono numerabili, o almeno non facilmente numerabili.

metodo dei rettangoli con la stessa rapidità con cui lo risolverebbe un calcolatore elettronico, ma può riconoscere con una facilità disarmante un cane da un albero, o la voce di un amico da lontano, anche se disturbata da altri rumori. Delle applicazioni simili in campo tecnologico le hanno anche le reti neurali, utili, ad esempio, per il riconoscimento visivo elettronico, per il riconoscimento vocale, e così via.

Struttura di una rete neurale

La struttura di una rete neurale si rifà esplicitamente a quella di una rete neurale umana. Nell'uomo i neuroni sono costituiti da un corpo cellulare (*soma*) e da *dentriti* che mettono il neurone in comunicazione con altri neuroni. In presenza di determinati segnali queste comunicazioni si attivano (*sinapsi*), con il rilascio di sostanze di tipo chimico-ormonale (*neurotrasmettitori*) che trasmettono lo stimolo da un neurone all'altro. L'input di un neurone non è altro che la media pesata di tutti i segnali provenienti in input dagli altri neuroni per il "*peso sinattico*" della sinapsi in questione, ovvero l'"importanza" che riveste quella sinapsi nel collegamento. In base a questo valore, chiamato *potenziale post-sinattico*, il neurone può rispondere con una valore di output, che può essere minore o maggiore in fatto di intensità rispetto al precedente (effetto "calmante" o "eccitante") in base ai valori degli input in quel dato momento.

Una rete neurale artificiale ha una struttura molto simile.

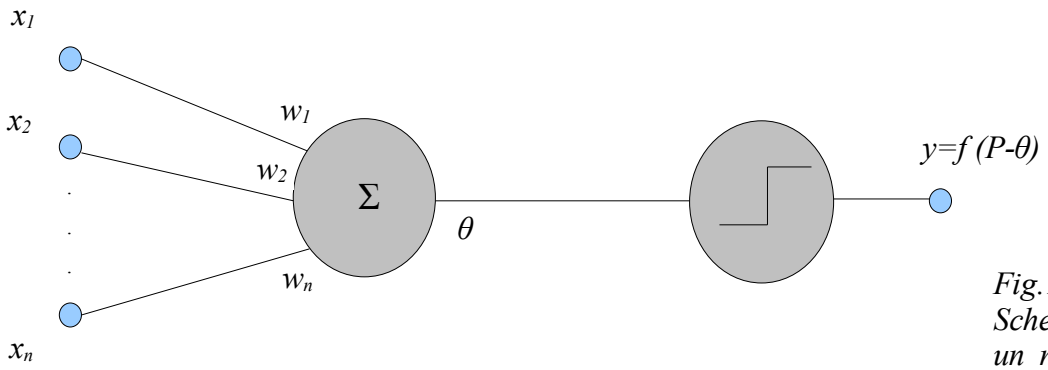

Fig.1 Schema di un neurone artificiale

Dove $x_1,...,x_n$ sono gli input presentati al neurone o alla rete neurale, $w_1,...,w_n$ sono i pesi sinattici delle singole connessioni (ovvero quanto quella connessione influenza il risultato finale). La media pesata degli input per i pesi sinattici delle singole connessioni fornisce il *potenziale post-sinattico* del neurone:

$$P=\sum_{i=1}^{n} w_i x_i$$

L'output y del neurone è dato da $f(P-\theta)$, dove θ è una soglia caratteristica del neurone, mentre f è una funzione di trasferimento. Le principali funzioni di trasferimento utilizzate nelle reti neurali sono la funzione a gradino, la funzione sigmoidale e la tangente iperbolica, tutte funzioni aventi codominio nell'intervallo [0,1] (o [-1,1] nel caso della tangente iperbolica).

La funzione a gradino, il tipo di funzione di trasferimento più semplice usata nelle reti neurali, è una funzione così definita:

$$f(x) = \begin{cases} 1 & \text{se } x \geq 0 \\ 0 & \text{se } x < 0 \end{cases}$$

che graficamente si traduce così:

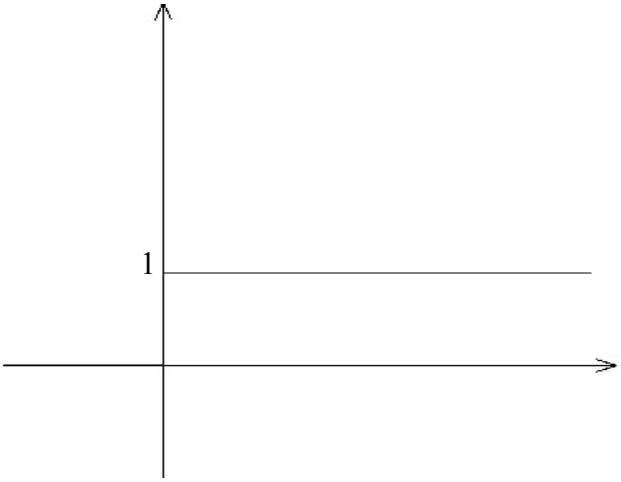

Fig.2 Grafico della funzione "a gradino"

Usando questa funzione, il neurone emette un segnale *y=1* quando $x = (P-\theta) \geq 0$, quindi $P \geq \theta$, mentre emette un segnale *y=0* (quindi rimane inattivo) quando $P < \theta$.

Un'altra caratteristica funzione di trasferimento è la sigmoidale, o curva logistica, di equazione

$$f(x) = \frac{1}{1+e^{Ax}}$$

Al variare del parametro *A* la curva può diventare più o meno "ripida". In particolare, la curva tende alla funzione a gradino *g(x)* che abbiamo visto prima per $A \rightarrow -\infty$, mentre invece tende *g(-x)* quando $A \rightarrow +\infty$.

Se *x=0*, ovvero se $P=\theta$, allora il valore di uscita del neurone artificiale sarà 0.5, mentre invece sarà approssimativamente 0 (ovvero il neurone è "spento") per $\theta \gg P$ e 1 per $\theta \ll P$.

Una proprietà molto interessante di questa funzione, una proprietà molto utilizzata nella fase di apprendimento delle reti neurali, riguarda sua sua derivata prima. In particolare:

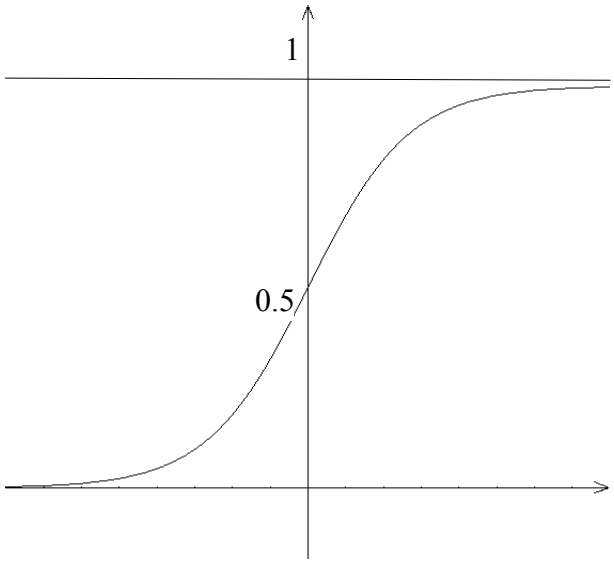

Fig.3 Grafico della curva sigmoidale

$$\frac{dy}{dx} = Ay(1-y)$$

Questa proprietà implica che la derivata della funzione sigmoidale si può scrivere come un semplice prodotto, sorvolando le regole di derivazione, e questo è molto utile a fine computazionale (un calcolatore potrà trovare facilmente la derivata di una funzione così costruita).

Una funzione alternativa alla sigmoidale, relativamente meno usata nel campo delle reti neurali, è la tangente iperbolica, di equazione

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

Una funzione definita $f : \mathbb{R} \rightarrow [-1,1]$, a differenza delle due che abbiamo visto prima che sono a codominio in $[0,1]$.

Ogni neurone può dare in un dato momento, come abbiamo visto, un solo valore in output in funzione dei suoi input, mentre una rete neurale può complessivamente dare un numero variabile di valori in output. Se quindi una rete ha input n valori $x_1,x_2,...,x_n$ in un dato momento, la rete darà in output m valori $y_1,y_2,...,y_m$ in quel momento, in funzione delle x_i. Ovvero

$$y_j = f_j(x_1,x_2,...,x_n) \quad \text{per} \quad 1 \leq j \leq m$$

Quindi i valori in input in un certo momento sono un vettore X di n componenti, generalmente compresi tra 0 e 1. Anche gli m valori del vettore di output Y sono compresi tra 0 e 1, in quanto vengono confinati in questo intervallo dalla funzione di trasferimento usata (funzione a gradino o sigmoidale), quindi il vettore X va a identificare un punto A all'interno di un ipercubo booleano di n dimensioni, e Y un punto B all'interno di un'altro ipercubo booleano a m dimensioni. Nel caso di $n=m=2$ gli ipercubi degenerano in 2 quadrati di $2^n=2^m=4$ vertici, mentre invece nel caso $n=m=3$ gli ipercubi degenerano in 2 cubi di $2^n=2^m=8$ vertici. La rete neurale può quindi essere vista come un'applicazione binaria che associa a ogni punto A contenuto nel primo ipercubo un punto B contenuto nel secondo.
La grande idea alla base delle reti neurali però non è solo l'applicazione binaria tra i punti di un insieme e i punti di un altro insieme. L'applicazione associa il punto A e anche un suo *intorno* ad un intorno del punto B del secondo insieme. Questo è molto utile nel caso in cui i segnali di input sono "sporcati", ad esempio nel caso di una rete neurale per il riconoscimento vocale, in grado di fare il suo dovere anche quando il suono è sporcato da rumori esterni, oppure una rete neurale per il riconoscimento calligrafico, in grado di fare il suo dovere anche quando il simbolo grafico non è perfettamente identico a quello appreso in fase di training. Questa proprietà deriva proprio dalle proprietà tipicamente *fuzzy* delle reti neurali.

Tecniche di apprendimento

Le reti neurali possono apprendere in due modi diversi: in modo *supervisionato* e in modo *non supervisionato*.

Nel primo caso, in ogni istante t vengono presentati alla rete dei campioni x_i in input, e i corrispondenti valori di output d_j desiderati. Le variazioni dei pesi sinattici

$$\Delta w = w(t+1) - w(t)$$

sono una funzione dell'errore, e quindi dello scarto $y_j - d_j$, dove y_j è l'output ottenuto, e d_j l'output desiderato. Gli algoritmi di apprendimento in genere hanno come obiettivo quello di minimizzare l'errore quadratico medio. Quindi un'apprendimento supervisionato richiede la conoscenza sia dei valori di input x_i, sia dei valori desiderati d_j. Questi due tipi di dato forninscono quello che viene definito il *training set* della rete neurale.

Nel caso dell'apprendimento non supervisionato, vengono forniti alla rete molti campioni di input $X_i = (x_{i1},...,x_{in})$, da associare a un numero m di classi $C_1,...,C_m$. Il programmatore non fornisce alla rete la classe di appartenenza di ogni vettore di input; è la rete stessa ad auto-organizzarsi, modificando i suoi pesi sinattici in modo da poter eseguire classificazioni corrette. Gli algoritmi di apprendimento hebbiani sono classificabili all'interno di questa categoria. Questi algoritmi, basati sulla legge di Hebb, rafforzano il peso sinattico w_{ij} tra due generici neuroni i, j quando la loro attività è concorde (ovvero quando i risultati delle loro funzioni di trasferimento y_i, y_j sono di segno concorde), mentre lo indebolisce nel caso opposto:

$$\Delta w_{ij} = \eta y_i y_j \quad \text{con} \quad 0 \leq \eta \leq 1$$

Dove η è un coefficiente compreso tra 0 e 1 da cui dipende la variazione del peso sinattico.

Sviluppo di una rete neurale

Vediamo ora come implementare un'elementare rete neurale sfruttando algoritmi in C. La rete neurale che intendiamo sviluppare è relativamente semplice, ed è addestrata per svolgere la somma algebrica tra due numeri. Anche la sua funzione di attivazione è semplice (per questo esempio non useremo né la funzione a gradino né la sigmoidale viste precedentemente, nonostante siano queste le funzioni più utilizzate), useremo la funzione identità $y=f(x)=x$ (ma potremmo usare una funzione qualsiasi, anzi ve lo lascio come esercizio).

Cominciamo a definire, magari in un file header, i tipi di dato di cui abbiamo bisogno:

```
typedef struct TypeNeuron neuron;
typedef struct TypeSynapsis sinapsi;
typedef struct TypeLayer layer;
typedef struct TypeNN neuralnet;
```

Ovvero i tipi di dati strutturati per i neurodi, le sinapsi, i layer e la rete neurale.
Definiamole in questo modo:

```
#define     _PRECISION  float

struct TypeNeuron {
    _PRECISION trans_value;
    _PRECISION prop_value;
```

```
    sinapsi* in_links[16];
    int num_in_links;
    sinapsi* out_links[16];
    int num_out_links;
    _PRECISION (*trans_func)(_PRECISION prop_value);
};
```

Come valore di precisione della rete ho deciso di usare *float* (precisione semplice a virgola mobile), ma nulla ci impedisce di usare *int*, *long int* o *double*. Le variabili *prop_value* e *trans_value* non identificano altro che, rispettivamente, il valore di propagazione x_i del neurone e il suo valore di trasferimento, ovvero il valore prodotto in output dalla funzione di trasferimento. Per il resto, dichiaro 16 collegamenti in entrata e 16 in uscita (ovvero 16 sinapsi che collegano il neurone ad altri 16 neuroni in ingresso e altre 16 che collegano il neurone a 16 neuroni in uscita), e tengo il conto delle sinapsi rispettivamente nelle variabili *num_in_links* e *num_out_links*. Infine dichiaro un puntatore a funzione *(*trans_func)*, in modo da poter scegliere successivamente che funzione di trasferimento usare per quello specifico neurone.

```
struct TypeSynapsis {
    _PRECISION delta;
    _PRECISION weight;
    neuron *in,*out;
};
```

Qui dichiaro il tipo sinapsi, caratterizzato da un puntatore al neurone di ingresso e uno a quello di uscita (*in* e *out*), un suo peso sinattico *weight* (corrispondente al w_i che abbiamo considerato finora nelle formule) e una sua *delta*, corrispondente alla variazione dei pesi sinattici in fase di apprendimento della legge di Hebb.

```
struct TypeLayer {
    neuron** elements;
    int num_elements;

    void (*update_weights)(layer* lPtr);
};
```

Un layer consiste in un insieme di neuroni, e conterrà quindi un puntatore alla lista di neuroni che ne fanno parte (*elements*), il loro numero (*num_elements*) e un puntatore a una funzione per aggiornare i pesi sinattici nella fase di apprendimento (*update_weights*).
Una rete neurale semplice è generalmente composta di 3 layer:

- un *layer di input* che prende in ingresso i dati
- un *layer di output* che fornisce i dati elaborati
- uno o più *layer nascosti* (nel nostro caso ne basta uno) che connettono i due layer di input e output. Sono deputati alla fase di elaborazione dei dati

```
struct TypeNN {
    int max_epochs;
    _PRECISION l_rate;

    layer*      input_layer;
    layer*      hidden_layer;
    layer*      output_layer;
};
```

Questa struttura identifica la rete nel suo complesso, con i puntatori ai 3 layer che la compongono e due parametri che useremo in fase di apprendimento. In particolare, *max_epochs* è il numero massimo di *epoche*, ovvero di cicli di aggiornamento dei pesi sinattici, che la rete può effettuare, mentre *l_rate* è il *learning rate* della rete (corrispondente all'η che abbiamo visto negli algoritmi di apprendimento).

Veniamo ora alle funzioni per inizializzare gli elementi della nostra rete:

```
void init_net(neuralnet *net)  {
      net = (neuralnet*) malloc(sizeof(neuralnet));
      max_epochs=1024;  // Valore arbitrario
      l_rate=0.5;  // Valore arbitrario
}

void new_layer(layer *l)  {
      l = (layer*) malloc(sizeof(layer));
      num_elements=0;
}

void new_neuron(neuron *n)  {
      n = (neuron*) malloc(sizeof(neuron));
      num_in_links=0;
      num_out_links=0;
}
```

e ora una funzione per collegare tra di loro i layer:

```
void link_layers(layer* layer_in,layer* layer_out){
    int i,j;
    sinapsi* aux_syn;
    neuron *curr_in,*curr_out;

    for(i=0;i < layer_in->num_elements;i++)  {
        curr_in = layer_in->elements[i];

        for(j=0;j < layer_out->num_elements; j++)  {
            curr_out = layer_out->elements[j];
            aux_syn = (sinapsi*)malloc(sizeof(sinapsi));
            aux_syn->in = curr_in;
            aux_syn->out = curr_out;
            aux_syn->weight = norm(get_rand());
            curr_in->out_links[curr_in->num_out_links++] = aux_syn
            curr_out->in_links[curr_out->num_in_links++] = aux_syn;
        }
    }
}
```

Questa funzione collega tra di loro due layer (*layer_in* e *layer_out*). Per fare ciò alloca memoria per ogni sinapsi tra ogni neurone di *layer_in* e ogni neurone di *layer_out*, attraverso due cicli for (il primo cicla su tutto il layer di input e il secondo su tutto il layer di output). Per ogni collegamento neurone-neurone viene creata una sinapsi, una sinapsi che, ovviamente, dovrà sapere che neuroni collegare, quindi:

```
aux_syn->in = curr_in;
aux_syn->out = curr_out;
```

e i neuroni stessi dovranno sapere che sinapsi utilizzare per collegarsi:

```
curr_in->out_links[curr_in->num_out_links++] = aux_syn
curr_out->in_links[curr_out->num_in_links++] = aux_syn;
```

Per inizializzare il peso della sinapsi ho utilizzato un valore casuale, così calcolato dalla funzione *get_rand()*:

```
float get_rand()  {
     float x,y;

     srand( (unsigned) time(NULL));
     x = (float) rand();
     y = (sin(x)*sin(x))-0.5;

     return y;
}
```

Quello che faccio è inizializzare il seme dei numeri casuali e assegnare alla variabile *x* un numero casuale. Per portare questo valore all'interno dell'intervallo *[-0.5, 0.5]* sfrutto uno stratagemma matematico. Il codominio della funzione seno è in *[-1,1]*, quindi il codominio della funzione sin^2x sarà ovviamente in *[0,1]*. Se sottraggo 0.5 al valore di questa funzione ottengo un valore compreso tra *[-0.5, 0.5]*.

Passiamo ora agli input da fornire alla rete. In questo esempio, forniremo alla rete degli input tramite un file in cui sono salvati dei numeri separati da ';'. La nostra rete dovrà imparare ad effettuare la somma algebrica, quindi nel file i primi due numeri rappresentano le quantità da sommare, e il terzo numero il risultato desiderato. Esempio:

```
1;2;3;5;7;12;3;5;8;.....
```

Dapprima dichiariamo una funzione che apra il file in questione in modalità lettura:

```
#define   TRAINING_FILE    “training.txt”

int open_training_file()  {
      int fd;

      if ((fd=open(TRAINING_FILE, O_RDONLY)) < 0)
            return -1;
      else
            return fd;
}
```

Vediamo ora la funzione *int get_data(float *data, int fd)*. Questa funzione prende come parametri un puntatore a float (in cui verrà salvato il numero letto) e un file descriptor (ottenuto dalla funzione *open_training_file()*), e ritorna -1 in caso di errore, altrimenti salva in *data il numero letto dal file fino al successivo ';'.

```
int get_data(float *data, int fd)  {
      int curr_char;
      int status;
      int is_dec;
      char buf[1],ch;

      // Attenzione: così come è dichiarata questa stringa può essere soggetta
      // a buffer overflow. Imponete voi dei controlli ulteriori per evitarlo,
      // controllando prima quanti caratteri ci sono nel file fino al prossimo
      // ';' e dichiarando la stringa dinamicamente
```

```
        char aux_str[256];

        curr_char=0;

        // Ciclo finché ci sono caratteri da leggere nel file
        while( ( status = read(fd,buf,sizeof(buf)) ) != 0)  {
            // Se status < 0, c'è qualche errore
            if(status<0)if(_DEBUG)perror(strerror(errno));
            ch=buf[0];

            // Se il carattere letto è proprio un ';', esco dal ciclo
            if(ch == ';') break;

            // Altrimenti continuo. Gli a capo sono ininfluenti
            if(ch == '\n')continue;

            // Gli unici caratteri validi al fine della lettura sono . - e tutti
            // i valori numerici. Se il carattere letto non è uno di quelli,
            // ritorno errore
            if(ch != '.'  && ch != '-' && ( ch < 48 || ch > 57 )) {
                if(_DEBUG)fprintf(stderr,"invalid ch %d\n",ch);
                data = NULL;
                return -1;
            }

            // Controllo quanti punti ci sono nel numero
            if( ch == '.' ){
                // Se è già stato trovato un . allora c'è un errore
                if(is_dec){
                    aux_str[curr_char++]=ch;
                    aux_str[curr_char]='\0';
                    fprintf(stderr,"invalid format: two '.' found in
        %s\n",aux_str);
                    return -1;
                }

                // Altrimenti, il numero è decimale
                else
                    is_dec=1;
            }

            // Salvo l'ulteriore carattere letto nella stringa aux_str
            aux_str[curr_char++] = ch;
        }

        // Termino la stringa
        aux_str[curr_char]='\0';

        // Converto la stringa in float e salvo il valore in *data
        *data = atof(aux_str);
        return 0;
}
```

Il codice è già abbastanza commentato, quindi non mi dilungherò ulteriormente.
A questo punto studiamo il modo in cui la rete processa l'output. Quando la rete legge i valori di input, i neuroni del layer di input cambiano di valore, e i nuovi valori che assumono sono quelli della coppia di numeri. A questo punto, l'informazione passerà ai neuroni del layer nascosto, che calcoleranno prima il potenziale post-sinattico quindi il valore di uscita della funzione di trasferimento che, nel nostro caso, è una semplice funzione del potenziale post-sinattico. In particolare, avendo scelto come funzione di trasferimento la funzione identità, avremo

$$y = f(x) = x$$

Per cominciare, facciamo leggere gli input al layer di ingresso:

```
int fd;
int status;
float temp;

// Apro il file con gli input
fd=open_training_file();

// Ciclo su tutti gli elementi del layer di input
for(i=0; i < net->input_layer->num_elements; i++)  {
    // Se la funzione get_data ritorna un valore negativo, allora c'è qualcosa
    // che non va negli input
    if((status = get_data(&temp,fd)) < 0){
        fprintf(stderr,"Invalid input data\n");
        free(net);
        return -1;
    }

    // Il valore del potenziale post-sinattico del neurone è quello appena
    // letto da input, e il valore di trasferimento sarà uguale in virtù della
    // scelta di funzione di trasferimento che abbiamo fatto
    net->input_layer->elements[i]->prop_value=(_PRECISION)temp;
    net->input_layer->elements[i]->trans_value=(_PRECISION)temp;
}
```

Per quanto riguarda invece il layer nascosto

```
void propagate_into_layer(layer* lPtr){
    int i;
    neuron* nPtr;

    // Ciclo for su tutti gli elementi del layer
    for(i=0;i < lPtr->num_elements;i++)  {
        nPtr = lPtr->elements[i];
        // Per ogni neurone calcolo il potenziale post-sinattico...
        nPtr->prop_value = potential(nPtr);
        // ...e la funzione di trasferimento
        nPtr->trans_value = nPtr->trans_func(nPtr->prop_value);
    }
}
```

La funzione *potential()* ha questo codice:

```
_PRECISION potential(neuron* nPtr){
    _PRECISION aux_value=0;
    int i=0;

    // Per ogni sinapsi in ingresso al neurone...
    for(i=0; i<nPtr->num_in_links; i++)  {
        // ...il valore del potenziale è la sommatoria del peso sinattico della
        // sinapsi in questione moltiplicato per il suo valore di trasferimento
        aux_value += (nPtr->in_links[i]->weight * nPtr->in_links[i]->in-
>trans_value);
    }
    return aux_value;
}
```

Per i neuroni appartenenti al layer di output, il discorso è esattamente lo stesso fatto con il layer nascosto, e il codice rimarrà perfettamente identico.

A questo punto, abbiamo già visto che è possibile rendere più preciso il valore di uscita della rete neurale agendo sui singoli pesi sinattici.
Per la struttura che abbiamo dato al file di input, la rete legge dal file sia i valori da sommare sia il risultato esatto, quindi provvederemo a far leggere il risultato giusto alla rete con la funzione *get_data()*. Una volta che abbiamo sia il risultato desiderato, sia il risultato effettivo della rete, opereremo sui pesi sinattici della rete in questo modo:

$$\Delta w_{ij} = -\eta D_j x_i$$

Dove η è una costante della rete compresa tra 0 e 1 chiamata *learning rate*, ed è definita a nostro piacimento (più il valore di η è alto, più la rete modificherà sensibilmente i suoi pesi sinattici in seguito a un errore). Un learning rate alto renderà più veloce l'apprendimento della rete a discapito della precisione, mentre invece un learning rate basso renderà l'apprendimento più lento, ma la rete guadagnerà in fatto di precisione (diciamo pure che un valore intorno a 0.5 rappresenta un buon compromesso). x_i è il valore in input al neurone e D_j (*delta di output*) è così definita:

$$D_j = (y_j - d_j) f'(P_j)$$

dove y_j e d_j sono rispettivamente il valore ottenuto in output e il valore desiderato, ed $f'(P_j)$ è la derivata della funzione di trasferimento calcolata nel potenziale post-sinattico P_j. Questa è la base dell'algoritmo di apprendimento *Widrow-Hoff*, un algoritmo di apprendimento supervisionato che calcola i pesi necessari partendo da pesi casuali, e apportando a questi delle modifiche progressive in modo da convergere alla soluzione finale.

Nel nostro caso, poiché

$$y = f(P) = P \Rightarrow f'(P) = 1$$

avremo semplicemente

$$\Delta w_i = y_i - d_i$$

La variazione verrà calcolata così:

```
_PRECISION compute_output_delta(
      _PRECISION output_prop_value, _PRECISION des_out)  {

    _PRECISION delta;

    delta =
      (output_prop_value - des_out) * linear_derivate (output_prop_value);

    return delta;
}
```

dove *des_out* è il valore desiderato in output e *linear_derivate()* sarà, nel nostro caso, una funzione che ritornerà sempre 1 (ovviamente cambiando la funzione di trasferimento cambierà anche questa funzione).

Per aggiornare i pesi useremo l'equazione appena esaminata:

```
void update_output_weights(layer* lPtr,_PRECISION delta,_PRECISION l_rate){
    int i,j;
    sinapsi* sPtr;
    neuron* nPtr;

    for(i=0; i<lPtr->num_elements; i++)  {
        nPtr = lPtr->elements[i];
        for(j=0;j < nPtr->num_in_links;j++){
            sPtr = nPtr->in_links[j];
            // Δw_ij = − η D_j x_i
            sPtr->delta = -(sPtr->in->trans_value*delta*l_rate);
        }
    }
}
```

Lo stesso algoritmo sarà valido anche per il layer nascosto.
Ora, trovato l'incremento (o decremento) da applicare ai pesi sinattici, basterà ciclare su tutta la rete e applicare a tutte le sinapsi i nuovi pesi:

```
commit_weight_changes(net->output_layer);
commit_weight_changes(net->hidden_layer);
```

con

```
void commit_weight_changes(layer* lPtr){
    int i,j;
    neuron* nPtr;
    sinapsi* sPtr;

    // Ciclo su tutti gli elementi del layer
    for(i=0; i < lPtr->num_elements; i++)  {
        nPtr = lPtr->elements[i];

        // Ciclo su tutte le sinapsi collegate ad un certo neurone
        for(j=0; j < nPtr->num_in_links; j++)  {
            // La sinapsi sarà associata al j-esimo collegamento del neurone
            sPtr = nPtr->in_links[j];

            // Il peso della sinapsi viene aggiornato con il delta
            // appena calcolato
            sPtr->weight += sPtr->delta;

            // Resetto il valore di delta, in modo da potergli applicare
            // nuove modifiche
            sPtr->delta = 0;
        }
    }
}
```

Come intuibile, maggiore sarà il numero di *epoche* (ovvero di iterazioni di questo tipo, in cui modifico il valore dei pesi per convergere sempre più al valore desiderato), maggiore sarà la precisione dei valori di output della rete. Il valore massimo di iterazioni ammissibile l'avevamo precedentemente stabilito all'interno della variabile *max_epochs*.

A questo punto, nel nostro *main()* inseriamo un ciclo che effettua automaticamente questo procedimento per *max_epochs* volte:

```
// Ciclo per max_epochs volte
for(j=0; j<net->max_epochs; j++)  {
    // Leggo i valori in input dal file, con il procedimento già visto
    // in precedenza
    for(i=0; i<net->input_layer->num_elements; i++)  {
        if((status = get_data(&temp,fd)) < 0){
            fprintf(stderr,"errore irreversibile, closing...\n");
            free(net);
            return -1;
        }
        net->input_layer->elements[i]->prop_value=(_PRECISION)temp;
        net->input_layer->elements[i]->trans_value=(_PRECISION)temp;
    }

    // Passo i valori prima al layer nascosto, quindi al layer di output
    propagate_into_layer(net->hidden_layer);
    propagate_into_layer(net->output_layer);

    // Calcolo la delta di output
    if((status = get_data(&des_out,fd)) < 0){
        fprintf(stderr,"errore irreversibile, closing...\n");
        return -1;
    } else {
        out_delta = compute_output_delta(net->output_layer->elements[0]-
>prop_value,des_out);
        update_output_weights(net->output_layer,out_delta,net->l_rate);
    }

    // Calcolo la variazione dei pesi sinattici per il layer nascosto
    // e aggiorno tutti i pesi sinattici
    update_hidden_weights(net->hidden_layer,out_delta,net->l_rate);
    commit_weight_changes(net->output_layer);
    commit_weight_changes(net->hidden_layer);
    net_output = net->output_layer->elements[0]->prop_value;
    printf("DES=%f\tERROR=%f\tOUT=%f\tDELTA=%f\n",des_out,
                (des_out-net_output),net_output,out_delta);
}
```

Ed ecco che la nostra rete neurale è pronta per l'uso.

Ringraziamenti

Questo tutorial è nato per essere un'introduzione ad un argomento vasto e complesso come quello delle reti neurali, un argomento in cui si fondono insieme l'informatica, il formalismo matematico e le teorie neurologiche. Ho voluto scrivere questo tutorial con l'intento di non fornire solo una grande mole di conoscenze teoriche senza mostrare applicazioni pratiche della teoria stessa, e allo stesso tempo di mostrare esempi di applicazione della teoria senza sorvolare sul bagaglio formale e matematico che è alle sue spalle. Per quanto riguarda la parte teorica, devo ringraziare il formalismo e la chiarezza espositiva dei *Sistemi Fuzzy* di Silvio Cammarata, il libro che mi ha introdotto alla teoria delle reti neurali e dell'intelligenza artificiale. Per quanto riguarda invece gli esempi pratici e gli algoritmi che corredano il testo, ringrazio l'ottima guida *Imparare il C – una guida per Linux* di M.Latini e P.Lulli. Nonostante programmi in C da molti anni, credo che la loro sia la guida più completa sull'argomento nella nostra lingua.

BlackLight, © 2007

Per suggerimenti, chiarimenti o errata corrige, contattate l'autore all'indirizzo
blacklight86@gmail.com