Le reti neurali

Guidati dall'ambizione per la comprensione e per la riproduzione delle funzionalità del cervello, in molti campi della scienza e dell'ingegneria, sia in ambito accademico, sia nell'industria, lo studio delle reti neurali ha suscitato un larghissimo interesse. In tali strutture di elaborazione, l'eccitazione del progresso tecnologico è integrata dall'evocatore ed a volte sinistro, fremito di riprodurre l'intelligenza stessa

di Marco Pirrone e Andrea Pompili

Marco Pirrone *è laureato in Ingegneria Informatica all'università "La Sapienza" di Roma. Attualmente svolge il dottorato di ricerca in Intelligenza Artificiale presso il Dipartimento di "Informatica e Sistemistica" della Sapienza ed in particolare si occupa di Visione Artificiale e Navigazione Robotica Autonoma. Può essere contattato per e-mail all'indirizzo mpirrone@infomedia.it*

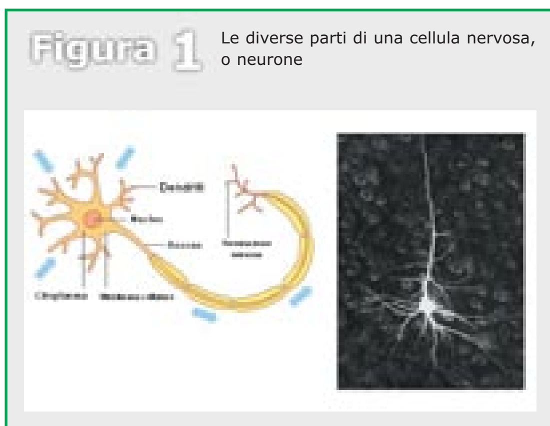
Intelligenza artificiale

Il confronto tra le capacità di un calcolatore elettronico e quelle del cervello umano ha fornito risultati particolarmente sorprendenti. Calcoli numerici praticamente improponibili ad un essere umano vengono portati a termine da un calcolatore in una frazione di secondo. Viceversa, compiti banali per un essere umano, come per esempio il riconoscimento di immagini, costituiscono per il calcolatore degli obiettivi di una difficoltà estrema.

I calcolatori convenzionali lavorano su dati espressi in forma digitale attraverso sequenze precise, forniscono risposte esatte ad ogni problema e conservano informazioni in modo tale da poter recuperare anche un singolo dato. I sistemi neurali artificiali, invece, prendono decisioni sulla base di dati incompleti e contraddittori, forniscono risposte veloci ma approssimate a problemi molto complessi e conservano l'informazione in modo tale da recuperare assieme al dato richiesto anche altre informazioni ad esso collegate.

I sistemi di elaborazione seriale sono molto utili ed efficienti nel risolvere compiti in cui gli esseri umani trovano normalmente difficoltà (come, ad esempio, la soluzione di calcoli matematici, la rotazione di complicate figure geometriche, la memorizzazione di enormi quantità di dati), ma non si possono definire intelligenti: l'unico elemento di intelligenza nell'intero processo è il programmatore che ha analizzato il compito e ha creato il programma. Affinché un sistema di intelligenza artificiale possa definirsi tale, esso dovrebbe per lo meno essere in grado di risolvere i problemi che gli esseri umani trovano semplici e naturali. Mediante l'uso delle reti neurali si cerca quindi di sviluppare modelli che abbiano le caratteristiche fondamentali e le capacità elaborative proprie del cervello umano.

Figura 1 Le diverse parti di una cellula nervosa, o neurone



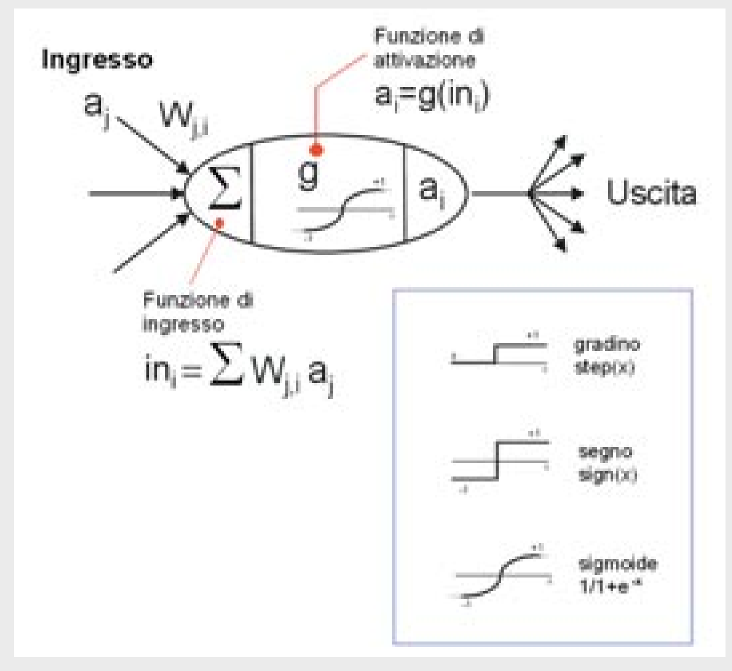
COME FUNZIONA IL CERVELLO

Il modo esatto in cui il cervello rende possibile il pensiero è uno dei grandi misteri della scienza: sappiamo che il neurone o cellula nervosa, è l'unità fondamentale di tutto il tessuto del sistema nervoso, cervello incluso. Nel cervello umano sono presenti un numero enorme di neuroni di diversi tipi e questa quantità è stimabile intorno a 150000 neuroni per mm^2. Considerando che l'area totale della corteccia cerebrale misura intorno ai 200000 mm^2 circa, possiamo stimare almeno $3*10^{11}$ neuroni.

Ciascun neurone (**Figura 1**) è costituito da un corpo cellulare o soma, che contiene il nucleo della cellula. Dal corpo cellulare si ramifica un gran numero di fibre chiamate dendriti ed una singola fibra chiamata assone, il quale si connette ai dendriti ed ai corpi cellulari degli altri neuroni (la giunzione si chiama sinapsi).

I segnali si propagano da neurone a neurone grazie ad una complicata reazione elettrochimica il cui effetto è di aumentare o ridurre il potenziale elettrico della cellula neuronale. Se il potenziale elettrico supera un certo valore di soglia il neurone "manda in uscita" un impulso elettrico propagando quindi il segnale ai neuroni adiacenti che reagiranno alla stessa maniera. Dato che, da un punto di vista elettrico, un neurone può essere visto come un dispositivo ad uscita binaria (assenza o presenza di segnale d'uscita), è possibile descrivere una relazione tra ingressi e uscita della cellula neuronale mediante un'equazione matematica. Un neurone può essere quindi visto come una cella elaborativa non programmabile che computa sempre la stessa equazione. Nonostante la relazione ingresso-uscita per ciascun neurone rimanga sempre la stessa, può cambiare la *forza* delle connessioni tra loro, ossia l'efficacia con cui un neurone può eccitare i suoi simili. Attraverso l'esperienza e l'apprendimento, la rete neurale del cervello modifica poi la propria struttura eccitando o inibendo le connessioni tra i neuroni in modo da ottenere risultati diversi.

Figura 2 Modello di neurone artificiale. È possibile utilizzare diverse funzioni di attivazione prendendo come esempio quelle mostrate (gradino, segno, ecc.)



LE RETI NEURALI ARTIFICIALI

Le reti neurali artificiali sono modelli computazionali costituiti da elementi di elaborazione (ispirati ai neuroni) connessi tra loro tramite collegamenti (ispirati alle sinapsi) in modo da formare uno schema assimilabile ad un semplicissimo tessuto nervoso.

Per definizione, le reti neurali artificiali (d'ora in avanti semplicemente reti neurali) sono strutture parallele e distribuite per il trattamento dell'informazione, consistenti in elementi di elaborazione interconnessi per mezzo di canali unidirezionali chiamati connessioni.

Ogni connessione ha un peso numerico associato, il cui valore varia in base all'esperienza maturata. I pesi diventano il principale mezzo di memorizzazione a lungo termine per le reti neurali e l'apprendimento di un particolare "comportamento" si ottiene in genere attraverso il loro aggiornamento.

A partire da osservazioni di carattere biologico è stato definito un modello generale di neurone artificiale (**Figura 2**) che può essere visto come una unità di elaborazione che effettua una somma pesata dei segnali di ingresso ed esegue una trasformazione non lineare del risultato ottenuto.

In particolare, quindi, ogni neurone (o più semplicemente nodo) è composto da:

- Un insieme di collegamenti di ingresso che provengono da altre unità.
- Una singola connessione di uscita che si dirama in diverse connessioni secondarie, ognuna delle quali trasporta lo stesso segnale (il segnale di uscita dell'unità).
- Una funzione che definisca il segnale di uscita dell'unità stessa (o livello di attivazione) dati gli ingressi (a_j: valore d'ingresso proveniente dall'uscita dell'unita *j*) e i rispettivi pesi ($W_{j,i}$: peso associato al collegamento dall'unità *j* all'unità *i*). Il calcolo è suddiviso in due componenti: in primo luogo una componente *lineare*, chiamata funzione d'ingresso che effettua la somma pesata dei valori degli ingressi all'unità *i*. Quindi una componente non lineare *g*, chiamata funzione di attivazione (una funzione elementare tipo la funzione gradino, segno o la cosiddetta sigmoide) che calcola il valore di uscita dell'unità a partire dalla somma pesata precedente.

Alcune unità possono essere collegate con l'ambiente esterno e possono essere designate come unità di ingresso o di uscita.

L'elaborazione dell'informazione è locale in ciascun nodo, ossia il risultato deve dipendere solamente dai valori correnti dei segnali d'ingresso che giungono all'unità (tramite connessioni entranti) e dai valori memorizzati nella memoria locale dell'unità stessa.

STRUTTURA DELLE RETI

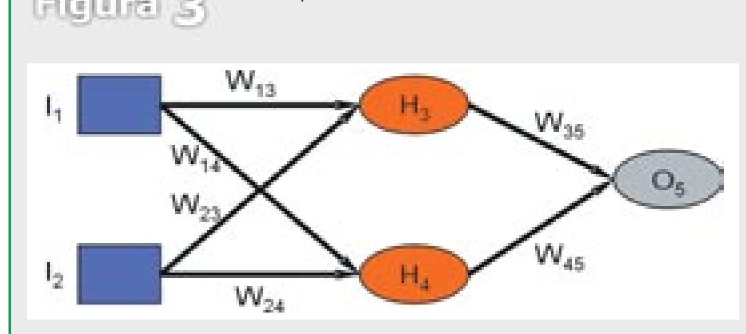
Per costruire una rete neurale si deve decidere che tipo di unità adottare e come interconnettere ciascuna unità. Generalmente la distinzione tra i diversi modelli è basata sulla struttura interna dei nodi per cui vengono individuate due classi:

- Reti feed-forward
- Reti ricorrenti

Nelle reti di tipo *feed-forward* i neuroni sono organizzati in strati (layer) e le connessioni presenti sono esclusivamente da uno strato al successivo (come un grafo diretto aciclico). Nessun nodo può essere connesso ai nodi del medesimo strato o ad uno strato precedente, oppure saltare lo strato successivo.

La **Figura 3** mostra un esempio molto semplice di rete feed-forward con due *strati* (poiché le unità di ingresso servono sola-

Figura 3 Un esempio di rete feed-forward a due strati



Andrea Pompili *è laureato in Ingegneria Informatica. Ha svolto attività di analisi e sviluppo per la realizzazione di portali di tipo enterprise su piattaforma Java. Attualmente lavora presso WIND Telecomunicazioni come referente per la sicurezza informatica. Può essere contattato per e-mail all'indirizzo apompili@infomedia.it*

mente a passare le attivazioni allo strato successivo essi non vengono considerati nel conto), due ingressi, due nodi nascosti (nodi senza collegamento diretto con l'ambiente esterno) e un nodo di output. Con un numero sufficientemente grande di unità nascoste ed un solo strato interno si può rappresentare una qualunque funzione continua; con due strati interni si possono invece rappresentare anche funzioni discontinue. Le reti senza unità nascoste sono chiamate *percettroni* e sono state le prime tipologie di rete studiate.

I sistemi neurali artificiali prendono decisioni sulla base di dati incompleti e contraddittori

Nelle reti *ricorrenti*, invece, i collegamenti possono formare configurazioni arbitrarie: ad esempio l'ingresso di un neurone può dipendere dall'uscita del neurone stesso o di nodi appartenenti agli strati precedenti. La caratteristica principale di queste reti è di avere una sorta di "memoria a breve termine" dovuta al fatto che ogni nodo mantiene uno stato interno (se l'uscita di una unità viene trasferita in ingresso a lei stessa esiste uno stato interno immagazzinato sotto forma di livelli di attivazione). Per esempio, il cervello umano non può essere solamente una rete feed-forward poiché in questo caso non potremmo avere una memoria a breve termine. Alcune regioni del cervello sono largamente alimentate in avanti ed in qualche modo stratificate, ma sono presenti anche delle evidenti connessioni all'indietro. Quindi, nella nostra terminologia, il cervello può essere formalizzato come una rete ricorrente.

Le reti ricorrenti possono diventare instabili, oscillare o comportarsi in modo caotico. La classe meglio compresa di reti ricorrenti è costituita delle *reti di Hopfield*. Esse usano connessioni bidirezionali con pesi simmetrici ($W_{i,j} = W_{j,i}$).

ALGORITMI DI APPRENDIMENTO

Definita la struttura della rete, diviene necessario inizializzare i pesi dei vari collegamenti e provvedere ad "addestrarla" per risolvere il particolare compito per cui è stata progettata. L'apprendimento di una rete avviene modificando opportunamente i valori dei pesi in modo da rende-

Listato 1 Codice per l'implementazione di una Rete Neurale base in C++. Si definisce una serie di array multidimensionali per i pesi e i valori intermedi calcolati dalla rete

```
//file: FFnet.cpp

#include <math.h>
#include <time.h>
#include <stdlib.h>

#define BETA  1

double actfunction(double x) {
  return 1.0 / (1.0 + exp(-BETA * x));
}

double derivate_actfunction(double x) {
  return BETA * x * (1-x);
}

// Struttura per un training example
// ------------------------
struct example {
  double* input;
  double* target;
  example* next;
};

// Classe generica per reti feed-forward
// ------------------------
class FFnet {
  protected:
    example* training_set;  //training set
    double ***W;             //matrice dei pesi
    double **X;              //matrice dei risultati intermedi
    int nL;
    int* dL;

  public:
    FFnet(int* dimLayer, example* tset);
    ~FFnet();

    double* value(double* input);
    double get_error(example* ex);
    double get_error();
    void init_weights();

    int learn_error(double soglia, int max_epoche);
    virtual void learn(example* ex) {};
};

// Costruttore
// ------------------------
FFnet::FFnet(int* dimLayer, example* tset) {
  int l,j;
  dL = dimLayer;
  training_set = tset;
  nL = 0;

  while(dimLayer[nL]) nL++;

  //definisce gli array per le uscite intermedie
  X = new double*[nL];
  for(l = 0; l < nL; l++) {
    X[l] = new double[dL[l] + 1];
    X[l][dL[l]] = 1;
  }

  //definisce l'array dei pesi (multidimensionale)
  W = new double **[nL];
  for(l = 1; l < nL; l++) {
    W[l] = new double *[dL[l]];
    for(j = 0; j < dL[l]; j++) {
      W[l][j] = new double[dL[l - 1] + 1];
    }
  }
}

// Distruttore
// ------------------------
FFnet::~FFnet() {
  int l,j;
  for(l = 0; l < nL; l++) delete X[l];
  for(l = 1; l < nL; l++) {
    for(j = 0; j < dL[l]; j++) delete W[l][j];
    delete W[l];
  }
```

...continua...

Listato 1 ...segue...

```
    delete X;
    delete W;
}

// Calcola l'output della rete in base ad un array di input
// ------------------------
double* FFnet::value(double* input) {
    int i,j,l;
    double s;

    for(i = 0; i < dL[0]; i++) X[0][i] = input[i];

    for(l = 1; l < nL; l++) {
      for(i = 0; i < dL[l]; i++) {
        s = 0;
        for(j = 0; j <= dL[l - 1]; j++) {
          s += W[l][i][j] * X[l - 1][j];
        }
        X[l][i] = actfunction(s);
      }
    }
    return X[nL - 1];
}

// Inizializza i pesi con valori randomici
// ------------------------
void FFnet::init_weights() {
    srand((unsigned)time(NULL) );
    int i,j,k;
    for(i = 1; i < nL; i++)
      for(j = 0; j < dL[i]; j++)
        for(k = 0; k <= dL[i - 1]; k++)
            W[i][j][k] = (rand() - RAND_MAX) / RAND_MAX ;
}

// Calcola l'errore della rete per l'esempio passato
// ------------------------
double FFnet::get_error(example* ex)  {
    double *output = value(ex->input);
    double err = 0;
    double tmp;
    for (int i = 0; i < dL[nL - 1]; i++) {
      tmp = output[i] - ex->target[i];
      err += tmp * tmp;
    }
    return (0.5 * err) / dL[nL - 1];
}

// Calcola l'errore globale della rete su tutto il training set
// ------------------------
double FFnet::get_error() {
    double err = 0;
    int count = 0;
    example* tmp = training_set;

    while (tmp) {
      err += get_error(tmp);
      tmp = tmp->next;
      count++;
    }
    return (err/count);
}

// Inizia l'addestramento della rete e imposta un valore di soglia
// per bloccare l'iterazione
// ------------------------
int FFnet::learn_error(double soglia, int max_epoche) {
    example* tmp;
    int count = 0;
    double err = get_error();

    while (err > soglia) {
      if (++count >= max_epoche) return -1;
      tmp = training_set;

      while(tmp) {
        learn(tmp);
        tmp = tmp->next;
      }
      err = get_error();
    }
    return count;
}
```

re minimo l'errore commesso in relazione ad un insieme prefissato di esempi. La procedura che realizza in maniera automatica un apprendimento di questo tipo viene detta *algoritmo di addestramento* e costituisce la parte essenziale della rete neurale artificiale. Dalla sua efficienza e affidabilità dipende infatti la capacità di quest'ultima di acquisire conoscenza.

Le tecniche di apprendimento possono essere suddivise in due categorie:

- Apprendimento supervisionato
- Autorganizzazione o apprendimento non supervisionato

L'*apprendimento supervisionato* richiede un insieme di esempi per i quali l'uscita desiderata della rete è nota a fronte di un particolare ingresso. Il processo di apprendimento consiste quindi nell'adattare la rete in modo che essa fornisca la risposta corretta per il set di esempi. La rete risultante dovrebbe poi essere in grado di generalizzare (dare una buona risposta) quando vengono sottomessi esempi non contemplati tra quelli di addestramento.

Il processamento dell'informazione eseguito da ciascun nodo è di tipo locale

Nell'*apprendimento non supervisionato* l'apprendimento della rete neurale è autonomo: essa processa i dati che le vengono presentati, estrae alcune delle loro proprietà ed impara a riflettere queste proprietà sulle sue uscite. La rete neurale non fa altro che configurare sé stessa in modo tale che ogni neurone a livello output risponda in modo simile per input simili tra loro, ossia in qualche modo correlati.

Tutti gli algoritmi di apprendimento presentano comunque degli aspetti generali comuni:

- I valori iniziali dei pesi della rete vengono assegnati in modo casuale entro un piccolo campo di variazione (ad

esempio [-0.1, 0.1]) oppure vengono fissati tutti a zero.

- L'apprendimento consiste nella presentazione ripetuta di una serie di input (training set). La convergenza della rete (ossia arrivare ad un equilibrio per cui la rete si comporta esattamente come voluto) può essere raggiunta, infatti, elaborando più volte gli stessi input e modificando ad ogni ciclo i pesi di una quantità predefinita. Ciascun ciclo viene chiamato *epoca*.
- La velocità di apprendimento è regolata da una costante detta *tasso di apprendimento* che controlla la porzione di modifica che viene applicata ai valori dei pesi.

Per le reti feed-forward l'algoritmo di apprendimento supervisionato più utilizzato è sicuramente il *Back-Propagation* (formalizzato nel 1969 da Bryson e Ho). Questo algoritmo utilizza una tecnica di ricerca basata sul gradiente che minimizza una funzione costo uguale alla differenza media quadrata tra gli output desiderati e quelli generati dalla rete. Nel back-propagation la funzione di attivazione di un neurone artificiale deve essere una funzione continua e differenziabile (ad esempio, la sigmoide).

Operativamente la tecnica di back-propagation può essere suddivisa in quattro fasi:

- Inizialmente viene presentata allo strato di ingresso della rete una matrice contenente i segnali di ingresso e vengono calcolate le uscite di ciascun singolo strato fino ad ottenere il segnale di uscita.
- Il vettore di uscita finale *Out* viene confrontato con il segnale di riferimento conosciuto *T* (contenuto nel set degli esempi utilizzato) e viene calcolato il vettore dei residui $E = Out - T$ e la somma *e* dei quadrati delle componenti del vettore dei residui ($e = \sum(E_i)^2$). Il processo di apprendimento della rete viene interrotto se *e* risulta minore di un valore prefissato o se il numero massimo di iterazioni (epoche) stabilito per l'apprendimento viene raggiunto.
- Vengono calcolati i nuovi valori dei pesi nello strato di output e negli strati interni e si ricomincia il processo.

Per gli aggiornamenti si utilizza il vettore dei residui per correggere i pesi dello strato di output, poi si ripartisce l'errore residuo su tutti gli strati interni ricorsivamente fino a raggiungere lo strato di input. Se si numerano gli strati da 0 a *N* (dove 0 corrisponde allo strato di input e *N* corrisponde allo strato di output) si può applicare ricorsivamente una funzione di aggiornamento partendo dallo strato di output fino a raggiungere quello di input.

"...le reti neurali sono il secondo miglior modo per fare bene praticamente qualsiasi cosa..."

In particolare, per lo strato di output la funzione di aggiornamento è la seguente:

$in_{i,N} = \sum_j(W_{ij,N} * a_{j,N})$

$\Delta_{i,N} = E_i * g'(in_{i,N})$

$W_{ij,N} = W_{ij,N} + alfa * a_{j,N} * \Delta_{i,N}$

mentre per gli strati interni è:

$in_{i,k} = \sum_j(W_{ij,k} * a_{j,k})$

$\Delta_{i,k} = g'(in_{i,k}) * \sum_j(W_{i,j,k+1} * \Delta_{j,k+1})$

$W_{ij,k} = W_{ij,k} + alfa * a_{j,k} * \Delta_{i,k}$

dove $W_{ij,k}$ e $a_{j,k}$ identificano il peso W_{ik} e l'ingresso a_j per il nodo *i* del layer *k*, *g'* rappresenta la derivata della funzione di attivazione scelta e *alfa* una costante che identifica il tasso di apprendimento. Il valore $in_{i,k}$ identifica la somma pesata degli ingressi al nodo *i* dello strato *k* e $\Delta_{i,k}$ il delta di correzione da applicare ai pesi associati al nodo *i* sempre dello strato *k*.

UN PO' DI CODICE: IMPLEMENTAZIONE DI UNA RETE IN C++

A questo punto abbiamo tutte le informazioni necessarie per creare una rete neurale feed-forward di base che sfrutti il back-propagation come algoritmo di apprendimento. Per ottimizzare le performance si può rappresentare la rete organizzando i pesi di ciascun nodo in un array multidimensionale: la prima relativa allo strato, la seconda al nodo e la terza ad una delle connessioni con lo strato precedente. Inoltre si può introdurre un altro array multidimensionale per immagazzinare gli output di ciascuna cella per ciascuna epoca in modo da ottimizzare i successivi calcoli per l'algoritmo di back-propagation (si ricorda che l'algoritmo prevede di utilizzare gli output di ciascuno strato per modificare i pesi del precedente). Nel **Listato 1** viene riportato il codice C++ per una rete neurale base costituita da un numero arbitrario di livelli di dimensione qualsiasi, ciascuno completamente connesso con il livello successivo e costituito da neuroni con funzione di attivazione di tipo sigmoide ($Y = 1 / 1 + exp(-beta * X)$). La funzione di attivazione utilizza un valore *beta* per modificare l'inclinazione della sigmoide: al suo aumentare la funzione si avvicina sempre più alla funzione a gradino centrata sullo zero, mentre al suo diminuire la funzione si comporta in modo lineare.

Il training set viene rappresentato con una lista di record di tipo *example* che contengono gli input e gli output previsti sotto forma di array. Partendo da questo esempio è possibile estendere la classe principale per generare reti personalizzate che ridefiniscano l'algoritmo di learning o altre funzioni specifiche. In particolare nel **Listato 2** viene mostrata un'implementazione di una rete basata sull'algoritmo back-propagation descritto precedentemente. Per ottenere lo scopo si introduce un ulteriore array per rappresentare i delta di variazione per l'aggiornamento dei pesi e si ridefinisce la funzione di learning implementando l'algoritmo descritto nel paragrafo precedente. Il difetto principale di questa formalizzazione è la staticità della struttura che, una volta definita, non può essere modificata durante l'addestramento (ad esempio aggiungendo o eliminando nodi). Questa limitazione imporrà la scelta di un altro modello di rappresentazione nel caso si voglia aumentare l'adattabilità della rete e la sua complessità elaborativa.

Listato 2 Implementazione di una rete basata sull'algoritmo di Back-propagation

```
//file: BPnet.cpp

#include "FFnet.cpp"

// Classe per reti feed-forward con back-propagation
// -------------------------
class BPnet : public FFnet {
  protected:
    double learn_rate;  //velocita' di apprendimento
    double **delta;     //matrice degli errori

  public:
    BPnet(int* dimLayer, example* tset, double rate = 0.3);
    ~BPnet();

    void learn(example* ex);  //ridefinisce la funzione di learning
};

// Costruttore
// -------------------------
BPnet::BPnet(int* dimLayer, example* tset, double rate) : FFnet(dimLayer,
                                                              tset) {
  learn_rate = rate;
  init_weights();

  delta = new double*[nL];
  for(int i = 0; i < nL; i++)
    delta[i] = new double[dL[i] + 1];
}

// Distruttore
// -------------------------
BPnet::~BPnet() {
  int i,j,l;
  for(i = 0; i < nL; i++) delete delta[i];
  delete delta;
}

// Algoritmo di back propagation
// -------------------------
void BPnet::learn(example* ex) {
  int i,j,k;
  double s;
  double* output;

  //propagate
  output = value(ex->input);

  //backpropagate
  for(j = 0; j < dL[nL - 1]; j++)
      delta[nL - 1][j] = derivate_actfunction(output[j])
                        * (ex->target[j] - output[j]);

  for(i = nL - 2; i > 0; i--) {
    for(j = 0;j<dL[i];j++) {
      s = 0;
      for(k = 0; k < dL[i + 1]; k++) {
        s += W[i+1][k][j] * delta[i+1][k];
      }
      delta[i][j] = s * derivate_actfunction(X[i][j]);
    }
  }

  //aggiorna pesi
  for(i = 1; i < nL; i++) {
    for(j = 0; j < dL[i]; j++) {
      for(k = 0; k <= dL[i-1]; k++) {
        // gradient descent
        W[i][j][k] += learn_rate * delta[i][j] * X[i-1][k];
      }
    }
  }
}
```

ALCUNE APPLICAZIONI DELLE RETI NEURALI

Normalmente la risoluzione di un problema può essere effettuata in maniera ottima se si riesce a formalizzare il problema stesso in maniera teorica, ad esempio mediante un modello matematico. Nei casi in cui questo non è possibile ci si può orientare verso una struttura di elaborazione basata su esempi. Tali strutture utilizzano un insieme di esempi di riferimento (*training set*) come esperienza passata sulla quale basarsi per la risoluzione del problema in questione.

In particolare un training set è un insieme di coppie ingresso-uscita che corrispondono a dei valori desiderati o osservati nella situazione reale. Le reti neurali rientrano proprio in questa categoria di risolutori di problemi. Esempi classici di problemi risolvibili mediante rete neurale sono: pronuncia di testi scritti da parte di un calcolatore (il programma *NETtalk* si basa su una rete neurale che impara a pronunciare il testo scritto), riconoscimento di caratteri scritti a mano (sono stati implementati algoritmi basati su reti neurali per la lettura dei codici postali scritti a mano sul dorso delle buste da lettera), guida di un veicolo (*ALVINN* è una rete neurale utilizzata per controllare i veicoli del *NavLab* alla Carnegie Mellon University) .

LE RETI DINAMICHE E MR. ARMHANDONE

Come già accennato, uno degli svantaggi fondamentali degli algoritmi di apprendimento classici, consiste nella necessità di fissare a priori il numero e la grandezza degli strati della rete. Questo potrebbe portare ad un sopra o sotto-dimensionamento della struttura che necessita di un ulteriore ingegnerizzazione a posteriori. Per evitare questi inconvenienti ci si può orientare verso algoritmi di costruzione dinamica che modellino la struttura della rete durante l'addestramento stesso (questo tipo di reti vengono denominate "reti neurali artificiali dinamiche"). Queste classi di algoritmi includono sia tecniche di eliminazione dei nodi o pesi delle connessioni che non giocano un ruolo significativo, sia tecniche di inserimento di ulteriori nodi. L'idea base di tali algoritmi può essere illustrata rispondendo alle seguenti domande:

- Quando bisogna bloccare l'addestramento per stabilire le esigenze della rete?
- Premesso che lo split di una unità neurale consiste nella creazione, a partire da un nodo chiamato "template", di due nuovi nodi i cui pesi delle

Il progetto Mr. ArmHandOne concerne la realizzazione di un robot mobile capace di svolgere azioni intelligenti in un ambiente sconosciuto

Tutti i problemi in cui l'esperienza del passato è elemento determinante per arrivare ad una corretta soluzione sono applicazioni possibili per le reti neurali

Figura 4 Il robot Mr. ArmHandOne usato dal dipartimento di informatica e sistemistica dell'università "La Sapienza" di Roma

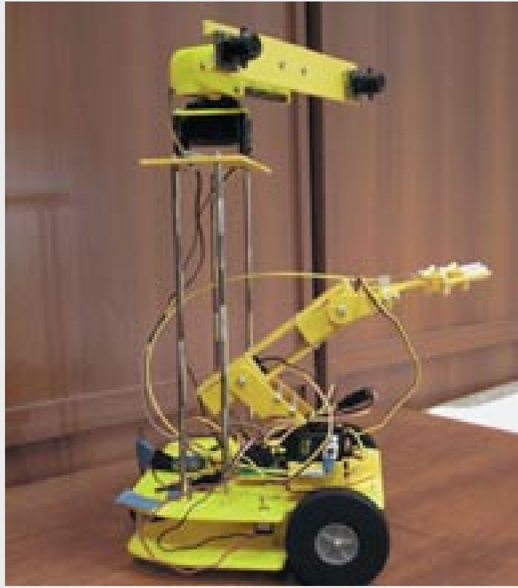


Figura 5 Il robot Mr. ArmHandOne con tutti i sensori montati

connessioni entranti sono determinati in modo da migliorare la classificazione degli esempi con cui la rete viene addestrata, come occorre identificare un nodo da splittare o da eliminare?

- Quale meccanismo deve essere utilizzato per aggiungere un nuovo nodo e come determinare i pesi di quest'ultimo?
- Come si deve procedere nel caso in cui l'eliminazione o lo split di un nodo ha provocato la crescita e non la diminuzione dell'errore globale della rete?

Normalmente la crescita della rete avviene quando si verificano le seguenti tre condizioni:

- La rete è caduta in un minimo globale, situazione che viene verificata dal confronto dell'errore della rete attraverso una finestra temporale di 'n' iterazioni con l'errore che si ha subito dopo la creazione di un nuovo nodo.
- L'errore che la rete commette è sempre maggiore di una tolleranza prestabilita.
- Il fattore di crescita relativo è minore del fattore di crescita di riferimento (essendo i fattori di crescita dei parametri che tengono conto di quanti nodi è cresciuta la rete a partire da una struttura minima iniziale e di quanti può anco-

Listato 3 Header C++ per la rappresentazione di una rete neurale mediante un grafo orientato. Le classi implementano una serie di nodi di tipo Neurone connesse da archi che rappresentano i pesi delle connessioni della rete

```
//file: graph_net.h

#include <math.h>
#include <stdlib.h>

#define TRUE  1
#define FALSE 0

/* Definizione delle funzioni di attivazione di base */
double sigmoid(double x) { return 1.0/(1.0 + exp(-x)); }
double dsigmoid(double x) {
  double t = 1 + exp(-x);
  return exp(-x)/(t*t);
}

class Neuron;
class Weight;

/* Classi di servizio */
class WeightList {
  private:
    Weight* _weight;
    WeightList* _next;

  public:
    WeightList(Weight* w, WeightList* wl = NULL) { _weight = w; _next = wl; }
    Weight* weight() { return _weight; }
    WeightList* next() { return _next; }
};

class NeuronList {
  private:
    Neuron* _neuron;
    NeuronList* _next;

  public:
    NeuronList(Neuron* n, NeuronList* nl = NULL) { _neuron = n; _next = nl; }
    Neuron* neuron() { return _neuron; }
    NeuronList* next() { return _next; }
};

/* Classe relativa ad un link pesato tra due nodi */
class Weight {
  private:
    Neuron *_from,*_to;
    double w;
```

...continua...

Listato 3 ...segue...

```
  public:
    Weight() { _from = NULL; _to = NULL; w = 0.2 * rand()/RAND_MAX - 0.1; }

    double value();     //restituisce il peso moltiplicato per il valore di
                                                  uscita del nodo origine
    double gradient();
    double get() { return w; }
    void set(double x);

    Neuron* from() { return _from; }
    Neuron* to() { return _to; }
};

/* Classe relativa ad un neurone (nodo) */
class Neuron {
  private:
    WeightList* entering_links;   /* l'insieme dei link entranti nel nodo */
    WeightList* exiting_links;    /* l'insieme dei link uscenti dal nodo */
    double _sum, _out;
    int changed;

  public:
    Neuron(WeightList* pwl = NULL, WeightList* swl = NULL);

    virtual double value();       /* restituisce il valore di output
                                                          del neurone */
    virtual double delta();       /* restituisce la derivata parziale per
                                                               la BP */
    virtual double delta(double derr); /* restituisce la derivata sull'errore
                                                            di output*/
    int is_changed() { return changed; }
    void set_changed(int x);
};

/* Classe relativa ad un neurone di input */
class InputTerminal: public Neuron {
  private:
    double in;  /* il valore di input */

  public:
    InputTerminal(double s = 0):Neuron(),in(s) {}
    InputTerminal(WeightList* pwl, WeightList* swl, double s=0):Neuron(pwl,
                                                         swl),in(s) {}

    virtual double value() {
      set_changed(FALSE);
      return in;
    }
    virtual double delta(double s=0) { return 0; }

    void set(double x) {
      if (in != x) set_changed(TRUE);
      in = x;
    }
};

/* Classe rappresentante una rete neurale complessa */
class FFNeuralNet {
  public:
    NeuronList neurons;   /* insieme di tutti i neuroni */
    NeuronList output;    /* neuroni di output */
    NeuronList input;     /* neuroni di input  */

    WeightList weights;   /* lista delle connessioni */

    //...
};
```

ra crescerne fino al termine dell'addestramento).

Nel caso in cui si sia giunti alla conclusione che la rete deve crescere, occorre stabilire come. Al fine di individuare il nodo da splittare si stabilisce una funzione che dia un'indicazione della rilevanza di un nodo nella rete ai fini dell'apprendimento. Il nodo da splittare è quello per il quale la funzione di rilevanza risulta maggiore rispetto a quella di tutti gli altri nodi.

Se la funzione di rilevanza del nodo individuato è minore di una soglia prestabilita dall'utilizzatore, si procede alla creazione di un nuovo nodo in un nuovo strato nascosto o nello strato che ha il minor numero di elementi nascosti (solo nel caso di una struttura non uniforme). Se esiste un nodo da splittare si procede all'aggiornamento dei pesi mediante un meccanismo opportuno.

Al contrario, l'eliminazione di un nodo dalla rete o pruning avviene se:

- L'errore totale commesso dalla rete è minore di una tolleranza prestabilita.
- Il fattore di crescita relativo è maggiore di zero.

Utilizzando la stessa funzione di rilevanza utilizzata per lo splitting, viene individuato ed eliminato il nodo per cui tale funzione risulta minore rispetto a quella degli altri nodi presenti nella rete. Se dopo un'eliminazione e trascorsi n cicli, l'errore della rete ha un valore maggiore rispetto a quello commesso prima del pruning, si procede al ripristino della rete precedente. Ciò determina l'interruzione di ulteriori cambiamenti nella struttura della rete fino al termine dell'addestramento.

Per comprendere meglio l'utilizzo delle reti neurali dinamiche, possiamo far riferimento all'uso che ne è stato fatto nell'ambito di un progetto, denominato *Mr. ArmHandOne*, sviluppato presso il Dipartimento di Informatica e Sistemistica dell'università di Roma "La Sapienza".

In tale contesto, i componenti dall'*A.L.C.O.R. Group* (Auto-agent Laboratory for COgnitive Robotics), si sono adoperati per la realizzazione di agente autonomo capace di svolgere azioni intelligenti in un ambiente sconosciuto.

Per consentire a Mr. ArmHandOne (**Figure 4** e **5**) di muoversi liberamente nello spazio circostante, è stata implementata, mediante una rete neurale dinamica, una tecnica di fusione di dati sensoriali ridondanti che permette all'agente di costruire una mappa del proprio ambiente di lavoro.

Nei **Listati 3** e **4** viene mostrato un possibile modello C++ per una rete neurale dinamica. La tecnica è di rappresentare l'intera rete come un grafo orientato per

Listato 4 Implementazione delle funzionalità specifiche in C++ per la rappresentazione di una rete neurale mediante un grafo orientato

```
//file: graph_net.cpp

#include <math.h>
#include <stdlib.h>
#include "graph_net.h"

// Metodi definiti nella classe Weight
// - - - - - - - - - - - - - - - - - - - - - - - - -
double Weight::value() {
    return w * (from()->value());
}

double Weight::gradient() {
  return to()->delta() * from()->value();
}

void Weight::set(double x) {
  if (w != x) {
    w = x;
    to()->set_changed(TRUE);
  }
}

// Metodi definiti nella classe Neuron
// - - - - - - - - - - - - - - - - - - - - - - - - -
Neuron::Neuron(WeightList* pwl, WeightList* swl) {
  _sum = 0;
  _out = 0;
  changed = TRUE;
  entering_links = pwl;
  exiting_links = swl;
}

double Neuron::value () {
  if (is_changed()) {
    _sum = 0;
    for (WeightList* p = entering_links; p != NULL; p = p->next()) {
      _sum += p->weight()->value();
    }
    _out = sigmoid(_sum);
    set_changed(FALSE);
  }
  return _out;
}

double Neuron::delta() {
  if (is_changed()) value();  /* ricalcola il valore se è stato cambiato */
  double _delta = 0;

  /* somma tutti i contributi sul delta */
  for (WeightList* p = exiting_links; p != NULL; p = p->next()) {
    _delta += p->weight()->get() * p->weight()->to()->delta();
  }
  _delta *= dsigmoid(_sum);   /* usa la derivata della sigmoide */
  return _delta;
}

double Neuron::delta(double derr) {
  if (is_changed()) value();
  return derr * dsigmoid(_sum);
}

void Neuron::set_changed(int x) {
  changed = x;
  if (changed) {
    for (WeightList* p = exiting_links; p != NULL; p = p->next()) {
      p->weight()->to()->set_changed(TRUE);
    }
  }
}
```

aggiungere o eliminare i nodi in maniera semplice ed efficiente.

All'interno del grafo tutti i nodi sono istanze della classe *Neuron* e i loro collegamenti sono modellati mediante la classe *Weight*, che rappresenta appunto un arco orientato etichettato. Un sottoinsieme di questi nodi viene designato come output della rete e rappresentato sempre con la classe Neuron; un altro viene invece designato come input e rappresentato con la classe figlia *InputTerminal*. Il calcolo dei delta è differenziato in base al tipo di nodo (se di input o interno).

Per semplicità si omette l'implementazione dell'algoritmo di back-propagation anche se, utilizzando le strutture e le funzionalità presentate, tale attività è piuttosto semplice.

Infine, si potrebbe realizzare una funzione di risoluzione chiamata dall'algoritmo precedente per modificare la topologia della rete in base alle considerazioni fatte in questo paragrafo (eliminazione di nodi, inserimento di nuovi nodi).

CONCLUSIONI

Anche se l'osservazione di John Denker (secondo la quale: *"...le reti neurali sono il secondo miglior modo per fare bene praticamente qualsiasi cosa…"*) può sembrare esagerata, si è riscontrato che con le reti neurali si raggiungono prestazioni accettabili per molti compiti che sarebbero difficili da risolvere esplicitamente servendosi di altre tecniche di programmazione. In particolare le reti neurali sono estremamente utili quando:

- Non possiamo formulare una soluzione algoritmica.
- Disponiamo di un gran numero di esempi del comportamento che vogliamo simulare.
- Vogliamo estrapolare la struttura da dati esistenti.

La valutazione delle potenzialità delle reti neurali, ottenuta mediante l'analisi di caratteristiche quali espressività, efficienza computazionale, sensibilità al rumore, ecc. suggerisce che rappresentazioni di questo tipo possono non solo fare tutto ciò che può essere fatto utilizzando un approccio computazionale tradizionale, ma anche eseguire compiti che, con un approccio tradizionale, sarebbero particolarmente difficili da realizzare.

BIBLIOGRAFIA

[1] Stuart J. Russell, Peter Norvig "*Artificial Intelligence, a modern approach*", Prentice-Hall International Inc., 1996.

[2] Tom M. Mitchell " *Machine Learning*", McGraw Hill, 1997.