UNIVERSITA' DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Esercitazioni per l'esame di
"Intelligenza Artificiale"
Prof. G. Soda

Neural Networks

di Sauro Menchetti

A.A.1998-99

Introduzione

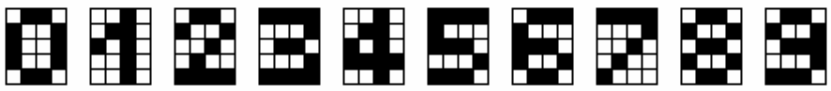
Il software bp è un simulatore di reti neurali che implementa l'algoritmo di propagazione all'indietro (backpropagation) per l'addestramento della rete. Una rete neurale rappresenta una funzione parametrizzata da un insieme di pesi ed è una rappresentazione particolarmente utile in quelle situazioni in cui si ha a che fare con funzioni che devono gestire ingressi affetti da rumore, situazioni in cui le tecniche basate sulla logica formale si trovano talvolta in difficoltà. Lo scopo dell'addestramento è quello di aggiustare i parametri della rete in base ad un insieme di esempi in modo che la funzione rappresentata si avvicini sempre più a quella che risolve correttamente il problema.

1.1 Descrizione del problema

Il problema trattato riguarda il riconoscimento delle dieci cifre numeriche: la rete prende come input una qualsiasi cifra disturbata o meno da rumore e cerca di fornire come output la cifra corrispondente. Un'apposita funzione si occupa di prendere in ingresso una cifra priva di disturbo e di fornire in uscita la stessa cifra alterata dal rumore. Si tratta quindi di un problema di classificazione dell'input in 10 classi distinte. L'obiettivo dell'addestramento della rete è quello di riuscire a classificare correttamente anche le cifre rumorose raggiungendo una certa capacità di generalizzazione, in modo da ricondurre un qualsiasi ingresso in una delle 10 classi previste che più gli assomiglia.

1.1.1 Codifica delle cifre

Ogni cifra viene codificata in una matrice di zero e di uno di dimensioni cinque per quattro. Ecco le dieci cifre non alterate dal rumore:



L'input è quindi costituito da un vettore di 20 elementi binari che linearizza la matrice cinque per quattro della cifra. Gli input riguardanti una singola cifra vengono forniti contemporaneamente agli ingressi della rete neurale. La funzione generatrice di rumore può aggiungere, togliere o spostare un pixel; non vengono prese in considerazione rotazioni o traslazioni dell'immagine.

1.2 Scelta dell'architettura della rete neurale

La scelta della corretta architettura di rete adatta a risolvere un dato compito è un problema di ricerca nello spazio delle possibili architetture (il problema dell'apprendimento è invece una ricerca nello spazio dei pesi): una rete troppo piccola può essere incapace di risolvere un certo problema mentre una rete con troppi parametri può creare fenomeni di overfitting, con un forte apprendimento dell'insieme degli esempi ed incapacità di generalizzare. Poiché si tratta di un problema di riconoscimento di caratteri, l'architettura della rete ricadrà tra gli autoassociatori. Un autoassociatore è una rete feedforward con uguale numero di ingressi e di uscite e con uno strato nascosto composto da un numero di neuroni inferiore. L'autoassociatore scelto ha 20 ingressi e 20 uscite. Il numero di ingressi e di uscite è quindi uguale al numero di elementi della matrice del carattere. La scelta del numero di neuroni nascosti è invece più critica. Poiché un autoassociatore effettua una compressione dell'informazione, 3 o 4 neuroni nascosti potrebbero essere sufficienti per il nostro problema in quanto per codificare 10 classi bastano 4 neuroni. Scegliere l'architettura più piccola che riesce ad apprendere i dati può però portare problemi di sensitività ai parametri iniziali: occorrono quindi dei test per individuare il numero di neuroni nascosti della rete sufficiente a risolvere il problema.

1.3 Il problema della generalizzazione a nuovi esempi

Quando un sistema viene addestrato per mezzo di esempi, una questione di importanza fondamentale è quella della sua capacità di riconoscere e classificare correttamente pattern che non appartengono all'ambiente di apprendimento. Se il sistema ha semplicemente acquisito l'attitudine a memorizzare gli esempi noti, avrà buone prestazioni in fase di apprendimento ma fallirà di fronte ad esempi nuovi che pure si discostano di poco da quelli conosciuti. Per ottenere un'elevata capacità di generalizzazione, deve essere immagazzinata nella rete quanta più conoscenza a priori possibile sul problema, limitando contemporaneamente il numero degli elementi costitutivi dell'architettura (neuroni e connessioni). Indipendentemente dall'algoritmo utilizzato per costruire e addestrare la rete neurale, l'apprendimento ottimo e la generalizzazione restano traguardi difficili da raggiungere contemporaneamente: si può dire che l'uno è inversamente proporzionale all'altro, tanto che la scelta di una rete che realizzi un compromesso soddisfacente fra caratteristiche così contrastanti rimane sempre molto difficile.

"apprendimento ottimo" $\propto$ *1 /* *"generalizzazione"* $\Rightarrow$ *compromesso*

Un approccio per evitare il fenomeno dell'overfitting dei dati è quello di valutare la capacità di generalizzazione della rete durante l'addestramento e di interrompere il processo quando si verifica una tendenza al ribasso: si suddivide approssimativamente in due parti uguali l'insieme di esempi a disposizione dette rispettivamente training set e test set e se ne utilizza una metà per l'addestramento e l'altra per la fase di verifica. L'addestramento termina quando l'errore di validazione comincia a crescere.

Il software bp

Il software bp è un simulatore di reti neurali che implementa l'algoritmo di propagazione all'indietro (backpropagation) per l'addestramento della rete. Utilizza vari file da cui legge le informazioni necessarie e stampa sullo schermo gli output relativi alle diverse elaborazioni. Sono di seguito descritti i vari file necessari a risolvere il problema.

2.1 Network file

Il network file descrive la struttura della rete ed ha generalmente estensione *.net*. Nel file devono essere indicati il numero di neuroni della rete compresi quelli fittizi di ingresso, divisi per strati e ordinati. Sui pesi possono essere definite delle limitazioni. Il file si divide in varie sezioni ognuna delle quali termina con il separatore *end*. Ecco il file per un autoassociatore con 20 ingressi, 20 uscite ed 8 neuroni nascosti:

```
definitions:
nunits   48
ninputs  20
noutputs 20
end
network:
%r 20  8  0 20
%r 28 20 20  8
end
biases:
%r 20 28
end
```

La prima sezione descrive una rete con 48 unità di cui 20 sono ingressi e 20 sono uscite. La sezione riguardante i vincoli sui pesi non è presente. La successiva sezione descrive le connessioni della rete. Indica che dal neurone numero 20 della rete per 8 neuroni, si hanno delle connessioni dal neurone 0 per 20 neuroni e tali connessioni sono inizializzate in modo random; inoltre dal neurone 28 per 20 neuroni, si hanno delle connessioni dal neurone 20 per 8 neuroni anch'esse inizializzate in modo random. L'ultima sezione pone delle soglie (bias) dal neurone 20 per 28 neuroni inizializzate in modo random.

2.2 Weights file

Il weights file contiene i valori dei pesi della rete ed ha generalmente estensione *.wts*. Può essere letto o scritto direttamente dal programma bp tramite i comandi *get / weights* e *save / weigths* dopo che il file *.net* è stato caricato per costruire l'architettura. Consiste in una lista di pesi seguita da una lista di soglie. La lista di pesi è formata da una serie di righe, una per ogni unità con connessioni convergenti. Le righe sono ordinate in base al numero dell'unità e consistono in una serie di valori ordinati secondo il numero dell'unità a cui sono collegati. Il comando *save / weights* salva un solo peso in ogni riga. I pesi non contengono nessuna indicazione relativamente al link al quale si applicano, né è segnato il confine fra pesi effettivi e bias, perché bp è in grado di trarre queste informazioni dal file di descrizione della rete. Bias inesistenti (come quelli dei neuroni in ingresso) sono posti a 0.

2.3 Template file

Il template file stabilisce il layout di visualizzazione dei risultati ed ha generalmente estensione *.tem*. È suddiviso in due sezioni: una prima opzionale in cui si cui si descrive il layout dello schermo e che termina con il separatore *end* ed una seconda in cui si danno le specifiche dei vari template[1]. Di seguito è riportato tale file:

```
define: layout
                                patno  ipatterns   tpatterns
epoch $    tss  $                $     $           $
           gcor $                      $           $
pname $    pss  $                      $           $
                                       $           $
                                       $           $
hidden   $

output1  $
target1  $

output2  $
target2  $

end

epochno    variable  1   $  0  epochno        4    1.0
tss        floatvar  1   $  1  tss            6    1.0
gcor       floatvar  2   $  5  gcor           6    1.0
cpname     variable  2   $  8  cpname        -4    1.0
pss        floatvar  2   $  9  pss            6    1.0

env.patno  variable  3   $  2  patno          3    1.0
env.ipat   vector    3   $  3  activation  h  2    1.0   0   4
env.ipat   vector    3   $  6  activation  h  2    1.0   4   4
env.ipat   vector    3   $ 10  activation  h  2    1.0   8   4
env.ipat   vector    3   $ 12  activation  h  2    1.0  12   4
env.ipat   vector    3   $ 14  activation  h  2    1.0  16   4
```

[1] Un template è un qualunque oggetto disegnabile sullo schermo.

```
env.tpat   vector    3   $  4   target      h  2     1.0   0   4
env.tpat   vector    3   $  7   target      h  2     1.0   4   4
env.tpat   vector    3   $ 11   target      h  2     1.0   8   4
env.tpat   vector    3   $ 13   target      h  2     1.0  12   4
env.tpat   vector    3   $ 15   target      h  2     1.0  16   4

hidden     vector    3   $ 16   activation  h  5  1000.0  20   3
output1    vector    3   $ 17   activation  h  5  1000.0  23  10
target1    vector    3   $ 18   target      h  5  1000.0   0  10
output2    vector    3   $ 19   activation  h  5  1000.0  33  10
target2    vector    3   $ 20   target      h  5  1000.0  10  10
```

Ogni simbolo `$` nella parte dedicata al layout indica l'angolo in alto a sinistra da cui verrà stampato il template. Le stringhe che compaiono nel layout vengono stampate così come compaiono. Il file template produce un layout di uscita in cui si mostrano il numero di epoche, gli errori quadratici relativi al pattern corrente (l'ultimo di ogni epoca) e totale, la quantità *gcor*, il nome e il numero del pattern corrente insieme al pattern stesso ed al target atteso. Nella parte bassa dello schermo si mostrano le uscite dei neuroni nascosti, dei neuroni di output insieme ai target attesi, suddivise in varie righe. Nelle sezione successiva dedicata alle specifiche, ogni template è descritto con tutti i suoi vari attributi. Consideriamo il seguente template:

```
hidden     vector    3   $ 16   activation  h  5  1000.0  20   3
```

La prima stringa indica il nome del template che identifica l'oggetto visivo, segue poi il tipo: in questo caso abbiamo a che fare con un vettore. Il numero `3` rappresenta il valore del *dlevel* per questo template. La coppia `$ 16` indica che il template deve essere stampato a partire dal sedicesimo `$` del layout. `activation` è il nome della variabile alla quale si riferisce il template così come è nota al programma. `h` indica l'orientamento del vettore (`h` per orizzontale e `v` per verticale). Il numero `5` fissa il numero di spazi per la stampa del template, mentre il valore `1000.0` è il fattore di scala. Gli ultimi due valori specificano di stampare 3 elementi del vettore a partire dal ventesimo.

2.4 Pattern file

Il pattern file memorizza le coppie di esempio per la rete ed ha generalmente estensione *.pat*. Ogni coppia consiste di un nome, di *ninputs* ingressi che specificano il pattern in ingresso e di *noutput* uscite che indicano il target atteso. I valori numerici sono separati da spazi e non ci sono separatori speciali tra i pattern ed i target. Ecco un esempio di pattern file per un autoassociatore con 4 ingressi e 4 uscite:

```
one    1 0 0 0   1 0 0 0
two    0 1 0 0   0 1 0 0
three  0 0 1 0   0 0 1 0
four   0 0 0 1   0 0 0 1
```

Il pattern file per il nostro problema è troppo complesso ed è quindi generato in modo automatico da un programma.

2.4.1 Il generatore del pattern file

Il programma genera un file di testo con tutte le informazioni utili sull'insieme di addestramento. Si utilizzano i seguenti tipi di dato definiti dall'utente:

```
typedef int Numero[NRIG][NCOL];

typedef struct coppia
{
     char nome[10];
     Numero pattern;
     Numero target;
} Coppia;

typedef Coppia Allenamento[NUMCOPPIE];
```

Il tipo `Numero` rappresenta la matrice della cifra, la struttura `Coppia` contiene il nome del pattern in ingresso, il pattern e il target di esempio e il vettore `Allenamento` memorizza l'intero ambiente di addestramento. La procedura fondamentale del programma è quella che aggiunge del disturbo ad un carattere che ne è privo. Opera in tre diversi modi: può togliere, aggiungere o spostare un pixel alla cifra. Ecco la sua dichiarazione:

```
void rumore(const Numero originale, Numero disturbato);
```

Un'altra procedura si occupa poi di creare l'insieme di allenamento ed ha la seguente dichiarazione:

```
void creaAllenamento(Allenamento all);
```

Le prime dieci posizioni del vettore `all` sono occupate dalle coppie prive di disturbo, mentre le rimanenti dagli esempi rumorosi. Infine un'apposita procedura si occupa di generare il pattern file, scrivendo su disco il vettore precedentemente creato.

2.5 Start file

Lo start file fornisce le principali informazioni di configurazione al software bp ed ha generalmente estensione *.str*. Per utilizzare il package che implementa backpropagation, si deve lanciare il comando:

bp template.tem start.str

dove *start.str* rappresenta il file di start. I principali comandi contenuti nello start file sono i seguenti:

```
get network  network.net
get patterns pattern.pat
get weights  weights.wts

set mode lgrain pattern
set ecrit   0.4
set nepochs 500
set param lrate  0.5
set param wrange 1.0

set mode follow  1
set mode cascade 0
set dlevel 3
set slevel 2
set lflag  1

set single 1
set stepsize epoch

disp opt standout 1

log logfile.log
```

I primi tre comandi si occupano di caricare l'architettura della rete, l'insieme iniziale di pesi e l'ambiente di addestramento. Si fissa a 500 il numero di epoche dell'addestramento on-line, con un learning rate uguale a 0.5; l'addestramento si ferma se l'errore quadratico relativo ad un'epoca è inferiore a 0.4. I pesi sono inizializzati in modo casuale con valori compresi tra –0.5 e 0.5. Si richiede il calcolo della correlazione del gradiente e si indica di fermarsi dopo ogni epoca. I valori negativi vengono stampati sullo schermo senza il segno meno in reverse video. Viene creato anche un logfile in cui vengono memorizzati certi template ed infine si inizializzano alcune variabili che tengono conto dell'aggiornamento dello schermo.

Test

I test svolti per riuscire ad ottenere un'architettura di rete addestrata in grado di classificare correttamente i vari input, riguardano la compressione dell'informazione, l'apprendimento delle dieci cifre in presenza ed in assenza di rumore e la capacità di generalizzazione a cifre rumorose non note. Il criterio seguito è di partire da un'architettura minimale e di incrementarla se necessario.

3.1 Le variabili *pss* e *tss*

Per capire quant'è l'errore che si sta commettendo nell'addestrare la rete, sono utili le due quantità *pss* e *tss* che rappresentano rispettivamente la somma dei quadrati degli errori ottenuti su ciascun neurone di uscita per un singolo pattern e per tutti i pattern appartenenti ad un'epoca:

$$pss = \tfrac{1}{2} \sum_{j(L)=1}^{n(L)} \left[x_{j(L)}(t) - d_{j(L)}(t) \right]^2$$

$$tss = \tfrac{1}{2} \sum_{t=1}^{T} \sum_{j(L)=1}^{n(L)} \left[x_{j(L)}(t) - d_{j(L)}(t) \right]^2$$

Nel nostro caso il numero di neuroni di output *n(L)* è 20. Supponendo di stabilire una soglia di uscita uguale a 0.5 per cui tutte le uscite inferiori a 0.5 sono assimilate a 0 mentre quelle superiori a 1, vogliamo calcolare quei valori di *pss* per cui certamente ci saranno o meno degli errori nell'apprendimento di un pattern. Il caso più sfortunato che può capitare è quello in cui 19 uscite sono proprio uguali a quelle del target atteso mentre la ventesima vale proprio 0.5. In questo caso si ha che:

$$pss = \tfrac{1}{2}(0.5)^2 = 0.125$$

Il caso più fortunato invece è quello in cui tutte le uscite sono uguali a 0.5. Si ha che:

$$pss = \tfrac{1}{2}(0.5)^2 * 20 = 2.5$$

Per quanto riguarda l'errore su un'intera epoca *tss*, si possono fare solo delle osservazione valide in media e non in assoluto supponendo che la distribuzione degli errori sui vari pattern sia di tipo uniforme. Detto *T* il numero dei pattern in un'epoca, si può affermare che:

- se *pss* ≤ 0.125, allora tutti i pixel del pattern sono stati appresi correttamente;
- se *pss* ≥ 2.5, allora almeno un pixel del pattern è sbagliato;
- se *tss* ≤ 0.125 * *T*, allora *probabilmente* tutti i pattern sono corretti;
- se *tss* ≥ 2.5 * *T*, allora *sicuramente* almeno un pattern non è corretto.

3.2 Compressione dell'informazione

Poiché un autoassociatore fa compressione dell'informazione, vediamo quale codice binario viene associato a ciascuna cifra numerica. Dato che le cifre sono 10, ci vogliono almeno 4 bit per codificarle: usiamo quindi 4 neuroni interni per la rete e vediamo quali sono le uscite di tali unità nascoste dopo che la rete è stata addestrata. Di seguito si riporta il risultato ottenuto, illustrando anche se la rete ha imparato correttamente la cifra e riportando il *pss* corrispondente:

cifra	0	1	2	3	4	5	6	7	8	9
codice	1110	0100	1001	0011	0000	0011	0010	0001	1010	1011
appresa	si	si	si	no	si	no	si	no	si	si
pss	0.099	0.099	0.099	2.280	0.103	1.171	0.098	0.477	0.099	0.099

Il valore finale di *tss* dopo 1500 epoche è di poco superiore a 4.5. Si può osservare che la rete ha troppi pochi parametri per riuscire nel suo compito: le cifre 3 e 5 che non sono correttamente apprese, presentano anche lo stesso codice. Anche addestrando la rete con dei nuovi pesi iniziali, i risultati variano di poco: occorre aumentare il numero dei neuroni nascosti per dare più gradi di libertà al sistema.

3.3 Apprendimento senza rumore

Si cerca adesso di far apprendere alla rete l'insieme delle 10 cifre non disturbate da rumore. Certamente 4 neuroni nascosti non sono sufficienti ad apprendere l'insieme di allenamento: vari test effettuati con 5 neuroni hidden forniscono più o meno gli stessi risultati, con le cifre 3, 5 e 7 che non sono classificate correttamente. Effettuando delle prove con 6 neuroni nascosti, si giunge alla corretta classificazione di tutte le cifre dopo circa 50 epoche. Riportiamo di seguito i dati relativi ad un allenamento di 200 epoche:

cifra	0	1	2	3	4
codice	000101	101100	000000	010010	100100
appresa	si	si	si	si	si
pss	0.034	0.036	0.086	0.036	0.031

cifra	5	6	7	8	9
codice	011100	011101	110000	010111	010001
appresa	si	si	si	si	si
pss	0.026	0.028	0.033	0.024	0.053

Questa volta anche il codice associato alle cifre non è ambiguo.

3.4 Apprendimento con rumore

Vediamo ora di far apprendere alla rete delle cifre rumorose senza preoccuparsi del problema della generalizzazione. Oltre ai 10 pattern non rumorosi, vengono presentati alla rete 100 pattern disturbati da rumore generati con l'apposita funzione che aggiunge, toglie o sposta un pixel. I 100 pattern rumorosi sono ripartiti uniformemente tra le varie cifre.

3.4.1 Rete con 6 neuroni nascosti

I test effettuati su una rete con 6 neuroni hidden, mostrano che tale rete è in grado di apprendere correttamente tutti i pattern ma solo in corrispondenza di certe condizioni iniziali. Si può dunque affermare che tale rete è la più piccola in grado di risolvere tale compito e quindi mostra una certa dipendenza dai parametri iniziali. Per giungere ad un corretto addestramento della rete, occorre innanzitutto eseguire un addestramento on-line della rete per circa 50 epoche fino a che l'errore totale *tss* non è vicino a 10. Un addestramento batch sulle epoche iniziali non consegue l'effetto voluto in quanto l'errore accumulato su un'intera epoca è troppo grande e l'algoritmo di addestramento non riesce a ripartirlo correttamente: l'errore oscilla tra dei valori che sembrano casuali senza mai scendere. Dopo essere giunti ad un errore accettabile con un addestramento on-line, si può applicare un addestramento batch che porta velocemente la rete nella sua condizione finale. Spesso occorre variare il learning rate per uscire da alcuni minimi locali ma nonostante tutto non sempre si arriva ad un corretto apprendimento di tutti i pattern. Quando un pixel non è correttamente appreso, quasi nella totalità dei casi si registra un valore opposto rispetto a quello atteso (questo porta ad un errore superiore ad 1 su quel pattern) e di solito è un uno che viene scambiato per uno zero e non viceversa. Due situazioni molto comuni che si verificano sono le seguenti: se alla fine dell'addestramento si ottiene un valore di *tss* di circa 2.5, molto spesso l'unico pattern sbagliato è il numero 29 che rappresenta un 9 disturbato in cui manca il pixel nell'angolo in basso a sinistra; se invece si giunge ad un valore di *tss* di circa 13 o 14, certamente un pattern è stato appreso male e tutte le volte che compare, rumoroso o meno, incrementa di 1 l'errore totale: spesso è la cifra 3 che dà origine a questa situazione.

3.4.2 Rete con 7 neuroni nascosti

Dato che la rete con 6 neuroni interni è sensibile ai parametri iniziali, vediamo come si comporta una rete con 7 neuroni hidden. Si può subito osservare che scompare completamente la dipendenza dai parametri iniziali ed non occorre più variare neanche il learning rate: qualunque sia la configurazione iniziale, è garantito il completo apprendimento di tutti i pattern. Inoltre solo raramente occorre applicare in cascata un addestramento batch ad uno on-line. Il numero di epoche necessarie per giungere ad un configurazione stabile diminuisce di 4-5 volte, attestandosi a circa 50. Il valore di *tss* può essere reso arbitrariamente piccolo aumentando il numero di epoche.

3.5 Generalizzazione con rumore

Per ultima cosa, cerchiamo di costruire una rete in grado di comportarsi "bene" anche in presenza di ingressi non noti. Si utilizza la cross-validation per evitare il fenomeno dell'overfitting dei dati interrompendo l'apprendimento quando si verifica una tendenza al ribasso della capacità di generalizzazione. Il training set è composto dalle 10 cifre non rumorose e da 100 cifre disturbate da rumore, mentre il test set contiene 50 cifre rumorose. I pattern rumorosi sono ripartiti in modo uniforme tra le varie cifre.

3.5.1 Rete con 7 neuroni nascosti

Iniziamo vedendo come si comporta una rete con 7 neuroni nascosti. Riportiamo di seguito una tabella in cui si evidenziano i valori di *tss* sul training set e sul test set al variare del numero di epoche ottenuti con un addestramento on-line. Tali valori sono stati stampati su un file di tipo *.log* alternando un passo di addestramento ad uno di test:

epoca	0	1	2	3	4	5	6	7	8
training	557,508	251,451	131,851	97,759	69,122	61,058	89,189	45,254	49,058
test	253,412	79,905	44,451	42,512	40,224	17,329	29,498	19,832	21,400

9	10	11	12	13	14	15	16	17	18
27,521	39,370	38,699	21,970	51,094	20,377	10,116	32,402	21,506	13,867
20,722	25,083	42,519	16,082	33,633	19,627	18,860	19,239	23,171	19,564

19	20	21	22	23	24	25	26	27	28
25,861	16,166	16,916	4,444	4,340	4,069	3,555	7,051	3,931	3,873
22,169	23,855	15,881	16,100	15,714	16,303	19,683	7,711	7,894	8,252

29	30	31	32	33	34	35	36	37	38
3,818	3,757	3,628	1,989	6,312	4,078	3,682	3,577	3,553	3,534
9,197	11,582	16,367	20,083	16,244	5,325	5,291	5,315	5,326	5,325

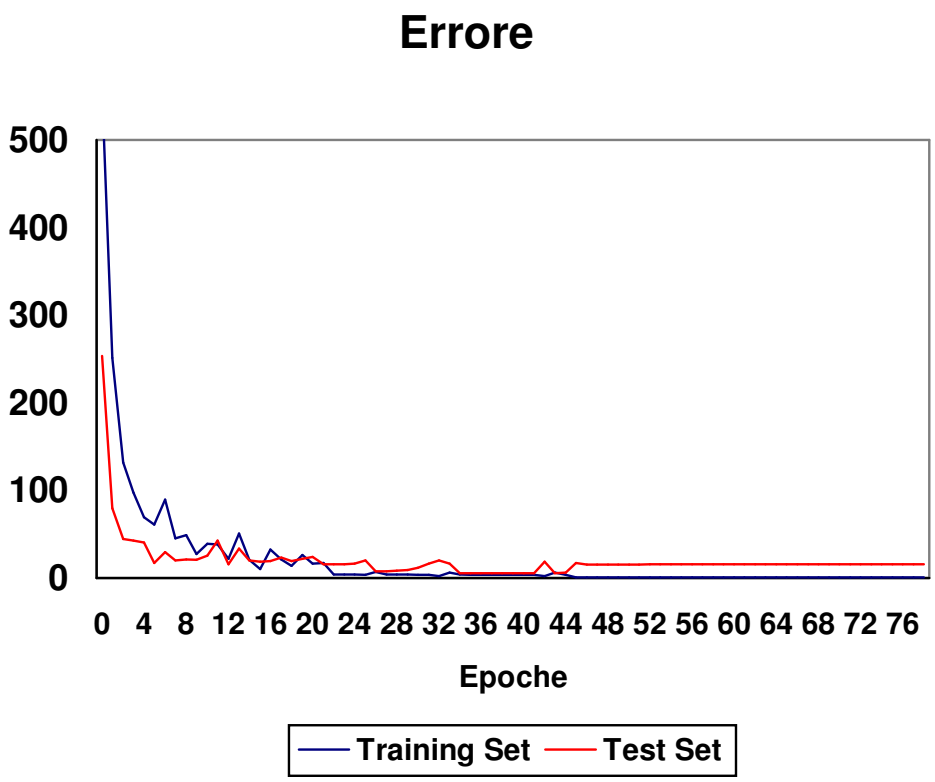
39	40	41	42	43	44	45	46	47	48
3,511	3,479	3,409	2,117	5,941	3,170	0,890	0,703	0,661	0,638
5,321	5,325	5,569	18,809	5,394	6,171	17,425	14,965	15,120	15,229

49	50	51	52	53	54	55	56	57	58
0,619	0,601	0,585	0,570	0,557	0,544	0,531	0,520	0,509	0,498
15,312	15,379	15,435	15,484	15,526	15,564	15,598	15,628	15,656	15,681

59	60	61	62	63	64	65	66	67	68
0,488	0,479	0,470	0,461	0,453	0,444	0,437	0,429	0,422	0,415
15,705	15,726	15,746	15,764	15,781	15,797	15,812	15,825	15,838	15,850

69	70	71	72	73	74	75	76	77	78
0,408	0,402	0,395	0,389	0,383	0,378	0,372	0,367	0,361	0,356
15,861	15,872	15,882	15,891	15,900	15,909	15,917	15,924	15,931	15,938

Ecco il grafico di tali valori:



Si può subito vedere che l'errore sul training set ha un andamento mediamente decrescente, mentre quello sul test set presenta dei minimi e da un certo punto in poi incomincia a crescere. Per quanto riguarda l'apprendimento dell'insieme di addestramento, si vede che il passaggio dall'epoca 44 alla 45 provoca il completo apprendimento di tutto l'insieme e l'errore da questo punto in poi tende in modo monotono a zero (nel grafico non compare più nessuna linea nera). L'errore sul test set raggiunge il suo minimo all'epoca 35 quando ancora non si ha un completo apprendimento del training set (solo il pattern 106, che rappresenta un 6 con un pixel aggiunto a destra che lo fa sembrare un 8, non è stato appreso correttamente) e tende poi asintoticamente a crescere. L'unica cifra del test set che non è stata appresa all'epoca 35 è la 112 che rappresenta un 2 rumoroso in cui il pixel in basso a destra è stato spostato di una casella verso l'alto. All'epoca 50 tutti i 110 pattern di allenamento sono appresi correttamente, mentre 5 pattern rumorosi del test set sono sbagliati. Andando avanti con le epoche, si registra una certa costanza dei valori che tendono a modificarsi molto lentamente (questo conferma la fine dell'apprendimento).

3.5.2 Rete con 8 neuroni nascosti

Vediamo quali risultati si ottengono aggiungendo un neurone nascosto ed effettuando un addestramento on-line:

epoca	0	1	2	3	4	5	6	7	8
training	556,797	248,783	117,939	85,187	87,768	34,031	34,199	20,540	33,999
test	252,910	47,974	50,455	56,976	27,210	26,518	25,700	11,146	21,304

9	10	11	12	13	14	15	16	17	18
21,093	31,638	36,938	24,091	10,533	25,278	44,524	33,870	43,972	53,213
25,156	16,613	22,224	24,106	17,966	24,004	26,305	24,234	26,288	41,498

19	20	21	22	23	24	25	26	27	28
30,157	77,324	63,307	35,742	19,593	20,101	26,128	10,650	6,949	2,009
33,339	34,574	23,534	15,929	18,058	24,326	18,737	12,373	11,644	15,359

29	30	31	32	33	34	35	36	37	38
1,321	1,166	1,095	1,038	0,990	0,947	0,908	0,873	0,841	0,812
14,210	14,273	14,283	14,277	14,264	14,247	14,230	14,213	14,196	14,179

39	40	41	42	43	44	45	46	47	48
0,785	0,760	0,737	0,715	0,694	0,675	0,657	0,640	0,624	0,608
14,164	14,149	14,136	14,123	14,111	14,099	14,089	14,078	14,069	14,060

49	50	51	52	53	54	55	56	57	58
0,594	0,580	0,567	0,554	0,542	0,530	0,519	0,509	0,499	0,489
14,051	14,043	14,035	14,027	14,020	14,013	14,006	13,999	13,992	13,986

59	60	61	62	63	64	65	66	67	68
0,480	0,471	0,462	0,454	0,446	0,438	0,431	0,424	0,417	0,410
13,979	13,973	13,967	13,961	13,955	13,949	13,943	13,937	13,932	13,926

69	70	71	72	73	74	75	76	77	78
0,404	0,397	0,391	0,385	0,380	0,374	0,369	0,363	0,358	0,353
13,920	13,914	13,909	13,903	13,898	13,892	13,887	13,881	13,876	13,871

L'errore sul training set e sul test hanno degli andamenti simili ed a differenza di prima, decrescono asintoticamente entrambi (praticamente raggiungono dei valori costanti). Il training set è correttamente appreso nella sua totalità dopo circa 30 epoche, mentre 5 elementi del test set non vengono classificati correttamente; aumentando il numero di epoche, si scopre che la rete ha raggiunto una configurazione stabile. Tale rete ha probabilmente troppi parametri, mentre quella con 7 neuroni nascosti è quella ottimale: infatti non si verifica neppure un aumento del valore dell'errore sul test set se si continua ad addestrare la rete, cosa invece molto comune se l'architettura di rete ha dimensioni adeguate.

3.6 Conclusioni

Il problema del riconoscimento delle 10 cifre numeriche è stato trattato usando un autoassociatore con 20 ingressi e 20 uscite. Ovviamente questo non è l'unico modo possibile, ma ad esempio poteva essere usata una rete con 10 uscite, una per ogni cifra. Come si è visto, la scelta critica riguarda il numero di neuroni nascosti dell'autoassociatore: se tale numero è superiore o inferiore al numero ottimale, allora nascono dei problemi durante l'apprendimento ed in fase di generalizzazione. A differenza di quanto potrebbe suggerire l'intuizione, 4 o 5 neuroni nascosti non sono sufficienti ad apprendere e codificare correttamente tutte e 10 le cifre. Occorrono 6 neuroni nascosti per imparare le 10 cifre numeriche e per ottenere un codice non ambiguo. Se si hanno anche dei pattern rumorosi, un autoassociatore con 7 neuroni nascosti è l'architettura ideale per riuscire ad avere una buona capacità di apprendimento e di generalizzazione. Non si devono effettuare molte epoche di addestramento in quanto un valore troppo basso dell'errore sul training set porta ad un errore elevato sul test set. Infine, una rete con 8 neuroni nascosti ha troppi gradi di libertà e mostra un comportamento peggiore di quella con 7.

```
/*****************************************************************************

                        GENERATORE DI CARATTERI RUMOROSI

*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <time.h>

/********************************** COSTANTI **********************************/

const int      NRIG =  5;  // numero di righe del carattere
const int      NCOL =  4;  // numero di colonne del carattere
const int NUMCOPPIE = 50;  // numero di coppie patter-target generate

const  char  *const  NOMEFILE  =  "ricnum.pat";     //  file  degli  esempi  per
l'addestramento

/************************** DEFINIZIONI DEI TIPI ******************************/

typedef int Numero[NRIG][NCOL];  // matrice del carattere

typedef struct coppia            // coppia di esempio con nome
{
       char nome[10];
       Numero pattern;
       Numero target;
} Coppia;

typedef Coppia Allenamento[NUMCOPPIE]; // vettore delle coppie di esempio

/******************** NUMERI DA ZERO A NOVE SENZA DISTURBO ********************/

const Numero numeri[10] = {

       {       {0, 1, 1, 0},   /* 0 */
               {1, 0, 0, 1},
               {1, 0, 0, 1},
               {1, 0, 0, 1},
               {0, 1, 1, 0}   },

       {       {0, 0, 1, 0},   /* 1 */
               {0, 1, 1, 0},
               {1, 0, 1, 0},
               {0, 0, 1, 0},
               {0, 0, 1, 0}   },

       {       {0, 1, 1, 0},   /* 2 */
               {1, 0, 0, 1},
               {0, 0, 1, 0},
               {0, 1, 0, 0},
               {1, 1, 1, 1}   },

       {       {1, 1, 1, 1},   /* 3 */
               {0, 0, 0, 1},
               {0, 1, 1, 0},
               {0, 0, 0, 1},
               {1, 1, 1, 1}   },

       {       {0, 0, 1, 0},   /* 4 */
               {0, 1, 1, 0},
               {1, 0, 1, 0},
               {1, 1, 1, 1},
               {0, 0, 1, 0}   },

       {       {1, 1, 1, 1},   /* 5 */
               {1, 0, 0, 0},
               {1, 1, 1, 0},
               {0, 0, 0, 1},
               {1, 1, 1, 0}   },
```

```
        {       {0, 1, 1, 1},   /* 6 */
                {1, 0, 0, 0},
                {1, 1, 1, 0},
                {1, 0, 0, 1},
                {0, 1, 1, 0}   },

        {       {1, 1, 1, 1},   /* 7 */
                {0, 0, 0, 1},
                {0, 0, 1, 0},
                {0, 1, 0, 0},
                {1, 0, 0, 0}   },

        {       {0, 1, 1, 0},   /* 8 */
                {1, 0, 0, 1},
                {0, 1, 1, 0},
                {1, 0, 0, 1},
                {0, 1, 1, 0}   },

        {       {0, 1, 1, 0},   /* 9 */
                {1, 0, 0, 1},
                {0, 1, 1, 1},
                {0, 0, 0, 1},
                {1, 1, 1, 0}   },
};

/********************** STAMPA DEI NUMERI DA ZERO A NOVE ************************/

void stampaNumeri(void)
{
        for (int i = 0; i < 10; ++i)
        {
                for (int n = 0; n < NRIG; ++n)
                {
                        for (int m = 0; m < NCOL; ++m)
                                printf("%d ", numeri[i][n][m]);
                        printf("\n");
                }
                printf("\n");
                getch();
        }
}

/********************** COPIA DI UN NUMERO IN UN ALTRO ************************/

void copiaNumero(numero destinazione, const Numero origine)
{
        for (int i = 0; i < NRIG; ++i)
                for (int j = 0; j < NCOL; ++j)
                        destinazione[i][j] = origine[i][j];
}

/********************** AGGIUNTA DEL RUMORE AD UN CARATTERE ************************/

/*
        ci sono tre modi per aggiungere del rumore ad un carattere:
        1) spostare una casella nera da un posto in un altro
        2) aggiungere una casella nera
        3) eliminare una casella nera
*/

void rumore(const Numero originale, Numero disturbato)
{
        copiaNumero(disturbato, originale);

        time_t t;
        srand((unsigned) time(&t));

        int rig;
        int col;

        int r = rand() % 3;

        if (r == 0) // se r è 0, si sposta una casella nera
        {
                do
                {
                        rig = rand() % NRIG;
```

```
                        col = rand() % NCOL;
                }
                while (originale[rig][col] == 0);

                disturbato[rig][col] = 0; // la casella nera diventa bianca

                do
                {
                        rig = rand() % NRIG;
                        col = rand() % NCOL;
                }
                while (originale[rig][col] == 1);

                disturbato[rig][col] = 1; // la casella bianca diventa nera
        }

        if (r == 1) // se r è 1, si aggiunge una casella nera
        {
                do
                {
                        rig = rand() % NRIG;
                        col = rand() % NCOL;
                }
                while (originale[rig][col] == 1);

                disturbato[rig][col] = 1; // la casella bianca diventa nera
        }

        if (r == 2) // se r è 2, si toglie una casella nera
        {
                do
                {
                        rig = rand() % NRIG;
                        col = rand() % NCOL;
                }
                while (originale[rig][col] == 0);

                disturbato[rig][col] = 0; // la casella nera diventa bianca
        }
}

/******************** CREAZIONE DELL'INSIEME DI ALLENAMENTO **********************/

void creaAllenamento(Allenamento all)
{
        for (int i = 0; i < 10; ++i) // le prime dieci coppie sono senza rumore
        {
                char nome[10];
                sprintf(nome, "p%d", i);
                strcpy(all[i].nome, nome);
                copiaNumero(all[i].pattern, numeri[i]);
                copiaNumero(all[i].target , numeri[i]);
        }

        for (i = 10; i < NUMCOPPIE; ++i) // seguono poi le coppie con pattern rumo-
rosi
        {
                char nome[10];
                sprintf(nome, "p%d", i);
                strcpy(all[i].nome, nome);
                Numero disturbato;
                rumore(numeri[i % 10], disturbato);
                copiaNumero(all[i].pattern, disturbato);
                copiaNumero(all[i].target , numeri[i % 10]);
        }
}

/********************** STAMPA DELL'INSIEME DI ALLENAMENTO ************************/

void stampaAllenamento(Allenamento all)
{
        for (int i = 0; i < NUMCOPPIE; ++i)
        {
                printf("%d) %s\n", i + 1, all[i].nome); // stampa il nome
                for (int n = 0; n < NRIG; ++n)          // stampa il pattern
                {
                        for (int m = 0; m < NCOL; ++m)
                                printf("%d ", all[i].pattern[n][m]);
```

```
                        printf("\n");
                }
                printf("\n");
                for (n = 0; n < NRIG; ++n)  // stampa il target
                {
                        for (int m = 0; m < NCOL; ++m)
                                printf("%d ", all[i].target[n][m]);
                        printf("\n");
                }
                getch();
        }
}

/******************* CREAZIONE DEL FILE PER L'ALLENAMENTO *********************/

void salvaAllenamento(const Allenamento all)
{
        FILE *fp;

        if ((fp = fopen(NOMEFILE, "w")) == NULL)
        {
                fprintf(stderr, "\n Non posso scrivere su %s", NOMEFILE);
                exit(1);
        }
        else
        {
                for (int i = 0; i < NUMCOPPIE; ++i)
                {
                        fprintf (fp, "%s  ", all[i].nome);  // salva il nome
                        for (int n = 0; n < NRIG; ++n)      // salva il pattern
                        {
                                for (int m = 0; m < NCOL; ++m)
                                        fprintf (fp, "%d ", all[i].pattern[n][m]);
                                fprintf(fp, " ");
                        }
                        fprintf(fp, "  ");
                        for (n = 0; n < NRIG; ++n)  // salva il target
                        {
                                for (int m = 0; m < NCOL; ++m)
                                        fprintf (fp, "%d ", all[i].target[n][m]);
                                fprintf(fp, " ");
                        }
                        fprintf(fp, "\n");
                }

                if (ferror(fp))
                {
                        fprintf(stderr, "\n Errore di scrittura");
                        fprintf(stderr, " su %s", NOMEFILE);
                        exit(2);
                }
                fclose(fp);
        }
}

/*************************** PROGRAMMA PRINCIPALE ****************************/

void main(void)
{
        clrscr();
        stampaNumeri();
        Allenamento allena;
        creaAllenamento(allena);
        stampaAllenamento(allena);
        salvaAllenamento(allena);
}

/************************  FINE PROGRAMMA PRINCIPALE ***************************/
```