

Francesco Sisini

# Introduzione alle reti neurali

con esempi in linguaggio C

Terza edizione

Edizioni: i Sisini Pazzi, 2020

Proprietà intellettuale di Francesco Sisini, 2018-20

## **Ringraziamenti per la prima edizione**

Il primo ringraziamento va ad Anna, Valentina e Laura che mi hanno sostenuto durante la preparazione di questo testo. Ognuna a modo proprio ha contribuito in modo importante, senza di loro non ce l'avrei fatta.

Un altro doveroso ringraziamento va alla mia famiglia di origine che nei primi anni '80, quando si faceva fatica ad arrivare alla fine del mese, acquistò per me e mio fratello, prima un TI994A, poi uno ZX Spectrum. Queste due macchine furono il mio battesimo del byte e mi aprirono nuovi orizzonti.

Veniamo adesso alla mia prima guida nel modo dell'informatica: Roberto Melis. Mi ha iniziato ai primi concetti del BASIC e della programmazione, io avevo dodici anni, lui sedici, ha avuto pazienza e lo ringrazio davvero. Grazie Robi! Poi sono venuti il mio tutor di dottorato l'impareggiabile Giovanni Didomenico e l'insostituibile Alberto Gianoli, da loro ho appreso molto e ringraziarli qui mi fa un immenso piacere.

## **Ringraziamenti per la seconda edizione**

Vorrei ringraziare le centinaia di lettori che hanno acquistato la prima edizione del libro e, fra complimenti e critiche, mi hanno dato la motivazione per completare anche questa seconda edizione.

## **Ringraziamenti per la terza edizione**

Vorrei ringraziare le centinaia di lettori che hanno acquistato la seconda edizione del libro e vorrei ringraziare Valentina Sisini che mi ha assistito nello sviluppo della libreria ReLe, ragionata in modo che potesse essere adatta a chi si affaccia per la prima volta al mondo delle reti neurali in C.

## Informazioni sulla proprietà intellettuale e la licenza d'uso

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Tutto il codice sorgente presentato in questo testo è opera di Francesco Sisini ed è usabile secondo i termini della licenza GPL v3 che riporto qui sotto.

Listati x.y

Copyright (C) 2018-20 Francesco Sisini

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <https://www.gnu.org/licenses/>.

## Significato dei font (note tipografiche)

- Il testo principale è scritto usando il font seguente: Ciao!
- La prima volta che viene introdotto nel testo un termini non comune, viene scritto in italico: *percettrone*. Le volte successive viene scritto usando il font usato per il testo principale.
- I termini gergali sono scritti tra doppi apici: l'array viene "steso".
- Le espressioni possono essere riportate in linea nel testo come segue: La funzione  $f$  è definita come:  $f(x) = x^2$
- ...oppure fuori linea come segue:

$$\begin{cases} f(x) = x^2 \\ y = f(x) \end{cases}$$

- Le equazioni che vengono citate nel testo sono identificate da un numero che permette di citarle successivamente:

$$\sigma(x) = \tanh(x) \quad (3.2)$$

Il numero dell'equazione è un progressivo che ha come prefisso il numero del capitolo. La citazione avviene indicando il numero dell'equazione messo tra parentesi: Come descritto in (3.2) la funzione...

- Le tabelle sono numerate con i numeri romani: Come mostrato in tabella IX....
- Le figure hanno una descrizione in italico ed un numero composto come quello delle equazioni ma con una numerazione propria: Come si vede in figura 3.1, il neurone...
- Il codice presentato nella sezione 2 è riportato come segue: `int i=1;` mentre i lunghi listati della sezione 3 sono riportati per motivi di leggibilità con un font più piccolo `int i=1;.` Si ricorda che dopo l'acquisto si potrà disporre del formato vettoriale dell'ebook richiedendolo sul gruppo facebook.

## *Sommario*

- Prefazione
- Parte prima: fondamenti di algebra
  - Operazioni con espressioni letterali p.18
  - Matrici p.29
  - Equazioni e sistemi lineari p.40
  - Coordinate e vettori p.45
  - Prodotto scalare p.49
  - Funzioni e derivate p.51
- Parte seconda: fondamenti del linguaggio C
  - Architettura del Personal Computer p.56
  - Cenni sul linguaggio macchina e sul linguaggio Assembly p.94
  - Il compilatore C p.120
  - Il flusso delle istruzioni p.123
  - Compilazione p.126
  - La memoria e le variabili p.129
  - Strutture iterative e di controllo nel linguaggio C p.139
  - Astrazione: le funzioni p.155
- Parte terza: fondamenti di reti neurali
  - Il modello matematico p.162
  - Il modello in C p.166
  - Rete neurale ispirata dalla biologia p.184
  - Programma in C della regola di Hebb p.195
  - Struttura del percettrone e implementazione in C p.205
  - Una rete di percetroni, gli strati *deep* p.221
  - Un esempio in C di MLP: riconoscere le cifre digitali a sette segmenti p.235
  - Riconoscimento di scrittura a mano p.252
  - Rete neurale a tre strati p.277
  - La libreria ReLe p.288

- Il cognitron e il neocognitron di Fukushima p.317
  - Conclusioni: e da qui? Come andare avanti?
- 

- Riconoscimenti
- Bibliografia essenziale

## Prefazione

L'intelligenza artificiale è un tema interessante e attuale che sta incuriosendo molti appassionati. Tra questi alcuni sono già esperti informatici, altri biologi, altri ancora matematici, ma tantissimi sono semplicemente appassionati e curiosi che non hanno ancora compiuto il percorso di studi necessario per comprendere davvero questa problematica.

In questo testo ho organizzato un percorso logico, chiaro e continuo che partendo dalle basi dell'algebra arriva alle basi della formulazione informatica delle reti neurali artificiali. La mia idea è che applicandosi con continuità, il lettore appassionato, ma completamente privo di basi sia informatiche che matematiche, arriverà a "mettere le mani" sulle reti neurali nel giro di un paio di mesi, mentre chi già un po' ne "masticava", accorcerà questo periodo in base alla propria esperienza e alla velocità di lettura. In tutti i casi raccomando di non saltare nessuna parte del libro perché questo non è un manuale tecnico, ma un testo didattico che sviluppa un discorso completo costruendo ogni paragrafo in conseguenza di quelli precedenti.

Per motivare il lettore e anche per fissare le idee con continuità, ho introdotto diversi termini e concetti propri delle reti neurali già dai primi paragrafi del libro, per cui anche se foste matematici provetti, non saltate le pagine senza di averle lette!

Il testo propone idee, modelli matematici e codici completi in linguaggio C. Ho volutamente inserito il codice sorgente degli esempi, per rendere questo testo un'opera autoconsistente senza bisogno di risorse esterne. Va da sé, che per mettere in pratica i codici, avrete bisogno di un elaboratore elettronico e un compilatore C. Consiglio un personal computer con una installazione di Linux con compilatore GCC.

Una domanda naturale che ci si potrebbe porre è: *Perché un libro sulle reti neurali in C invece che in Python?* Il linguaggio Python è diventato rapidamente celebre negli ultimi anni da quando il concetto

di *Big data* ha preso il sopravvento su quello di *Distribute computing* che aveva visto il linguaggio Java come protagonista. Python non è un linguaggio di programmazione per computer, è un linguaggio che serve a programmare un automa software, in pratica una macchina virtuale molto distante dalle attuali architetture hardware disponibili sul mercato. Si badi bene che questa non è una critica, ma solo una osservazione sul linguaggio che d'altra parte si rivela molto utile ed efficiente nei compiti legati al Machine Learning. Per Java vale la stessa considerazione.

Il linguaggio macchina e l'assembly invece sono in corrispondenza uno a uno con l'architettura del computer. Questa caratteristica permette all'assembly di sfruttare al meglio le caratteristiche hardware di un computer e questo è il motivo per cui l'assembly è stato il linguaggio di riferimento nei primi anni '80 quando sul mercato entrarono i personal computer, ognuno dei quali portava con sé una propria architettura con caratteristiche uniche sviluppate per offrire soluzioni ottimizzate a richieste specifiche. Quando l'architettura hardware dei computer si omogeneizzò attorno ai PC IBM compatibili, l'assembly offriva un livello di dettaglio ormai inutile, perché ogni macchina aveva la stessa architettura. In questo scenario si impose il linguaggio C, che offriva una corrispondenza logica uno a uno con il l'architettura di von Neumann adottata dai PC, ma permetteva al programmatore di astrarre diversi aspetti della programmazione che erano inutilmente dettagliati nell'assembly.

Dopo questa lunga prefazione torniamo al punto: perché il linguaggio C? La risposta è semplice: lo scopo principale di questo libro è di mostrare come sia possibile implementare una rete neurale artificiale su un computer.

Come ci si renderà conto nel proseguo della lettura, ma anche leggendo altri manuali sul tema, la programmazione di una rete neurale non presenta nessuna difficoltà specifica, il fatto di saper scrivere un codice che implementa gli algoritmi di *feed forward* e *back propagation* non ci rende esperti né di AI né di reti neurali.

Il valore che cerca di trasmettere questo testo consiste nelle idee profonde legate al concetto di *memoria associativa* e di *apprendimento supervisionato* e di come queste possano essere presenti in una *rete di cellule biologiche* come è appunto il nostro cervello. Il modo migliore per evidenziare questi concetti è quelli di presentarli con lo strumento paradossalmente più lontano che ci sia da essi, cioè un computer digitale che usa la memoria in maniera completamente diversa da come la usa il cervello.

Ho cercato di mantenere i codici abbastanza brevi in modo da poterli riportare nel testo per intero e tali da non scoraggiare il programmatore a scriverli con calma in un editor di testo, comunque, al tempo in cui scrivo, i codici sono disponibili sulla mia pagina di GitHub.

Sicuramente digitare tutto il codice potrebbe sembrare un onere pesante e un po' retrò, ma anche nei primissimi anni '80, per imparare a programmare si acquistavano le poche riviste specializzate e si copiava con calma tutto il codice dall'inizio alla fine per vederlo in esecuzione. Poi uscirono i codici già caricati su musicassetta.

Questo testo è costituito per intero da materiale originale concepito a questo preciso scopo. Non si tratta quindi né di materiale derivante da una raccolta di articoli né di copia incolla di altri testi, tanto meno delle dispense che ho preparato anni fa per i corsi di programmazione e linguaggi.

Raccomando un'ultima volta di leggere il libro per intero, ricordando che il suo valore è nell'opera didattica che va dall'algebra ai primi fondamenti di reti neurali, e non nella specifica implementazione del codice. Buona lettura.

# Introduzione alla memoria associativa e alla rappresentazione dell'informazione

Se vi chiedo di pensare ad una mela, siete capaci di immaginarla e di vederne la forma con "gli occhi della mente". Si potrebbe questionare sulla questione se l'immaginazione crei davvero immagini nel cervello paragonabili alle immagini che possiamo vedere nell'esperienza concreta, ma fatto sta che se vi chiedo di disegnare una mela anche senza averne un modello davanti ne siete capaci, quindi l'immagine della mela da qualche parte ce l'avete. Possiamo poi aggiungere che se siamo capaci di distinguere una mela da un pianoforte è sempre perché da qualche parte, presumibilmente nel cervello, c'è una serie di immagini di mele e di pianoforti che ci permettono di associare ad una specifica mela, la categoria astratta mela e quindi, a dispetto del terrorismo fatto dalla Walt Disney con Biancaneve, mangiarla. Lo stesso per i pianoforti.

Quindi, se accettiamo di avere memorizzato nel cervello delle immagini, è naturale chiedersi in che forma esse lo siano. Il recente modello digitale che abbiamo sviluppato negli ultimi sessant'anni, potrebbe portarci a pensare che esse siano memorizzate in formato *bitmap*, cioè come una matrice di punti colorati. Se così fosse vorrebbe dire che nel nostro cervello dovrebbero esistere delle unità di memorizzazione, simili ai *flip-flop* che sono elementi circuitali che vedremo nella seconda sezione del libro, con registrati i valori dei pixel di ogni immagine che ricordiamo.

Sempre secondo questo modello tali unità dovrebbero essere organizzate in righe e colonne, oppure in forma sequenziale, comunque con un preciso ordine, un elemento iniziale ed uno finale. Vedendola così, sarebbe naturale pensare che per richiamare alla mente una certa immagine sia necessario conoscere la sua ubicazione nella memoria, quindi conoscere il primo flip-flop da cui partire per accedere agli altri. In pratica dicendo la parola *mela* dovrebbe scattare un meccanismo di indirizzamento all'immagine

della mela che ci consente di visualizzarla nella mente. Un tale sistema potrebbe appoggiarsi ad una sorta di file-system della mente, una lista di parole chiave (keyword) con associato un indirizzo in memoria. Questo potrebbe anche essere possibile, nel cervello sono presenti delle cellule nervose dette neuroni che possono memorizzare, quindi l'idea non è del tutto balzana, però sembra non essere sufficiente a spiegare la capacità della mente di evocare delle immagini in condizioni diverse da quelle per ora analizzate. Supponiamo infatti che io vi mostri solo un centimetro quadrato di un' immagine di buccia di mela. Probabilmente, molti di voi sarebbero comunque in grado di riconoscere che si tratta di buccia di mela e poi di evocare l'immagine di una mela simile. Come è stato possibile? Se l'attivazione della memoria non è partita da una parola chiave, ma da una porzione di immagine, cosa avviene nel dettaglio?

Il riconoscimento di immagini, partendo da immagini parziali o addirittura trasfigurate, pone la questione della memorizzazione in modo nuovo a cui bisogna trovare modelli nuovi per dare delle risposte costruttive, cioè risposte che non soddisfino solo l'esigenza di avere una spiegazione ma che portino in sé il germe dello sviluppo, risposte capaci di replicare il meccanismo studiato, con mezzi artificiali. A dire il vero questo tema non è poi così recente, già Rosenblatt, in una ricerca che citerò nella terza sezione del libro, si pose questo problema e, gettando le basi per una risposta diversa alla problematica delle memorizzazioni, egli propose una visione che all'epoca era appena emergente:

**nel cervello potrebbero essere memorizzate le relazioni che le immagini hanno con gli stimoli che le generano anziché le immagini stesse**

Per farci un'idea intuitiva, immaginate che da nessuna parte nel cervello esista l'immagine della mela, ma esistano delle connessioni che si attivano se vi viene mostrato qualcosa di pertinente ad una mela e che tali connessioni portino alla formazione della immagine

della mela. Questa è l'idea alla base di ciò che si chiama *memoria associativa*.

Le conoscenze scientifiche di cui l'umanità è in possesso, ci spingono a vedere nell'intricata rete di connessioni delle cellule neurali che abbiamo nel cervello, l'"hardware" di base per il funzionamento della memoria associativa e quindi a fare dei neuroni, e delle reti di neuroni (neurali o neuronali che dir si voglia), un interessante oggetto di indagine.

Nella terza e ultima sezione del libro analizzeremo alcune idee e modelli di reti neurali artificiali, proponendone dei modelli matematici e il relativo codice in linguaggio C.

## Il modello neurale

Premetto che la mia esperienza con la biologia empirica è nulla, quindi nel raccontarvi cosa fa e a cosa serve un neurone non faccio altro che ripetervi cose che ho studiato e su cui mi sono confrontato con altri.

I neuroni sono le cellule che compongono il tessuto nervoso. Come le altre cellule specializzate hanno la caratteristica di compiere bene o male tutti la stessa routine quotidiana: ricevono dei segnali, li elaborano, li rispediscono. Noioso? La maggior parte dei lavoratori fa qualcosa del genere! In figura (a) è rappresentato un neurone in cui sono visibili i dendriti, il soma, il nucleo, l'assone e i terminali. Questi sono termini con cui dobbiamo prendere confidenza perché le reti neurali artificiali sono astrazioni di questi elementi costitutivi del neurone. Per parlare in termini semplici, pensiamo i neuroni come degli elementi che si scambiano dei segnali. Per cominciare, pensiamo al sistema che ci conduce dalla percezione di un oggetto nello spazio all'afferrarlo. L'immagine dell' oggetto giunge all'occhio come onde elettromagnetiche (luce visibile), queste illuminano la retina dove sono presenti delle cellule neurali che ricevono l'impulso luminoso. In biologia, queste cellule sono dette neuroni sensitivi o *afferenti*, e spesso ci riferiremo ad essi come ai neuroni di input.

L'energia luminosa viene trasdotta da detti neuroni in segnale elettrico e poi trasmessa lungo un canale chiamato *assone* che fa parte del neurone. Il segnale si dirama lungo i *terminali assonici* che sono poi collegati ai dendriti di altri neuroni attraverso dei collegamenti chiamati sinapsi. Il segnale "ottico" è quindi trasformato in un segnale nervoso (di tipo elettrico) e trasmesso ad altre cellule nervose, chiamate neuroni *intercalari* o *interneuroni*, che, tanto per far una anticipazione, sono i neuroni che danno ragione al termine *deep* del deep learning!

Qui i segnali vengono ricevuti da migliaia di neuroni attraverso migliaia di dendriti. Immaginiamo che ogni cellula neurale della retina possa essere collegata a migliaia di cellule nervose del cervello, quindi lo stesso stimolo ottico verrà elaborato da migliaia di cellule. Elaborato come? Diciamo intanto che non tutti i neuroni intercalari produrranno la stessa risposta allo stesso stimolo. Infatti la risposta di ognuno di essi dipende dalla frazione dello stimolo/segnales/impulso originale che raggiunge il *soma*, cioè il centro del neurone. Questa frazione dipende dalla connessione sinaptica tra i terminali dei neuroni afferenti e i dendriti dei neuroni intercalari. In base al livello di segnale ricevuto nel soma, ogni neurone potrà innescare (to fire) o meno una risposta/segnales che sarà poi trasmessa lungo il suo assone per raggiungere i terminali. I terminali sono connessi a dei neuroni detti *motori* o *efferenti*, per esempio le fasce muscolari del braccio, che consentono di afferrare l'oggetto in questione.

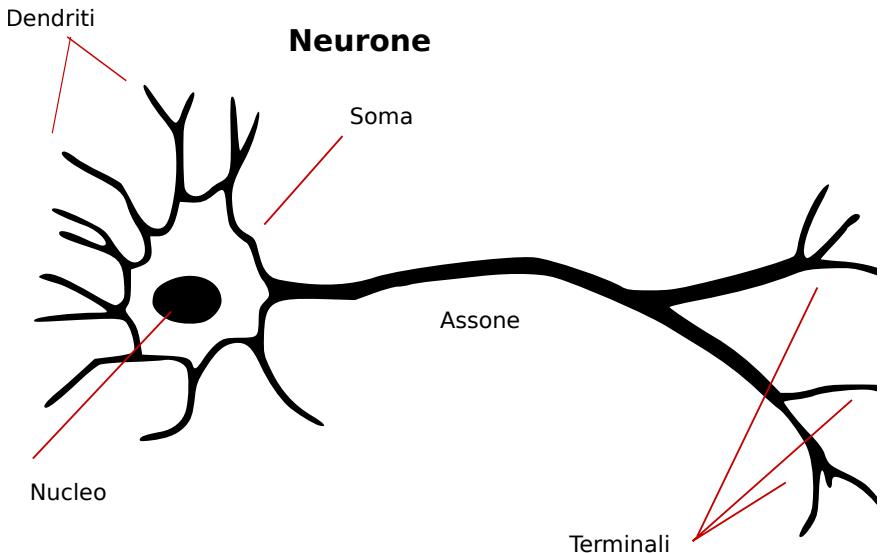


Fig. (a) - Schema di una cellula neurale (da Wikimedia)

## Come funziona l'apprendimento?

Le connessioni tra i neuroni non sono stabilite una volta per tutte, né in quantità né in qualità: esse possono cambiare sia in numero, cioè un neurone può aumentare il numero di neuroni a cui è connesso, sia in intensità, cioè il segnale tra il terminale e il dendrite viene trasmesso con minor smorzamento. Possiamo quindi dire che una rete di neuroni è *plastica* e questa plasticità è dovuta sia alla plasticità delle stesse connessioni sinaptiche che anche alla plasticità *strutturale* del neurone. La plasticità è anche il meccanismo che permette ad un cervello di recuperare alcune funzionalità dopo un trauma, ma in questo contesto, noi limitiamo la nostra analisi al ruolo della plasticità nell'apprendimento. Sia ben chiaro che quanto sto per esporre è scientificamente fondato, ma è comunque solo una interpretazione degli esperimenti che la scienza ha condotto fin'ora. Ricordiamo che non esiste verità scientifica, ma solo teorie non ancora falsificate!

Con questa premessa, posso dire senza troppa cautela che possiamo vedere il processo di apprendimento come un flusso di impulsi

nervosi che modella le strutture e le connessioni all'interno del cervello. Immaginiamoci nei panni di un bambino di pochi mesi che vuole afferrare per la prima volta un giocattolo. Il suo cervello non possiede ancora le *istruzioni* per permettergli di farlo in modo sicuro. Il bambino ha imparato che può muovere il braccio ma non ancora come afferrare un dato gioco, magari una palla. Per tentare un movimento il cervello dovrà attivare certi neuroni che andranno a loro volta ad attivare certe sinapsi. Il bambino tenta la presa, se la presa ha successo, le sinapsi attivate verranno rinforzate dal segnale di soddisfazione che verrà generato nel cervello del bambino, mentre al contrario se la presa avrà insuccesso, alcune sinapsi verranno inibite a causa del segnale di delusione. In pratica, possiamo vedere l'apprendimento in termini di connessioni tra neuroni. L'idea delle reti neurali artificiali è quindi quella di provare a ricostruire i complicati meccanismi che regolano il cervello e il sistema nervoso, replicandone la struttura mediante modelli fisici (macchine) o matematici (software). Più specifica e approfondita diventa la ricerca in questo campo, maggiore è la possibilità di migliorare i modelli. Per quel che riguarda questo testo, noi ci limiteremo all'introduzione della problematica, limitandoci a modellare le connessioni tra i neuroni, ma spero sia chiaro che la modellazione non si ferma a questo primo livello.

## **Stiamo per iniziare**

Tutte le cose scritte fin qui sul funzionamento del cervello sono molto interessanti, ma si possono leggere anche sfogliando delle riviste di divulgazione scientifica se non direttamente sul web, quindi perché comprare un libro sulle reti neurali per leggerle?

Il motivo è presto spiegato: studiando il contenuto di questo libro, prima la parte matematica, poi i fondamenti di architetture degli elaboratori, quindi l'assembly e il linguaggio C, darete un significato completamente nuovo alla parola capire, perché sarete in grado di scrivere da zero una rete neurale e spiegare davvero come funziona.

Quindi, **perché l'algebra?** Le connessioni tra i neuroni possono essere rappresentate mediante la matematica. Dai primi studi sulle reti neurali è stato comunemente accettato di usare le matrici per rappresentare dette connessioni, così le basi del calcolo matriciale sono fondamentali per poter leggere e comprendere sia i lavori pionieristici sulle reti neurali che gli studi più moderni. Nella sezione che segue queste basi saranno presentate al lettore in forma chiara e semplice, cercando di focalizzare la teoria ai soli concetti necessari per la messa in pratica.

# 1. Parte prima: fondamenti di algebra

In questa sezione vengono presentati i fondamenti dell'algebra che sono necessari per comprendere i modelli matematici usati per descrivere le reti neurali. L'obiettivo è quello di fornire al lettore digiuno di matematica degli strumenti per completare con profitto la lettura del libro. Per questo ho cercato di mantenere il livello di astrazione al minimo possibile prediligendo la semplicità espositiva alla perfetta coerenza e astrazione formale. Il lettore che, completata la lettura, aspiri a migliorare la propria concezione matematica, potrà farlo su testi specifici, come ad esempio quelli di Giuseppe Zwirner.

## 1.1 Operazioni con espressioni letterali

$$1 + 1$$

è una espressione algebrica come lo è anche:

$$3^2 + \frac{1}{2^3}$$

Le regole dell'algebra indicano come calcolare le espressioni algebriche in modo coerente. Per esempio se definiamo la quarta potenza del numero tre, come segue:

$$3^4 = 3 \times 3 \times 3 \times 3$$

dobbiamo accettare la proprietà per cui  $3^4 \times 3^3 = 3^7$  infatti si ha:

$$\begin{aligned} (3 \times 3 \times 3 \times 3) \times (3 \times 3 \times 3) &= \\ = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 &= 3^{(3+4)} = 3^7 \end{aligned}$$

Dimostrata questa proprietà, va da sé che essa vale anche per qualsiasi altro numero, infatti nella dimostrazione vista sopra non abbiamo usato nessuna caratteristica intrinseca del numero tre, anzi lo abbiamo trattato semplicemente come un simbolo e quindi possiamo sostituire a quel simbolo un altro simbolo, per esempio il simbolo 9 che rappresenta anche il numero nove. Con detta sostituzione si ha:

$$\begin{aligned}9^4 \times 9^3 &= (9 \times 9 \times 9 \times 9) \times (9 \times 9 \times 9) = \\&= 9 \times 9 \times 9 \times 9 \times 9 \times 9 \times 9 = 9^{(4+3)} = 9^7\end{aligned}$$

A questo punto è lecito domandarsi se la stessa proprietà valga anche per un numero composto da più cifre. Per rispondere a questo quesito, sfruttiamo quanto abbiamo già fatto finora. La proprietà dimostrata vale per qualsiasi simbolo, infatti non c'è nulla di differente tra esprimerla usando come simbolo il 3, o il 9, come abbiamo fatto ma anche una lettera, per esempio la lettera  $a$ . Possiamo quindi anche riscrivere la proprietà usando un simbolo letterale:

$$a^3 = a \times a \times a$$

e quindi anche:

$$\begin{aligned}a^3 \times a^4 &= (a \times a \times a \times a) \times (a \times a \times a) = \\&= a \times a \times a \times a \times a \times a \times a = a^{(3+4)} = a^7\end{aligned}$$

Bene, dimostrato che la proprietà può essere espressa per mezzo di un simbolo anziché per mezzo di un numero, possiamo fare un ragionamento, forse un po' grezzo ma efficace: possiamo pensare alla lettera  $a$  come ad una variabile, cioè un segnaposto a cui possiamo sostituire un numero, quello che vogliamo, per esempio il numero 97. Operando tale sostituzione si ha:

$$\begin{aligned}97^3 \times 97^4 &= (97 \times 97 \times 97 \times 97) \times (97 \times 97 \times 97) = \\&= 97 \times 97 \times 97 \times 97 \times 97 \times 97 \times 97 = 97^{(3+4)} = 97^7\end{aligned}$$

Quindi abbiamo visto sia un esempio di calcolo letterale, cioè di calcolo in cui trattiamo l'algebra applicata puramente ai simboli, che di calcolo con variabili, cioè calcolo in cui le lettere rappresentano dei segnaposto per i numeri. Abbiamo visto che possiamo dare un significato preciso ad ogni espressione del tipo  $3^5$ , ma anche del tipo  $a^4$ , allora possiamo anche dare un significato preciso alle espressioni del tipo  $(3^2)^3$  infatti, seguendo ricorsivamente la definizione vista sopra sia ha:

$$\begin{aligned}
 (3^2)^3 &= (3^2) \times (3^2) \times (3^2) = & (1.1) \\
 &= (3 \times 3) \times (3 \times 3) \times (3 \times 3) = 3^6 = 3^{(2 \times 3)}
 \end{aligned}$$

Vediamo ora cosa succede se dividiamo tra loro due potenze che abbiano la stessa base:

$$\frac{3^4}{3^2} = \frac{3 \times 3 \times 3 \times 3}{3 \times 3} = \frac{\cancel{3} \times \cancel{3} \times 3 \times 3}{\cancel{3} \times \cancel{3}} = 3^{(4-2)} = 3^2$$

poi, in particolare, sia ha:

$$\frac{3^4}{3^4} = 1 = 3^{4-4} = 3^0$$

e questa è la spiegazione del perché qualsiasi numero (tranne lo 0) elevato a 0 da come risultato 1, per cui si ha in generale che  $x^0 = 1$  per ogni  $x$  diverso da 0. Continuiamo con questa linea di ragionamento e spostiamo l'attenzione dalla base delle potenze con cui abbiamo operato, agli esponenti. Abbiamo dimostrato che  $3^3 \times 3^4 = 3^{(3+4)}$  però per dimostrare questa proprietà degli esponenti abbiamo fatto uso esplicito del significato aritmetico degli esponenti stessi, infatti abbiamo scritto  $3^3 = 3 \times 3 \times 3$  ripetendo il prodotto tante volte quanto il valore dell'esponente. Quello che vogliamo vedere adesso è che possiamo fare lo stesso con un esponente letterale, cioè possiamo scrivere:

$$3^b \times 3^c = 3^{(b+c)} \quad (1.2)$$

Il problema che si incontra cercando di dare una dimostrazione logica della definizione qui sopra è che non abbiamo una buona definizione della potenza ad esponente letterale, mentre è molto più semplice la definizione di una potenza con base letterale ma esponente numerico. Possiamo però lavorare il problema ai fianchi per abituarci, almeno in modo intuitivo, all'idea di usare potenze con esponente letterale. Vediamo anzitutto che anche se non sappiamo dare un significato aritmetico alla potenza  $3^b$  dobbiamo richiedere che valga comunque la seguente proprietà:

$$\frac{3^b}{3^b} = 1 \quad (1.3)$$

perché questo, ancor prima che un significato matematico, ha il significato datogli dal senso comune, che si può esprimere dicendo: «Una mela sta in una mela una volta!». Nello stesso modo, l'espressione  $3^b$  debba stare in se stessa una sola volta. Proviamo a dimostrare la (1.2) usando la (1.1) e la (1.3). Sebbene ancora non abbiamo dato un significato preciso alle espressioni con esponente letterale (i.e.  $5^b$ ), dalla (1.1) possiamo dire che:

$$(3^b)^2 = (3^b) \times (3^b) = 3^{2 \times b} \quad (1.4)$$

D'ora in poi, per comodità, il prodotto di un numero per una lettera lo indicheremo omettendo il segno " $\times$ ", quindi al posto di scrivere  $2 \times b$  scriveremo  $2b$ . Con questo possiamo riscrivere la (1.4) come  $(3^b)^2 = 3^{2b}$ . Con la (1.4) abbiamo quindi dimostrato che:

$$(3^b) \times (3^b) = 3^{(b+b)}$$

nello stesso modo possiamo scrivere che:

$$\frac{3^{2b}}{3^b} = \frac{3^b \times 3^b}{3^b} = \frac{\cancel{3^b} \times 3^b}{\cancel{3^b}} = 3^{(2b-b)}$$

Possiamo scrivere anche :

$$3^b = \frac{3^b \times 3^c}{3^c} = 3^{b+(c-c)}$$

da cui

$$3^b \times 3^c = 3^{b+c}$$

che è una specie di dimostrazione della (1.2), anche se come avvertito precedentemente è solo una dimostrazione intuitiva e non rigorosa. Lo scopo di questi primi conti è solo quello di abituarsi all'idea che è possibile esprimere le regole dell'algebra anche rispetto a dei simboli non numerici.

Ora che abbiamo preso confidenza con l'idea di scrivere una potenza che abbia una base o un esponente letterale, facciamo un passo ulteriore e proviamo a scrivere una potenza che abbia sia base che esponente letterale:

$$a^b \quad (1.5)$$

è un'espressione "ben formata" per i motivi visti sopra. Possiamo quindi combinare l'espressione (1.5) con altre espressioni algebriche, per esempio possiamo scrivere  $a^b + 1$  oppure possiamo scrivere  $a^b + a$ , entrambe sono espressioni ben formate, il fatto che nessuna di queste due espressioni permetta di giungere ad un risultato numerico non deve preoccupare.

Con questo piccolo bagaglio nozionistico che abbiamo preparato, possiamo prepararci ad un viaggio nelle possibilità espressive dell'algebra. Vediamo alcuni casi significativi. Cominciamo vedendo come si deve computare qualche espressione più complessa:

$$a(b + c) \quad (1.6)$$

Volendo computare la (1.6) il problema è che non sappiamo come calcolare la somma  $b + c$ , e questo è normale. Vogliamo però che la computazione della (1.6) dia un risultato che sia corretto per ogni simbolo, quindi anche eventualmente per dei numeri che possano essere sostituiti alle lettere  $a$ ,  $b$  o  $c$ , quindi nel prodotto (1.6) dobbiamo applicare la proprietà distributiva ottenendo:

$$a(b + c) = ab + ac \quad (1.7)$$

e la (1.7) è tutto quello che possiamo dire rispetto alla (1.6). Cosa succederebbe se nella (1.6) sostituissimo tre numeri a caso? Per esempio il 5 al posto della  $a$ , il 6 al posto della  $b$  e il sette al posto della  $c$ ? Si avrebbe:

$$5 \times (6 + 7) = 5 \times 13 = 65$$

che è lo stesso che otterremmo sostituendo le lettere con gli stessi numeri nella (1.7), infatti si ha:

$$ab + ac = 5 \times 6 + 5 \times 7 = 30 + 35 = 65$$

Si parla allora di calcolo letterale quando si applicano ad espressioni letterali le proprietà del calcolo dedotte dal calcolo numerico. Vediamo di seguito le principali.

## Monomi

Si chiama monomio un prodotto di un numero, detto coefficiente numerico per una parte letterale che può essere una sola lettera o il prodotto di più lettere, ognuna delle quali può essere elevata ad un esponente numerico e naturale. Quelli che seguono sono esempio di monomi:

$$3a, 5ab, b, 3a^2b^5$$

invece non sono monomi i seguenti:

$$3a + 1, a + b, 5b^{\sqrt{2}}$$

in particolare si ha che la prima e la seconda espressione sono binomi, mentre la terza ( $5b^{\sqrt{2}}$ ) è un'espressione algebrica valida ma non è un monomio, né un binomio, né più in generale un polinomio, e vediamo subito il perché: per i monomi vogliamo definire una proprietà che chiamiamo grado e corrisponde alla somma degli esponenti delle lettere che compaiono nella parte letterale. Se consideriamo nuovamente l'espressione  $5ab$  che equivale a scrivere  $5a^1b^1$  il grado del monomio è  $1+1=2$ . Il grado deve essere un numero naturale, pertanto non si nomina "monomio" una espressione con parte letterale avente un esponente non intero. Vediamo subito che la somma tra due monomi che hanno la stessa parte letterale (si dice che sono simili) equivale ad un monomio avente sempre la stessa parte letterale ma coefficiente numerico dato dalla somma dei coefficienti; vediamo un esempio:

$$4abc^3 + 3abc^3 = 7abc^3$$

Il prodotto tra due monomi è un monomio, che ha per coefficiente numerico il prodotto dei coefficienti numerici e per parte letterale il prodotto delle parti letterali, per esempio:

$$4abc^3 \times 3a^2b = 12a^3b^2c^3$$

Da notare che il grado del polinomio prodotto è 8 ed è uguale alla somma del grado dei due fattori, cioè 5 e 3. È inoltre possibile definire la potenza di un monomio come segue:

$$(3a^2b^3)^2 = 9a^4b^6$$

La divisione tra due monomi è definita, e se la parte letterale del denominatore si semplifica possiamo dire che il risultato è ancora un monomio:

$$\frac{4a^2b^3c}{3abc} = \frac{4}{3}ab^2$$

Voglio far notare che alcuni testi chiamano monomio anche una espressione letterale in cui una o più lettere abbiano un esponente negativo (i.e.  $3a^{-1}b$ ). Non c'è nulla di sbagliato in questo anche se poi bisogna porre molta attenzione agli enunciati dei teoremi riguardanti il grado dei monomi e come vedremo dei polinomi. Comunque questa questione è praticamente irrilevante ai fini di questo libro.

La somma di due monomi non simili come  $3ab + 5ac$  non è un monomio ma si chiama genericamente polinomio:

## Polinomi

Un polinomio è la somma di più monomi, per esempio:

$$3a^2b + 2ab + 5b$$

è un polinomio. Tra i polinomi distinguiamo i binomi, cioè particolari polinomi costituiti dalla somma di due soli monomi. Questi sono interessanti perché nell'algebra elementare per essi sono studiati alcuni prodotti notevoli, per esempio si ha:

$$\begin{aligned}(a + b)^2 &= (a + b) \times (a + b) = aa + ab + ba + bb = \\ &= a^2 + 2ab + b^2 = a^2 + b^2 + 2ab\end{aligned}$$

si chiama quadrato di un binomio e per esso ci si ricorda la ricetta: "Il

quadrato di un binomio è dato dal quadrato del primo più il quadrato del secondo più il doppio prodotto del primo per il secondo calcolati tenendo conto del segno".

Tenendo conto di quanto detto vediamo di applicarlo ad una somma leggermente più complessa come  $3ab - c$ , si ha subito:

$$(3ab - c)^2 = 9a^2b^2 + c^2 - 6abc$$

Vediamo come impraticchirsi con questi conti. Supponiamo di dover calcolare il quadrato di questo binomio:  $-\sqrt{2}a^3b + \sqrt{2}b^2$ . Anzi tutto procediamo seguendo la regola:

- il quadrato del primo:  $(-\sqrt{2}a^3b) = +2a^6b^2$
- il quadrato del secondo:  $(\sqrt{2}b^2)^2 = 2b^4$
- il doppio prodotto del primo per il secondo (con il loro segno):  
 $2 \times (-\sqrt{2}a^3b \times \sqrt{2}b^2) = -2(\sqrt{2})^2a^3b^3 = -4a^3b^3$       dove  
abbiamo usato  $(\sqrt{2})^2 = 2$

quindi:

$$(-\sqrt{2}a^3b + \sqrt{2}b^2)^2 = 2a^6b^2 + 2b^4 - 4a^3b^3 \quad (1.8)$$

Dopo aver eseguito questo conto è normale essere insicuri sul risultato e, per far bene, bisogna provare che è corretto con un esempio. Se l'espressione che abbiamo ottenuto è corretta allora, sostituendo due numeri qualsiasi ad  $a$  e  $b$  si dovrà ottenere una uguaglianza corretta. Scegliamo  $a = 2$  e  $b = 3$  e calcoliamo il quadrato dopo aver sostituito  $a$  e  $b$  nella (1.8) con i valori scelti:

$$\begin{aligned} (-\sqrt{2}a^3b + \sqrt{2}b^2)^2 &\rightarrow (-\sqrt{2} \times 2^3 \times 3 + \sqrt{2} \times 3^2)^2 = \\ &= (\sqrt{2} \times 3^2 - \sqrt{2} \times 2^3 \times 3)^2 = \\ &= [\sqrt{2}(3^2 - 2^3 \times 3)]^2 = (\sqrt{2} \times 18)^2 = 450 \end{aligned}$$

ora prendiamo l'espressione ottenuta in (1.8) e sostituiamo direttamente  $a$  e  $b$ :

$$2a^6b^2 + 2b^4 - 4a^3b^3 = 2 \times 2^6 \times 3^2 + 2 \times 3^4 - 4 \times 2^3 \times 3^2 = 450$$

Questo tipo di verifiche è fondamentale nello studio della matematica per prendere sicurezza e non spaventarsi di fronte alle forme astratte. Bisogna sempre ricordare che si sta trattando con cose concrete e che se non è possibile fare un esempio concreto di un teorema o di una proprietà allora è meglio non studiarlo neanche.

Un secondo prodotto notevole, di cui si fa poi spesso uso è il seguente:

$$(a + b)(a - b) = a^2 - b^2$$

Si provi per esercizio che  $(3ab - c)(3ab + c) = 9a^2b^2 - c^2$ .

## Regole dei segni

Prima di procedere con l'algebra delle matrici ricordiamo brevemente le regole dei segni nelle operazioni algebriche con una serie di esempi per il prodotto e la divisione:

$$(-1) \times 7 = -7$$

$$(-1) \times (-7) = 7$$

$$(-a) \times (-b) = ab$$

$$(-2) \times (a^2) = -2a^2$$

$$\frac{1}{-1} = -1$$

$$\frac{1}{2} \times (-1) = \frac{-1}{2} = -\frac{1}{2}$$

ricordo poi che per la somma si ha:

$$-1 - 1 = (-1) + (-1) = (-1) \times (1 + 1) = -2$$

$$-1 + 2 = 2 - 1 = 1$$

$$-7 + 3 = 3 - 7 = -4$$

$$-a - a = -2a$$

$$-a - b = -(a + b)$$

## 1.2 Matrici

Le matrici servono a descrivere le connessioni tra i neuroni nelle reti neurali artificiali sia software che hardware. I programmi che implementano degli algoritmi di machine learning basati sulle reti neurali operano principalmente con le matrici. Il professor Giuseppe Zwirner, nel suo testo "Lezioni di Analisi Matematica 1", introduce le matrici in modo molto diretto dicendo che sono tabelle di numeri, e questa è la definizione che io pure preferisco. Vediamo un esempio di matrice:

$$\begin{bmatrix} 1 & 2 & 1 \\ 0.1 & 1.1 & 2.8 \\ 1 & 0 & 1 \\ 2 & 21 & 12.9 \end{bmatrix} \quad (1.9)$$

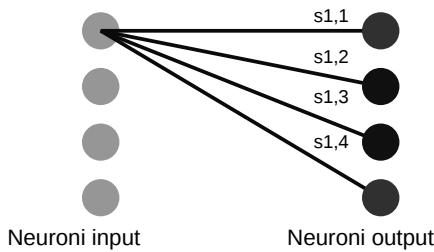
La matrice mostrata in (1.9) ha 4 righe e 3 colonne. Ogni matrice è identificata dalla coppia  $n \times m$  di righe e colonne, la (1.9) è una matrice  $4 \times 3$ . Alle matrici si assegna un nome, per esempio possiamo scrivere:

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 0.1 & 1.1 & 2.8 \\ 1 & 0 & 1 \\ 2 & 21 & 12.9 \end{bmatrix}$$

per indicare una generica matrice di ordine  $n \times m$  si scrive  $\mathbf{M}_{n,m}$ . Prima di esaminare le proprietà matematiche delle matrici e le regole per eseguire somme e prodotti tra esse, vediamo un esempio di come sono applicate alla teoria e al calcolo delle reti neurali artificiali. Tutti i concetti introdotti ora saranno ripresi a tempo debito, quindi prego il lettore di non preoccuparsi nel caso non capisca a pieno l'esempio che mi appresto a portare.

Consideriamo un insieme di 4 neuroni che rappresentano l'interfaccia di input di un certo sistema e un altro insieme di 4 neuroni che rappresentano l'interfaccia di output dello stesso sistema.

Supponiamo ora che esista una sinapsi tra ogni neurone di input ed ogni neurone di output, cioè che da ogni neurone appartenente all'insieme di input si possa raggiungere ognuno dei neuroni appartenenti all'insieme di output. Supponiamo di indicare con la lettera  $s$  la sinapsi tra due neuroni, allora la sinapsi tra il neurone 1 di input e quello 1 di output potrebbe essere indicata con il simbolo  $s_{1,1}$  e quella tra il neurone 1 di input e il neurone 2 di output con il simbolo  $s_{1,2}$  e via dicendo per il neurone 3 e il 4.



*Fig. 1.1 - Esempio di collegamento tra neuroni.*

Passando al secondo neurone dell'insieme di input, avremmo che il collegamento sinaptico tra esso ed il primo dell'insieme di output verrebbe indicato con  $s_{2,1}$  e così via. L'insieme dei simboli  $s_{i,j}$ , dove  $i$  e  $j$  sono due indici che in questo caso possono variare tra 1 e 4, conta  $4 \times 4 = 16$  elementi che possono essere disposti in una tabella usando la semplice regola di disporre il generico elemento  $s_{i,j}$  nella  $i$ -esima riga e  $j$ -esima colonna:

$s_{1,1}$	$s_{1,2}$	$s_{1,3}$	$s_{1,4}$
$s_{2,1}$	$s_{2,2}$	$s_{2,3}$	$s_{2,4}$
$s_{3,1}$	$s_{3,2}$	$s_{3,3}$	$s_{3,4}$
$s_{4,1}$	$s_{4,2}$	$s_{4,3}$	$s_{4,4}$

Dalla tabella possiamo costruire la matrice con un meccanismo molto semplice, in realtà si tratta solo di una costruzione mentale di un concetto. Il meccanismo è il seguente, diciamo che esiste la matrice  $\mathbf{S}$  i cui elementi sono le righe e le colonne della tabella vista sopra, quindi diciamo che la matrice  $\mathbf{S}$  ha elementi  $s_{i,j}$  e la rappresentiamo

come segue:

$$S = \begin{bmatrix} s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} \\ s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} \\ s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} \\ s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} \end{bmatrix}$$

Prima di passare alla somma e al prodotto tra matrici è meglio definire un lessico comune che ci sarà comodo nelle prossime pagine.

- Le righe e le colonne vengono anche dette *linee*. Con il termine *linee parallele* ci si riferisce a più righe o colonne considerate insieme.
- In due linee parallele (sia righe che colonne) si chiamano *corrispondenti* gli elementi che occupano lo stesso posto nella linea.
- Due linee si possono sommare o anche sottrarre, sommando o sottraendo da ogni elemento quello corrispondente.
- Due linee parallele si possono scambiare tra loro scambiando ogni elemento con quello corrispondente.
- Due linee sono uguali tra loro se ogni elemento dell'una è uguale al corrispondente nell'altra. Stesso dicasì per due linee proporzionali.
- Una matrice si dice nulla se tutti i suoi elementi sono nulli, cioè valgono 0.
- Presa una matrice, per esempio la matrice  $\mathbf{S}$  vista sopra, si può costruire un'altra matrice,  $\mathbf{S}^T$ , ottenuta dalla prima scambiando le righe con le colonne:

$$\mathbf{S} = \begin{bmatrix} s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} \\ s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} \\ s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} \\ s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} \end{bmatrix}, \quad (1.10)$$

$$\mathbf{S}^T = \begin{bmatrix} s_{1,1} & s_{2,1} & s_{3,1} & s_{4,1} \\ s_{1,2} & s_{2,2} & s_{3,2} & s_{4,2} \\ s_{1,3} & s_{2,3} & s_{3,3} & s_{4,3} \\ s_{1,4} & s_{2,4} & s_{3,4} & s_{4,4} \end{bmatrix}$$

La matrice ottenuta nella (1.10) si chiama *matrice trasposta* e si indica apponendo la lettera  $T$  al nome della matrice, per esempio la trasposta di  $\mathbf{S}$  è  $\mathbf{S}^T$

- Due matrici che abbiano lo stesso numero di righe e colonne si dicono *simili*. Tra due matrici simili, si dicono corrispondenti gli elementi che occupano lo stesso posto.
- Due matrici simili sono uguali se sono uguali tutti gli elementi corrispondenti.
- Una matrice si dice quadrata se il numero di righe è uguale al numero di colonne. Per una matrice quadrata si definisce la *diagonale principale* e la *diagonale secondaria*. La principale va da sinistra a destra e dall'alto in basso, quindi è costituita da tutti gli elementi che hanno indice di riga uguale a quello di colonna, per la matrice  $\mathbf{S}$  sono gli elementi  $s_{1,1}, s_{2,2}, s_{3,3}, s_{4,4}$ . La diagonale secondaria va da destra a sinistra dall'alto in basso, quindi:  $s_{1,n}, s_{2,n-1}, s_{3,n-2}, s_{4,n-3}$
- In una matrice quadrata, si chiamano elementi *coniugati* i due elementi  $s_{i,j}$  e  $s_{j,i}$  ottenuti l'uno dall'altro scambiando gli indici (i.e.  $s_{3,1}$  è coniugato a  $s_{1,3}$ ). Ovviamente, gli elementi coniugati sono simmetrici rispetto alla diagonale principale, mentre tutti gli elementi della diagonale principale sono coniugati a sé stessi.
- Sempre per una matrice quadrata, nel caso si abbia che valga per ogni elemento  $s_{i,j} = s_{j,i}$  allora la matrice si dice *simmetrica*. Se vale  $s_{i,j} = -s_{j,i}$  e si hanno tutti gli elementi della diagonale nulli, allora la matrice è *emi-simmetrica*.

## Somma e prodotto fra matrici

Nel paragrafo precedente abbiamo visto che ci sono quantità che possono essere rappresentate più facilmente attraverso una matrice che in altro modo. Per essere onesto devo dire che nella storia della matematica le matrici sono venute prima dello studio delle reti neurali artificiali, ma esistono anche altri sistemi che vengono modellati bene da una matrice, uno di questi è proprio lo studio di un sistema di equazioni, che come vedremo tra tre o quattro paragrafi, diventa algebricamente trattabile se si considerano i coefficienti delle incognite come delle matrici. Comunque il punto a cui voglio arrivare è che visto che le matrici sono oggetti matematici degni di interesse per esse, si sono sviluppate delle regole dell'algebra. La prima che vediamo è la somma tra due matrici. Ci tengo a sottolineare che la somma tra matrici è per così dire una invenzione, non mia, ma una invenzione. Comunque tale invenzione segue una regola piuttosto intuitiva. Immaginate di creare due piccoli porta oggetti costituiti da una griglia di legno a scacchiera. Ora riempite ognuna delle celle dei vostri porta oggetti con delle perline. Fatto? Adesso ponete i due porta oggetti (identici), l'uno di fianco all'altro e spostate le perline della prima cella in alto, del porta oggetti di sinistra, nella corrispondente celletta di quello di destra. Fatto? Bene, ripetete ora la procedura per tutte le cellette. Quante perline avete per ogni celletta nel porta oggetti di destra? Non mi pare che valga neanche la pena di dare la risposta, avete la somma delle quantità che erano presenti prima dell'operazione. In modo simile definiamo la somma tra due matrici **A** **B** sommando ad uno ad uno gli elementi corrispondenti. Va da sé che la somma ha senso solo se le matrici hanno lo stesso numero di righe e colonne. Siano **A** e **B** date da:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{bmatrix}$$

$$\mathbf{C} = \mathbf{A} + \mathbf{B} \rightarrow c_{i,j} = a_{i,j} + b_{i,j} \rightarrow$$

$$\rightarrow \mathbf{C} = \begin{bmatrix} b_{1,1} + a_{1,1} & b_{1,2} + a_{1,2} & b_{1,3} + a_{1,3} & b_{1,4} + a_{1,4} \\ b_{2,1} + a_{2,1} & b_{2,2} + a_{2,2} & b_{2,3} + a_{2,3} & b_{2,4} + a_{2,4} \\ b_{3,1} + a_{3,1} & b_{3,2} + a_{3,2} & b_{3,3} + a_{3,3} & b_{3,4} + a_{3,4} \\ b_{4,1} + a_{4,1} & b_{4,2} + a_{4,2} & b_{4,3} + a_{4,3} & b_{4,4} + a_{4,4} \end{bmatrix}$$

Prima di procedere, introduco il simbolo  $\sum$  che è una *sigma* maiuscola, quindi una "s" maiuscola dell'alfabeto greco. Con questo simbolo si indica una sommatoria rispetto ad un indice, per esempio  $i$ . Se voglio sommare i primi dieci numeri interi posso scrivere:

$$\sum_{i=1}^{10} i = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$$

Quindi con questa scrittura abbiamo indicato che vogliamo sommare per 10 volte l'espressione  $i$  al variare di  $i$  da 1 a 10. Proviamo a fare un secondo esempio, consideriamo la scrittura:

$$\sum_{i=0}^4 2^i = 2^0 + 2^1 + 2^2 + 2^3 + 2^4$$

dove abbiamo usato l'indice  $i$  per indicare l'esponente della base 2. L'indice può anche essere usato per far riferimento ad un elemento di una matrice, così ad esempio se vogliamo sommare tutti gli elementi della prima riga della matrice  $\mathbf{B}$ , possiamo scrivere:

$$\sum_{i=1}^4 a_{1,j} = a_{1,1} + a_{1,2} + a_{1,3} + a_{1,4}$$

Con questo nuovo simbolismo matematico, siamo in grado di continuare la presentazione delle matrici con maggior snellezza.

Definiamo quindi anche il prodotto fra matrici per il quale, a differenza della somma, è più difficile trovare una analogia con le mensoline porta oggetti, peccato. Il prodotto comporta operazioni simultanee sulle righe di una matrice e le colonne dell'altra.

Date le stesse **A** e **B** si definisce il prodotto come segue:

$$\mathbf{C} = \mathbf{AB} \rightarrow c_{i,j} = \sum_{k=1}^4 a_{i,k} b_{k,j}$$

ad esempio se consideriamo l'elemento  $c_{1,4}$  troviamo:

$$c_{1,4} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \end{bmatrix} \begin{bmatrix} b_{1,4} \\ b_{2,4} \\ b_{3,4} \\ b_{4,4} \end{bmatrix} = \\ = a_{1,1}b_{1,4} + a_{1,2}b_{2,4} + a_{1,3}b_{3,4} + a_{1,4}b_{4,4}$$

dove ho nascosto le righe e le colonne non interessate alla definizione dell'elemento  $c_{1,4}$ . Va da sé che, per fare un esempio, il calcolo dell'elemento  $c_{2,3}$  interesserà la riga 2 della matrice **A** e la colonna 3 della matrice **B**.

Il prodotto tra matrici non richiede che esse abbiano lo stesso numero di righe e colonne, ma solo che il numero di colonne dell'una sia uguale al numero di righe dell'altra. Mi spiego meglio, supponiamo che  $\mathbf{A} \in \mathbf{M}_{(m,n)}$  allora  $\mathbf{B} \in \mathbf{M}_{(n,o)}$  dove  $o$  e  $m$  possono essere numeri a piacere. Dove con il simbolo  $\mathbf{M}_{(m,n)}$  si indicano le matrici di  $m$  righe e  $n$  colonne. In pratica abbiamo detto che se **A** ha 3 righe e quattro colonne, la posso moltiplicare per **B** solo se **B** ha 4 righe.

Forse scordavo di dire una cosa molto importante che il numero di righe di una matrice potrebbe anche essere 1, in questo caso si parla di matrice riga, oppure una matrice potrebbe avere una sola colonna, e in questo caso si parla di matrice colonna. Data una matrice **A**  $n \times m$  possiamo riferirci alla sua riga  $i$ -esima come  $\mathbf{A}_i$  e alla sua colonna  $j$ -esima come  $\mathbf{A}^j$ . Una matrice può quindi essere vista come una matrice colonna i cui elementi sono matrici riga. Mi raccomando

di soffermarsi a ragionare su questo punto perché nello studio del C è fondamentale, così come lo è nella trattazione delle reti neurali ratificali. Infatti affrontando la teoria del *percettrone*, che è l'elemento minimale all'interno di una rete neurale artificiale, si vedrà che esso è caratterizzato da una matrice riga che descrive i pesi delle connessioni *sinaptiche* dei propri *dendriti* (parleremo indistintamente di pesi sinaptici o dendritici). Lo strato neurale formato dai percetroni è caratterizzato dalla matrice colonna di tutte le matrici riga dei percetroni, ed è la classica matrice **W** che si incontra ormai in qualsiasi trattato moderno sulle reti neurali (la 'W' sta per weight, cioè i pesi delle connessioni.)

Ora che abbiamo introdotto il prodotto fra matrici vediamo che esiste una matrice che gioca il ruolo dell'unità, cioè moltiplicata per un'altra matrice la lascia invariata, questa si chiama matrice identità ed è una matrice quadrata di ordine  $n$  che ha tutti gli elementi nulli tranne quelli sulla diagonale che valgono 1:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.11)$$

Nella (1.11) è stata definita la matrice identità per le matrici di ordine 4, ma può essere definita per matrici di qualsiasi ordine. Provate per esercitarvi, e vedrete che il prodotto di **I** per qualsiasi matrice **quadrata** dello stesso ordine da come risultato la matrice stessa. Ora che abbiamo definito l'elemento neutro della moltiplicazione tra matrici, viene voglia di sapere se, data una matrice esiste la sua inversa, cioè data **A**, ci chiediamo se esista **B** tale che **AB = I**. Se esiste abbiamo trovato quella che si chiama la matrice inversa di **A** e si indica con **A**<sup>-1</sup>. Proviamo a fare due conti, e per stare sul facile consideriamo che **A** ∈ **M**<sub>(2,2)</sub> e siano:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}$$

Se imponiamo  $\mathbf{AB} = \mathbf{I}$  si ha:

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (1.12)$$

Il calcolo delle (1.12) porta al sistema (provare per credere)

$$\begin{cases} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} = 1 \\ a_{2,1}b_{1,2} + a_{2,2}b_{2,2} = 1 \\ a_{1,1}b_{1,2} + a_{1,2}b_{2,2} = 0 \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} = 0 \end{cases} \quad (1.13)$$

Bene, se proviamo a risolvere le (1.13) per esempio rispetto a  $b_{1,1}$  troviamo:

$$b_{1,1} = \frac{a_{2,2}a_{1,1}}{a_{2,2}a_{1,1} - a_{1,2}a_{2,1}} \frac{1}{a_{1,1}} \quad (1.14)$$

La (1.14) può essere risolta solo se il termine  $a_{2,2}a_{1,1} - a_{1,2}a_{2,1}$  risulta diverso da 0. In generale questo non è un fatto scontato, per alcune matrici quel termine è diverso da 0, mentre per altre non lo è. Le prime si dicono *invertibili*, le seconde non invertibili. Detto termine determina quindi il carattere di invertibilità della matrice e per questo viene detto *determinante*. Un analogo ragionamento, anche se con calcoli più lunghi si applica anche a matrici  $3 \times 3$ ,  $4 \times 4$  ecc.

## Calcolo del determinante di una matrice

Abbiamo visto nel paragrafo precedente da dove nasce l'esigenza di calcolare il determinante di una matrice, e abbiamo anche visto che il calcolo per una matrice  $2 \times 2$  si riduce alla differenza del prodotto delle diagonali, la principale meno la secondaria. Vediamo ora come calcolare il determinante di una matrice di ordine 3 e vediamo che per le matrici di ordini maggiori, si potrà usare lo stesso metodo in

modo ricorsivo. Sia quindi  $\mathbf{A} \in \mathbf{M}_{(3,3)}$  indichiamo il determinante di  $\mathbf{A}$  come:

$$\det(\mathbf{A}) = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix}$$

e scriviamo che :

$$\begin{aligned} \det(\mathbf{A}) &= a_{1,1}(-1)^{1+1} \begin{vmatrix} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{vmatrix} + \quad (1.15) \\ &+ a_{1,2}(-1)^{1+2} \begin{vmatrix} a_{2,1} & a_{2,3} \\ a_{3,1} & a_{3,3} \end{vmatrix} + a_{1,3}(-1)^{1+3} \begin{vmatrix} a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{vmatrix} \end{aligned}$$

Le tre matrici:

$$\begin{bmatrix} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{bmatrix}, \quad \begin{bmatrix} a_{2,1} & a_{2,3} \\ a_{3,1} & a_{3,3} \end{bmatrix}, \quad \begin{bmatrix} a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{bmatrix}$$

si ottengono dalla matrice  $\mathbf{A}$  sopprimendo di volta in volta gli elementi che stanno sulla riga 1 e colonna 1, poi quelli sulla riga 1 e colonna 2 e infine quelli che stanno sulla riga 1 e colonna 3. In generale, data una matrice  $\mathbf{A} \in \mathbf{M}_{(n,n)}$ , preso un suo elemento, possiamo determinare una nuova matrice  $\mathbf{A}_{i,j} \in \mathbf{M}_{(n-1,n-1)}$  sopprimendo da  $\mathbf{A}$  gli elementi della riga i e colonna j. Con questa regola e le (1.15), possiamo definire in modo ricorsivo il determinante di una matrice  $\mathbf{A} \in \mathbf{M}_{(n,n)}$ :

$$\begin{aligned} \det(\mathbf{A}) &= a_{1,1}(-1)^{1+1} |\mathbf{A}_{1,1}| + a_{1,2}(-1)^{1+2} |\mathbf{A}_{1,2}| + \dots \quad (1.16) \\ &\dots + a_{1,n}(-1)^{1+n} |\mathbf{A}_{1,n}| \end{aligned}$$

Infatti, indipendentemente dalla dimensione  $n$ , possiamo applicare la (1.16) ad ogni matrice  $\mathbf{A}_{i,j}$  finché si riduce ad una matrice di ordine 2, per la quale sappiamo calcolare il determinante. Per completezza espositiva, ci tengo a dire che il termine  $A_{i,j} = a_{i,j}(-1)^{i+j} |\mathbf{A}_{i,j}|$  si chiama *complemento algebrico* o *aggiunto* dell'elemento  $a_{i,j}$ .

Sapere cosa è un determinante e saperlo eventualmente calcolare è

importante per muoversi con sicurezza nella teoria algebrica e nei libri che trattano le reti neurali in modo appropriato. Per questo invito i lettori a leggere con attenzione quanto qui esposto e a provare a fare qualche esercizio. Detto questo, nel proseguo non ci capiterà di calcolare dei determinanti, mentre ci capiterà di eseguire prodotti e somme tra matrici.

Per semplificare il calcolo del determinante ci sono diversi teoremi che risultano in regole di semplice applicazione. Consiglio caldamente di studiarle sullo Zwirner (vedi in bibliografia) che tratta le matrici e i determinanti in modo veramente dignitoso. Unica avvertenza per chi decidesse di approfondire lo studio su detto libro è di prestare attenzione alla scelta, per altro non errata, che lo Zwirner compie di definire il prodotto tra matrici come prodotto per righe anziché righe per colonne. In sé non ci sarebbe nulla di errato, ma l'intero mondo ormai intende il prodotto tra matrici come righe per colonne e non righe per righe. A causa di ciò anche l'algoritmo per il calcolo della matrice inversa proposta lì è da modificare.

### 1.3 Equazioni e sistemi lineari

Ora che abbiamo introdotto le matrici, possiamo vedere una loro importante applicazione ai meccanismi della memoria associativa delle reti neurali artificiali.

Consideriamo che ad un neurone *sensitivo*, come le cellule della retina che permettono di raccogliere gli stimoli luminosi, stiano arrivando più stimoli che indichiamo con le variabili  $x_1, x_2, \dots, x_n$ . L'intensità di ogni stimolo è indipendente dalle altre, per questo abbiamo usato  $n$  variabili indipendenti per rappresentarli. Supponiamo ora (cosa peraltro realistica) che il segnale inviato dal neurone al cervello sia la somma  $s$  dei segnali percepiti e scriviamo:  $s = x_1 + x_2 + \dots + x_n$ . Per esser ancora più realistici consideriamo che i segnali non vengano sommati insieme così come si presentano al neurone, ma, in base al percorso che hanno seguito per andare dalla cellula sensitiva al *soma* del neurone, essi siano più o meno smorzati. Per rappresentare questo comportamento introduciamo  $n$  costanti numeriche che saranno caratteristiche del neurone che stiamo considerando, e dette  $a_1, a_2, \dots, a_n$  le costanti, riscriviamo la somma  $s$  come segue:

$$s = a_1x_1 + a_2x_2 + \dots + a_nx_n \quad (1.17)$$

La somma (1.17) è detta anche essere una somma pesata. Notiamo anzi tutto che la (1.17) può essere scritta come il prodotto di una matrice riga **A** per una matrice colonna **X**:

$$s = (a_1, a_2, \dots, a_n) \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ \vdots \\ x_n \end{pmatrix}$$

Proviamo adesso a cambiare prospettiva: supponiamo di conoscere i pesi del neurone dati dalla matrice riga **A** e di conoscere la sua

attivazione data dalla somma pesata  $s$  e ci chiediamo se siamo in grado di risalire alla configurazione di segnali di *input* ricevuti dal neurone per produrre la detta attivazione. Tale quesito si trasforma in una equazione che possiamo scrivere invertendo i termini nella (1.17) come:

$$a_1x_1 + a_2x_2 + \dots + a_nx_n = s \quad (1.18)$$

e che interpretiamo nel modo seguente: si trovino i valori  $x_1, x_2, \dots, x_n$  tali che sostituiti nella (1.18) la soddisfino, cioè la somma a sinistra dia un valore uguale alla quantità a destra. Se provate a prendere qualche numero a caso, per esempio, stabilito che  $n = 3$ , per fare prima, scriviamo l'equazione (1.18) come:

$$5x_1 + 2x_2 + 2x_3 = 18$$

si trova immediatamente che la terna di valori  $(2, 3, 1)$  è una soluzione, ma è chiaro che ne esistono infinite altre, perché abbiamo una sola equazione e tre *gradi di libertà* o variabili indipendenti.

Molto bene, complichiamo un po' la cosa: supponiamo che lo stesso stimolo arrivi anche ad un secondo neurone che abbia i pesi dati dalla matrice riga **B**. Facciamo una cosa, rinominiamo le matrici e chiamiamo i pesi del primo neurone **A**<sub>1</sub> e quelli del secondo riga **A**<sub>2</sub>. Ora, la riga **A**<sub>2</sub> sono i pesi del secondo neurone ma l'input è lo stesso del primo. Supponiamo invece che la somma pesata dei segnali sia  $s_2$  (e intanto rinominiamo  $s$  in  $s_1$ ). Ci troviamo ad avere quindi una seconda equazione che accompagna la prima:

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,n}x_n = s_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \dots + a_{2,n}x_n = s_2 \end{cases} \quad (1.19)$$

Come si sarà intuito, le (1.19) costituiscono un sistema di equazioni *lineari*. Il termine lineare si riferisce al grado del polinomio rispetto alle variabili indipendenti  $x_i$  che appunto è 1. Ci interessa ora analizzare in quali condizioni le (1.19) sono risolvibili. Come sicuramente avrà già capito chi si interessa alle reti neurali artificiali, questo è un problema tutt'altro che accademico.

Cominciamo a studiare un sistema di equazioni lineari partendo dalla sua forma più semplice, cioè dal caso in cui la matrice  $\mathbf{A}$  data dalla matrice colonna delle matrici  $\mathbf{A}_i$  sia una matrice quadrata, quindi abbia tante righe quante colonne. Le (1.19) possono essere scritte in forma matriciale come segue:

$$\left\{ \begin{array}{l} a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,n}x_n = s_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \dots + a_{2,n}x_n = s_2 \\ \dots \\ a_{n,1}x_1 + a_{n,2}x_2 + \dots + a_{n,n}x_n = s_n \end{array} \right. \rightarrow \mathbf{AX} = \mathbf{S} \quad (1.20)$$

dove  $S$  è la matrice colonna delle somme  $s_1, s_2, \dots, s_n$ . La (1.20) ha soluzione immediata se il determinante di  $\mathbf{A}$  è diverso da 0, infatti, come visto nel paragrafo precedente, in questo caso possiamo calcolare la matrice inversa  $\mathbf{A}^{-1}$  e dimostrare che:

$$\mathbf{X} = \mathbf{A}^{-1}\mathbf{S}$$

È bene sapere che questo tipo di sistema, cioè con numero delle equazioni uguale al numero delle incognite, e determinante dei coefficienti diverso da 0, si chiama sistema di Cramer.

Completando i conti presentati in equazione (1.14) si ottiene una regola generale per risolvere i sistemi di Cramer. Consideriamo la matrice  $\mathbf{A}$  composta di matrici colonna  $\mathbf{A}^i$ . Si ha allora che la componente  $x_{i,1}$  della matrice colonna  $\mathbf{X}$  è data da:

$$x_{i,1} = \frac{\det(\mathbf{A}^1 \mathbf{A}^2 \dots \mathbf{A}^{i-1} \mathbf{S} \mathbf{A}^{i+1} \dots \mathbf{A}^n)}{\det(\mathbf{A})}$$

Vediamo un esempio. Sia dato il sistema

$$\left\{ \begin{array}{l} x_1 + x_2 = 1 \\ x_1 - x_2 = 2 \end{array} \right.$$

La matrice  $\mathbf{A}$  ha componenti:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

e determinante  $\det(\mathbf{A}) = -2$ , mentre la matrice colonna  $S$  ha componenti:

$$\mathbf{S} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Applicando la regola di Cramer si ha:

$$\begin{cases} x_1 = \frac{\begin{vmatrix} 1 & 1 \\ 2 & -1 \end{vmatrix}}{-2} = \frac{3}{2} \\ x_1 = \frac{\begin{vmatrix} 1 & 1 \\ 1 & 2 \end{vmatrix}}{-2} = -\frac{1}{2} \end{cases}$$

Che come si prova soddisfano il sistema:

$$\begin{cases} \frac{3}{2} - \frac{1}{2} = 1 \\ \frac{3}{2} + \frac{1}{2} = 2 \end{cases}$$

### **Caratteristica e rango di una matrice**

Per discutere la soluzione di un sistema **non** di Cramer è utile introdurre il concetto di caratteristica di una matrice. Consideriamo una matrice  $\mathbf{A}$   $m \times n$ , chiamiamo *minore* di ordine  $r$  una qualsiasi matrice ottenuta dalla intersezione di  $r$  righe e  $r$  colonne della matrice  $\mathbf{A}$ . Per esempio, data la matrice:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \end{bmatrix}$$

la matrice:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,4} \end{bmatrix}$$

è un minore di ordine 3, oppure

$$\mathbf{A} = \begin{bmatrix} a_{1,2} & a_{1,3} \\ a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{bmatrix}$$

è un minore di ordine 2. Ovviamente  $r$  deve essere minore o uguale al più piccolo tra  $m$  e  $n$ . Detto questo, definiamo caratteristica, l'ordine del più grande minore con determinante diverso da 0.

Definiamo ora la matrice completa di un sistema come la matrice ottenuta aggiungendo alla matrice  $\mathbf{A}$  dei coefficienti la matrice colonna  $\mathbf{S}$  dei risultati e la nominiamo  $\mathbf{A} : \mathbf{S}$  (in altri testi viene indicata usando le parentesi come segue:  $(\mathbf{A}, \mathbf{S})$ , ma si tratta sempre della matrice completa del sistema).

Bene, definita la caratteristica e la matrice completa siamo in grado di enunciare il teorema di Rouché-Capelli la cui tesi è che: un sistema lineare di equazioni ha soluzione se e solo se la caratteristica della matrice (incompleta)  $\mathbf{A}$  è uguale alla caratteristica della matrice completa  $\mathbf{A} : \mathbf{S}$ .

Questo teorema è molto utile nella risoluzione dei sistemi algebrici di equazioni lineari, comunque non ne faremo uso nel proseguo del testo e per questo non appesantisco la trattazione con ulteriori esempi.

## Matrice inversa

Il concetto di matrice inversa è già stato introdotto con l'equazione (1.14), di seguito riporto la formula generale per calcolare, data la matrice quadrata  $\mathbf{A}$  con determinante non nullo, gli elementi della matrice inversa. Detta quindi  $\mathbf{B}$  l'inversa di  $\mathbf{A}$  si ha che:

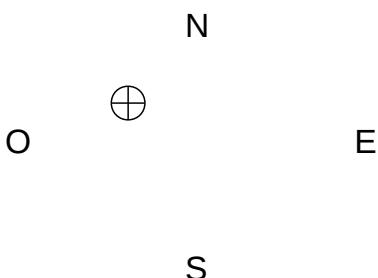
$$b_{i,j} = \frac{A_{j,i}}{\det(\mathbf{A})}$$

dove si noti che gli indici  $i$  e  $j$  sono scambiati, cioè al numeratore abbiamo la trasposta della così detta matrice aggiunta, cioè la matrice i cui elementi sono gli aggiunti (o complementi algebrici)

della matrice **A**.

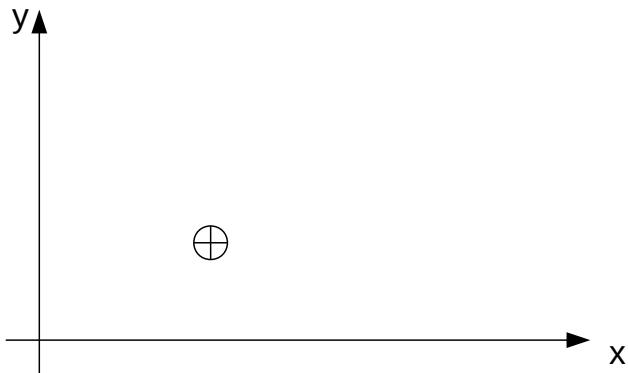
## 1.4 Coordinate e vettori

Consideriamo un punto in un piano, e per fissare le idee facciamo riferimento alla figura 1.2 dove si è evidenziato un punto usando una crocetta e si sono indicati i punti cardinali, giusto per intenderci sul fatto che stiamo facendo riferimento allo spazio concreto e non puramente matematico. Bene, se ora vi chiedessi di indicare la posizione del punto come procedereste? A parte indicarlo con il dito sul foglio del libro (o sullo schermo) non avete un sistema a cui riferirvi, l'assenza di altri punti nelle vicinanze non vi da modo di esprimere la sua posizione rispetto a qualcos'altro. Se invece vi mostrassi lo stesso punto in figura 1.3, cosa sareste in grado di dirmi?



*Fig. 1.2 - Un punto nello spazio. Non è fissato nessun sistema di assi.*

Se lo chiedessero a me risponderei che nelle vicinanze del punto ci sono due rette, o assi, e che il punto si trova più vicino alla retta orizzontale di quanto non si trovi vicino a quella verticale. La mia risposta rimane ancora imprecisa ma siamo sulla buona strada per renderla esatta. Ora mi invento uno stratagemma che consiste nel tracciare altre due rette, che siano perpendicolari a quelle presenti nel piano, ma che passino entrambe per il punto in questione (vedi Fig. 1.4).

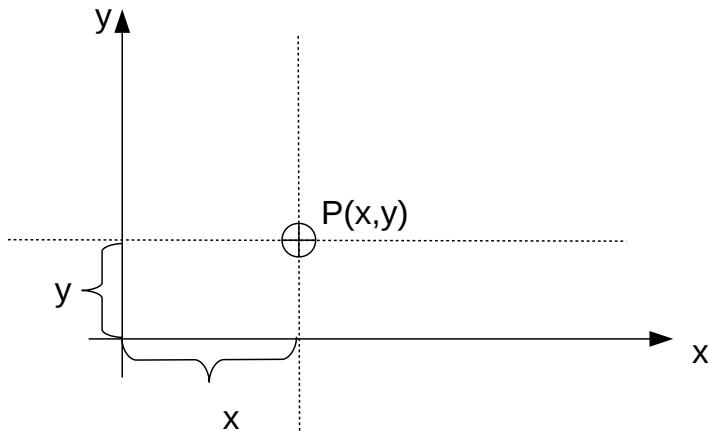


*Fig. 1.3 - Un punto nello spazio. Viene fissato un sistema di assi cartesiani.*

Come si può vedere dalla figura 1.4, le due rette che passano per il punto, al quale abbiamo assegnato l'etichetta  $P$ , intersecano l'una l'asse delle  $x$  e l'altra l'asse delle  $y$ . Misuriamo la distanza dal centro dei due assi fino alla loro intersezione. Indichiamo con la variabile  $x$  la distanza misurata sull'asse  $x$ , e con la variabile  $y$  quella sull'asse  $y$ . Ricordo che una variabile è solo un segnaposto per un numero. Fatto questo, alla mia domanda iniziale potrete rispondere sicuri che il punto  $P$  si trova in  $(x, y)$ , dove al posto delle variabili avrete sostituito le opportune misure eseguite lungo gli assi con il vostro bravo metro.

Ci sono una serie di proprietà che riguardano le posizioni dei punti e una serie di operazioni che sono indipendenti dal punto stesso. Possiamo esprimerele senza far riferimento ad un preciso punto ma rimanendo generici ed indicando il punto solo come  $P(x, y)$ . Per esempio è facile dimostrare che, se ho un secondo punto che chiamo  $Q$ , (vedi figura 1.4 bis) posso calcolare la distanza tra  $P$  e  $Q$  usando il teorema di Pitagora e ottenere una espressione generale di tale distanza secondo le coordinate dei due punti:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$



*Fig. 1.4 - Un punto nello spazio viene proiettato sugli assi cartesiani.*

Vediamo dunque che se consideriamo un punto, da un lato possiamo considerarne il valore delle sue coordinate rispetto ad un sistema di assi, per esempio la coppia  $(3,5)$ , dall'altro possiamo considerare in senso più astratto le sue componenti  $(x, y)$ . La coppia ordinata di valori  $(3,5)$  posso trattarla come una matrice riga  $(3,5)$  oppure anche come una matrice colonna  $\begin{bmatrix} 3 \\ 5 \end{bmatrix}$ . Le componenti  $(x, y)$  invece sono solo due variabili che considero insieme, ma in assenza di valori tangibili, ho difficoltà a definirle una matrice. Come abbiamo detto all'inizio una matrice è una tabella e tale vorremmo lasciarla. Per trattare le coppie o le terne di valori (poi estenderemo il ragionamento ad ancora più dimensioni) introduciamo una nuova quantità che chiamiamo vettore. Per farla breve, un elemento variabile, cioè un segnaposto, nello spazio, che sia appunto lo spazio ad una, due o tre dimensioni, lo chiamiamo vettore. Le diverse discipline che usano i vettori li distinguono dalle altre grandezze usando diverse notazioni tipografiche. In fisica un vettore si indica ponendo una freccetta sopra la lettera usata per il vettore, per esempio  $\vec{v}$  si usa frequentemente per indicare il vettore velocità. Normalmente, nei testi di algebra i vettori si indicano in grassetto come le matrici, ma usando una singola lettera minuscola, per

esempio:  $\mathbf{x}$  può essere usato per indicare il vettore posizione di un punto. Come avremo intuito, un vettore ha delle componenti, per esempio possiamo esprimerele in questo modo  $\mathbf{x} = (x, y)$  se si tratta di un vettore nel piano, o  $\mathbf{x} = (x, y, z)$  se il vettore è nello spazio (tre dimensioni). Se il vettore è un elemento di uno spazio astratto, come quello usato per rappresentare i possibili pesi delle connessioni sinaptiche sui dendriti di un neurone, ed ha più di tre dimensioni è uso numerare le sue componenti come segue:  $\mathbf{x} = (x_1, x_2, \dots, x_{n-1}, x_n)$ .

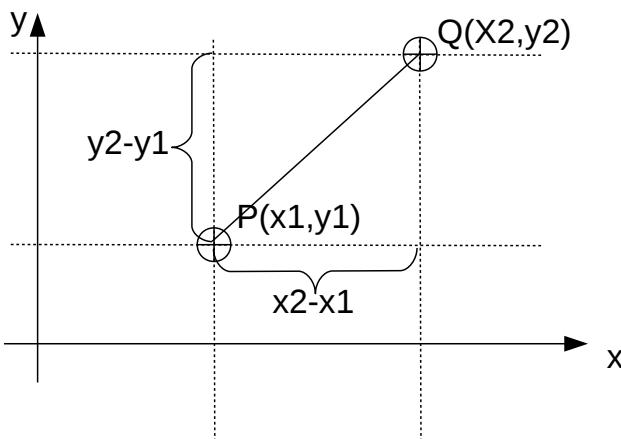


Fig. 1.4 bis - Distanza tra due punti nello spazio.

Spesso quando si parla di vettori si intendono delle freccette che hanno una lunghezza ed un verso (il verso in cui punta la freccia). Vediamo subito che si sta parlando sempre della stessa cosa di cui abbiamo parlato fin'ora. Osserviamo la freccia indicata in figura 1.5, che corrisponde esattamente all'idea di vettore che ci viene proposta già dalle prime lezioni di fisica elementare alle scuole medie o superiori. Per esso definiamo le componenti  $\vec{v} = (v_x, v_y)$ . Se adesso proviamo a trasportare il "sedere" del vettore nel punto di incrocio degli assi notiamo che esso punta esattamente al punto P (vedi figura 1.6).

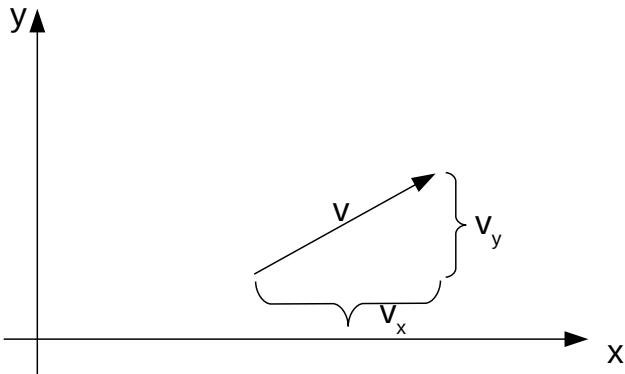


Fig. 1.5 - Un vettore nel piano cartesiano.

Come si intuisce esiste quindi una precisa relazione tra un vettore inteso come una freccia e un vettore inteso come un punto nello spazio. Si tratta sempre della stessa entità matematica, che però viene rappresentata in modo differente in base all'esigenza. I vettori freccia, si chiamano vettori *applicati*. Comunque non ne faremo uso in questo testo, per cui non approfondiamo oltre il discorso.

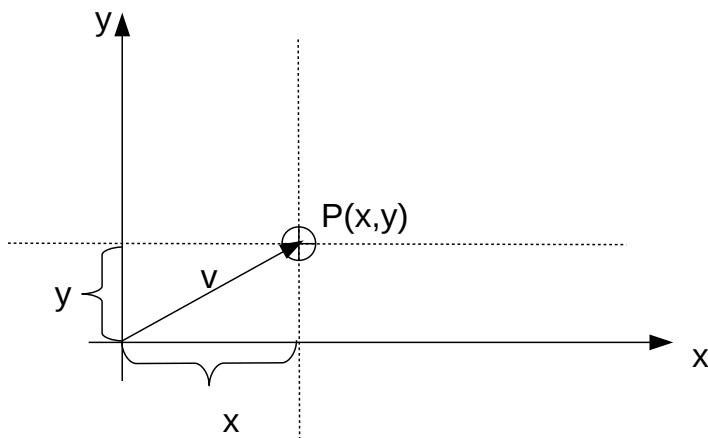


Fig. 1.6 - Un vettore traslato all'origine degli assi.

## 1.5 Prodotto scalare

Nel paragrafo precedente abbiamo introdotto l'idea di vettore e della matrice lineare (riga o colonna) che possiamo usare per rappresentarne le componenti. La relazione esistente tra vettori e matrici lineari non è una cosa tanto semplice dal punto di vista

formale. Purtroppo anche la forma è importante, e se certi problemi sembrano trattabili in termini intuitivi, poi possono portare ad importanti contraddizioni. Proprio per chiarire la relazione tra vettori e matrici è nata la nozione di **tensore**. Questo l'ho scritto in grassetto per farlo notare bene. Nei nostri giorni si parla spesso di tensori, forse anche per darsi delle arie, chissà. I tensori sono purtroppo una complicazione necessaria quando si devono formalizzare alcuni conti algebrici. Comunque, per ora possiamo stabilire le seguenti relazioni:

1. Un tensore di tipo (0,1) è un vettore colonna.
2. Un tensore di tipo (1,0) è un vettore riga.
3. Un tensore di tipo (1,1) è una matrice.

Detto questo introduciamo il prodotto scalare tra vettori colonna. Prendiamo due tensori  $\mathbf{v}$  e  $\mathbf{w}$  di tipo (0,1), e definiamo il prodotto scalare  $\mathbf{v} \cdot \mathbf{w}$  tra loro nel modo seguente:

$$\mathbf{v} \cdot \mathbf{w} = v_1 w_1 + v_2 w_2 + \dots + v_{n-1} w_{n-1} + v_n w_n \quad (1.21)$$

Come si evince dalla (1.21) il prodotto scalare è un numero (scalare) non un vettore. La (1.21) è definita sia per il prodotto di tensori di tipo (0,1) che di tipo (1,0). Se invece moltiplichiamo tra loro un vettore di tipo colonna per uno di tipo riga otteniamo una matrice, vediamo un esempio:

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \cdot [w_1 \quad w_2 \quad \dots \quad w_n] = \begin{bmatrix} v_1 w_1 & v_1 w_2 & \dots & v_1 w_n \\ v_2 w_1 & v_2 w_2 & \dots & v_2 w_n \\ \vdots & & & \\ v_n w_1 & v_n w_2 & \dots & v_n w_n \end{bmatrix}$$

I vettori riga possono essere scritti anche come la trasposta di un vettore colonna (e vice versa), quindi ad esempio se intendiamo che  $\mathbf{w}$  sia un vettore colonna, con la scrittura  $\mathbf{w}^T$  intenderemo la corrispondente riga. Pertanto la scrittura  $\mathbf{v} \cdot \mathbf{w}$  sarà da calcolare

secondo la (1.21), mentre la scrittura  $\mathbf{v} \cdot \mathbf{w}^T$  secondo la (1.22). Se mi sono spinto in questa trattazione complicata non l'ho fatto per sadismo matematico, ma perché questo conto in particolare è alla base della trattazione matematica del meccanismo di *apprendimento inverso* delle reti neurali supervisionate. Su molti testi troverete questa equazione:

$$\Delta \mathbf{w} = \Delta \mathbf{a}^T \Delta \mathbf{z} = \Delta \mathbf{a}^T \left[ (\mathbf{a} - \mathbf{y}) \sigma'(\mathbf{z}) \right]$$

che si riferisce alla variazione (il  $\Delta$  si usa per le variazioni) della matrice dei pesi sinaptici, scritta in funzione dell'uscita, dello stato di attivazione e del valore desiderato per una rete neurale artificiale.

Dell'equazione sopra si dovrebbe riuscire a comprendere bene o male tutto, tranne due cosette:

1. Cosa significa la scrittura  $\sigma(z)$ ?
2. Peggio ancora, cosa significa  $\sigma'(z)$ ?

Bene, lo vediamo nel prossimo paragrafo, che concluderà anche la sezione introduttiva di algebra.

## 1.6 Funzioni e derivate

Nel primo paragrafo abbiamo studiato i polinomi. Abbiamo visto che essi sono espressioni matematiche che comprendono sia una parte letterale che una numerica. Abbiamo anche puntualizzato che la parte letterale può essere intesa in due modi, sia come una lettera che tale rimarrà vita, natural durante, sia come una variabile, cioè una lettera che usiamo solo per indicare che al suo posto possiamo sostituire un numero qualunque. Bene, ricordato questo, consideriamo la seguente espressione:

$$a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x^1 + a_n \quad (1.23)$$

dove  $a_0, a_1, \dots, a_n$  sono dei numeri detti coefficienti della funzione.

Per esempio:

$$3x^5 + 2x^4 + x^3 + 1.12x^2 + 2 \quad (1.24)$$

è un esempio corretto della (1.23). Notiamo che alcuni coefficienti possono valere 0 ( $a_4$ ) oppure 1 ( $a_2$ ).

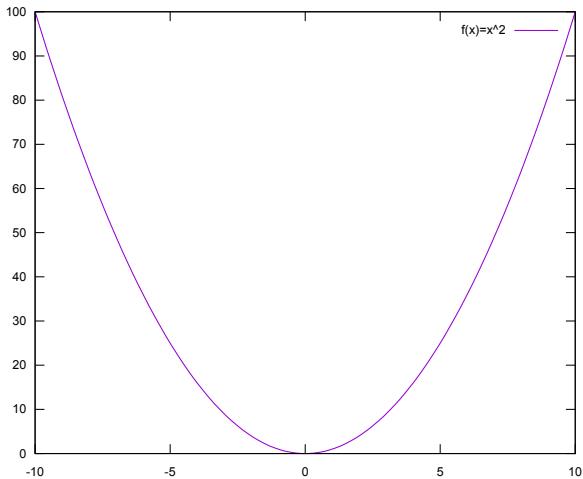
Al variare di  $x$ , possiamo calcolare il valore della (1.24) e vediamo che varia anche esso, per questo la (1.24) è detta essere una espressione funzione di  $x$ , o più semplicemente una funzione di  $x$  e si scrive:

$$f(x) = 3x^5 + 2x^4 + x^3 + 1.12x^2 + 2$$

Se si considerano le coppie di punti  $(x, f(x))$  e li si rappresenta sul piano cartesiano è possibile ottenere la rappresentazione grafica di una funzione. Consideriamo a titolo di esempio la funzione  $f(x) = x^2$ , calcolandone il valore assegnando una certa serie di valori alla variabile  $x$  è possibile ottenere un grafico come quello riportato in figura 1.7. Per completezza è giusto aggiungere che è uso comune scrivere insieme alla definizione di una funzione anche la seguente equazione:

$$\begin{cases} f(x) = x^2 \\ y = f(x) \end{cases}$$

con la quale diventa più intuitiva la relazione esistente tra la funzione e i punti nel piano.



*Fig. 1.7 - Grafico della funzione  $f(x) = x^2$ .*

Ora che abbiamo visto come ottenere delle funzioni attraverso i polinomi, diciamo subito che le espressioni che si possono comporre possono diventare complicate a piacere, e possono coinvolgere anche infiniti termini, in questo caso si parla di serie. Dico questo per spiegare che i grafici che si possono ottenere sono davvero incredibili, come quello che riporto in figura 1.8 che si riferisce appunto allo sviluppo in serie di potenze della  $x$  di una funzione nota come tangente iperbolica il cui simbolo è  $\tanh(x)$ . Introduciamo questa funzione perché è una delle classiche che viene usata per calcolare l'attivazione di un neurone. La funzione  $\sigma$  vista sopra è una sorta di segnaposto per le possibili funzioni che si possono usare a questo scopo. Ho fatto l'esempio della tangente iperbolica perché è quella che useremo nell'esempio in C nella terza sezione.

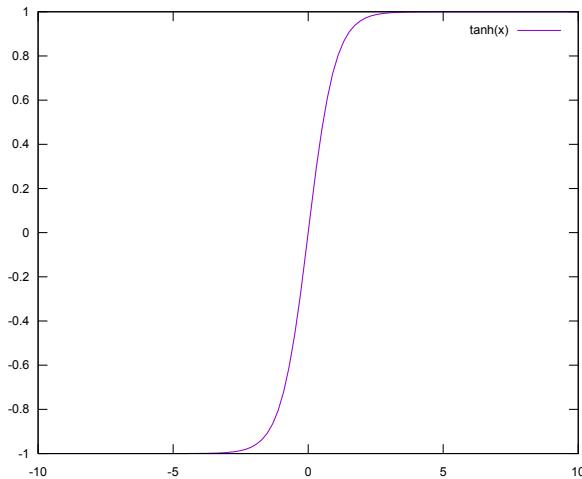


Fig. 1.8 - Grafico della funzione  $f(x) = \tanh(x)$ .

Per completare la trattazione matematica propedeutica all'analisi dei modelli delle reti neurali artificiali manca ancora il concetto di derivata di funzione, appunto quella che abbiamo indicato apponendo l'apice sopra il simbolo della sigma:  $\sigma'(x)$ . Anzitutto diciamo che la derivata può essere indicata anche in altri modi, per indicare la derivata di una funzione in una sola variabile  $x$  si usano indifferentemente queste espressioni:

- $Df(x)$ ,
- $f'(x)$ ,
- $\frac{df(x)}{dx}$

La derivata di una funzione è anch'essa una funzione, per esempio si ha:

$$\begin{aligned} f(x) &= x^2; \\ g(x) &= f'(x) = 2x, \\ g(x) &= 2x \end{aligned}$$

La derivata rappresenta il rateo di variazione della funzione rispetto alla variabile indipendente. Una trattazione completa di questo tema non mi è possibile in questo contesto. Ai fini della comprensione del resto del testo, ed in particolar modo dell'esempio riportato nella

terza sezione, ci basta sapere che la derivata della funzione  $\tanh(x)$  è la funzione  $1 - \tanh^2(x)$ .

## 2. Parte seconda: fondamenti di linguaggio C

Il linguaggio C è stato scritto da Brian Kernighan e Dennis Ritchie. Si tratta di un linguaggio imperativo caratterizzato dai seguenti aspetti:

- possibilità di controllare il flusso di esecuzione delle istruzioni mediante specifici costrutti sintattici
- possibilità di strutturare il codice in blocchi detti funzioni
- possibilità di accedere al contenuto della memoria mediante l'indirizzo hardware

Il terzo punto è quello che rende il linguaggio C, o C come spesso abbrevieremo, particolarmente utile nella programmazione di componenti del sistema operativo e in generale quando l'efficienza e la velocità sono essenziali. I primi due punti sono invece comuni a molti linguaggi di programmazione come il Java, il Pascal, il Fortran ecc. Chiariamo subito che il C non è un linguaggio ad oggetti mentre è ad oggetti il suo cuginetto C++.

### 2.1 Architettura del Personal Computer

La comprensione totale di un fenomeno probabilmente è qualcosa che compete solo a nostro Signore, ma, la comprensione generale di ciò che si sta facendo è una buona regola quando si intende lavorare con coscienza. Questo vale in molte professioni e senz'altro vale nell'informatica dove la tentazione di appoggiarsi eccessivamente agli strati tecnologici "inferiori" per applicarsi solo a problemi "superiori" può facilmente prendere la mano. In altre parole sto dicendo che anche se si potrebbe imparare a programmare in C e a scrivere algoritmi basati sulle reti neurali senza comprendere come funziona un elaboratore elettronico, sarebbe una strada che porterebbe a poco e prima o poi si fermerebbe. Le tecnologie hardware informatiche stanno rapidamente evolvendo. In pochi anni si è passati da elaboratori basati solo sull'utilizzo della *central processing unit* (CPU)

ad elaboratori che hanno sfruttato la *graphical processing unit*, fino ad elaboratori che sfruttano o sfrutteranno le nascenti (al momento in cui scrivo) *neural processing unit* (NPU). Per comprendere come scrivere e ottimizzare il software che si vuol far girare su tali tecnologie, è necessario comprendere almeno i fondamenti delle tecnologie stesse. In questo testo, mi limito ad una introduzione di base ma sufficiente a comprendere come un programma software possa essere eseguito su una architettura hardware.

## Primo passo: gli algoritmi

### L'algoritmo di un telaio

Negli anni '30 del XX secolo, il matematico e logico Alan Turing dimostrò che in linea di principio si poteva costruire una macchina capace di eseguire un algoritmo le cui istruzioni non fossero parte costituente della macchina stessa. Chiariamo: un telaio, di quelli che si usano da centinaia di anni per tessere i tessuti, esegue anch'esso un algoritmo, cioè esegue una sequenza di istruzioni precise che porta ad un preciso risultato, ma le istruzioni che segue sono codificate nella sua stessa struttura, cioè è costruito in modo che agendo con una forza meccanica dall'esterno, si mettano in moto una serie di ingranaggi che portano ad una sequenza ciclica di movimenti che ha come risultato finale quello della tessitura: cioè fare scorrere la navetta da un lato all'altro dell'ordito mentre srotola il filo della trama. Ad ogni passaggio, inoltre, gli ingranaggi del telaio muovono verso l'alto i fili dell'ordito che si trovano in basso, e quelli che si trovano in alto li muovono verso il basso, bloccando in questo modo il filo di trama. I movimenti del telaio producono un tessuto, e l'algoritmo per farlo è espresso nei termini degli ingranaggi che muovono le parti meccaniche del telaio. L'algoritmo può essere schematizzato come segue:

1. Apertura dell'ordito: i fili dell'ordito vengono separati alzandone uno ogni due, venendo a creare una sorta di forbice che

- permette il passaggio della navetta.
2. Passaggio della trama: la navetta guida del filo di trama viene passata da destra a sinistra.
  3. Chiusura dell'ordito: i fili sollevati vengono abbassati mentre quelli rimasti in basso vengono sollevati in modo da intrappolare il filo di trama.
  4. Passaggio della trama: la navetta guida del filo di trama viene passata da sinistra a destra.
  5. Si riparte dal punto 1.

Quindi il telaio esegue un algoritmo, però, se alla stessa macchina (il telaio) chiedessimo di eseguire una operazione di tornitura, non otterremmo nessun risultato, e per trasformarla da un telaio ad un tornio ci vorrebbe un intervento sulla sua struttura fisica (hardware), spostando, avvitando o saldando le sue parti secondo un diverso principio di funzionamento.

### **La macchina di Turing**

L'idea di Turing fu invece quella di una macchina progettata per eseguire un compito semplice quanto bizzarro. Si trattava di una macchina con il compito di far scorrere un nastro su cui, a distanza regolare, erano disposti dei simboli (per esempio, ottenuti forando in modo opportuno il nastro stesso). Lo scorrimento del nastro portava un simbolo alla volta sotto una testina, o sonda, capace di reagire al simbolo presentato modificando il contenuto del nastro presentato sotto la testina e poi spostando la testina a sinistra o a destra in base a quanto letto. Al termine dell'esecuzione, cioè quando il nastro non si muoveva più sotto la testina, sul nastro era riportata una sequenza di simboli diversa da quella iniziale. L'idea di Turing era che la sua macchina eseguisse sempre lo stesso algoritmo "meccanico" descritto dalla lista seguente:

1. Leggere un simbolo sotto la testina, se il simbolo corrisponde al simbolo speciale "stop" fermare l'esecuzione.
2. Scrivere un nuovo simbolo.

3. Muovere il nastro.
4. Ripetere da 1.

ma che al contempo, sul nastro fosse riportato il programma di un altro algoritmo, e che la sequenza di simboli riportata sul nastro al termine dell'esecuzione costituisse il risultato dell'algoritmo codificato nel nastro prima dell'esecuzione.

Il risultato di cosa, viene da chiedersi. Il risultato della funzione che si era chiesto di calcolare alla macchina. Ma chi e come aveva chiesto alla macchina di calcolare una funzione? La richiesta era codificata sul nastro nei termini della sequenza di simboli scritti o incisi sopra. In pratica, sul nastro era presente la sequenza di istruzioni che dovevano essere eseguite dalla macchina.

### **Da Turing al calcolatore**

La cosa che oggi potrebbe stupire, è che una macchina di quel tipo ha assolutamente le stesse capacità espressive di un calcolatore moderno, cioè qualsiasi programma che può essere eseguito su una architettura attuale, può essere eseguito anche da una macchina di Turing.

Voglio sottolineare la differenza concettuale tra il telaio tessile, che è costruito in modo da compiere dei movimenti, ognuno dei quali è funzionale allo scopo per cui è stato progettato, e la macchina di Turing, dove gli ingranaggi sono congegnati per compiere una procedura ripetitiva che però non ha una relazione diretta con il risultato. Questo punto è di estrema importanza, infatti, vedremo tra poco che i moderni calcolatori seguono lo stesso principio: hanno un'architettura hardware che esegue sempre il seguente algoritmo:

1. Fetch: pesca un'istruzione.
2. Decode: decodifica l'istruzione.
3. Execute: esegui l'istruzione.
4. Vai al punto 1.

mentre le istruzioni software codificano l'algoritmo pensato dal

programmatore.

In questo contesto non entriamo nel dettaglio della macchina di Turing, si trovano moltissimi riferimenti in rete e, per chi fosse interessato, il problema è trattato in modo formale anche nelle mie dispense del corso di Linguaggi di Programmazione del 2014 (si trova ancora qualcosa in rete) o ancora meglio nel testo di Carlucci (vedi bibliografia in fondo al testo). La cosa che mi premeva evidenziare è che una generica funzione, può essere scomposta in piccole istruzioni le quali possono essere eseguite da una macchina "stupida" cioè incosciente di ciò che sta facendo. Non solo, ricordiamo che esistono esempi di macchine di Turing realizzate in legno e che usano una sorta di mulino ad acqua come forza motrice. Esistono anche stupendi esempi realizzati con i componenti Lego®. La domanda che può sorgere è che relazione abbia l'esecuzione di una funzione con i software che siamo abituati a vedere in esecuzione sui dispositivi elettronici. Ebbene, tutti quei software altro non sono che la codifica in *istruzioni macchina* di una specifica funzione, quindi programmare significa in realtà scrivere funzioni.

Mi permetto di riportare una piccola osservazione storica. A quanto risulta, stando al lavoro pubblicato da Turing stesso, "On computable numbers, with an application to the Entscheidungsproblem", il suo scopo non era quello di dimostrare la possibilità di realizzare la macchina in questione, ma attraverso essa discutere l'allora "caldo" problema della fermata, che non discuterò qui. Le implicazioni delle idee proposte da Turing furono però colte dal fisico von Neumann, che lavorò ai primi elaboratori elettronici e al quale si deve la nota architettura omonima.

## **Algoritmi e programmi**

Da questa lunga dissertazione risulta una piccola ma importante informazione:

**gli algoritmi sono sequenze di azioni che devono poter**

**essere eseguite da una macchina o anche da un essere umano che le esegua in modo automatico**

come la procedura che seguiamo per eseguire una moltiplicazione in colonna. La codifica di dette istruzioni mediante uno specifico linguaggio adatto alla macchina si chiama *programma*. Detto questo, a chi si è preso il tempo di approfondire bene come funziona una macchina di Turing, sarà evidente che quella non è la soluzione ideale per eseguire dei programmi complessi. L'elettronica digitale, assieme all'algebra di Boole, hanno fornito la tecnologia e la base matematica per costruire delle alternative alla macchina di Turing. Torno ad insistere sul fatto che la macchina di Turing, presenta delle limitazioni tecnologiche, ma non concettuali.

## **Secondo passo: logica e matematica**

Sarò conciso su questo tema, ma è chiaro che per chi lo desidera, una volta capitone l'importanza, di materiale di approfondimento se ne trova tanto. Vediamo di capire le basi. Abbiamo visto nella prima parte, che l'algebra si occupa di come calcolare le espressioni simboliche espresse correttamente. Abbiamo visto che le regole dell'algebra non si limitano ai numeri ma possono essere estese anche al calcolo letterale mantenendo coerenza e significato. Vediamo in questo paragrafo che è possibile anche creare un'algebra per descrivere le regole della logica *proposizionale*, cioè la logica introdotta da Aristotele (quella dei sillogismi per intenderci). Tale algebra ha preso il nome di Algebra di Boole, dal matematico George Boole che l'ha codificata.

### **Algebra di boole**

L'idea, molto diretta, consiste nell'indicare le proposizioni (frasi) usando delle variabili, per esempio  $x$  e  $y$  e assegnando a loro un *valore logico* di vero o falso.

Per esempio le due proposizioni "Il C è facile" e "Il C è veloce" potrebbero essere rappresentate con le variabili  $x$  e  $y$  rispettivamente. A questo punto se volessimo dire che il C è facile e veloce potremmo affermare quanto segue:  $x \wedge y$ . Il segno  $\wedge$  è un nuovo segno logico che possiamo definire vedendo come agisce sui valori VERO e FALSO esistenti nella logica:

- Vero  $\wedge$  Vero = Vero
- Vero  $\wedge$  Falso = Falso
- Falso  $\wedge$  Vero = Falso
- Falso  $\wedge$  Falso = Falso

Quindi, dalla prima di queste relazioni, scopriamo che se il linguaggio C è sia facile che veloce, allora l'espressione  $x \wedge y$  è vera. In effetti l'operazione sottintesa dal simbolo  $\wedge$  è la congiunzione logica, quella che nelle proposizioni si indica con la lettera e:

## Il C è facile e il C è veloce

Quindi possiamo vedere  $x \wedge y$  come una espressione algebrica di cui si può calcolare il valore e il cui valore dipende dal valore assegnato alle variabili  $x$  e  $y$ .

## Gli operatori

In informatica, ma, anche in matematica, si usa parlare non solo di *operazione* ma spesso di *operatore*, sottintendendo che l'azione della operazione è dovuta all'operatore. Bisogna abituarsi a questa visione e accettare che  $\wedge$  così come  $+$  e  $-$  sono operatori che agiscono sugli operandi, cioè variabili o valori.

Ci sono altri due operatori logici (esiste anche un altro operatore che qui non trattiamo), questi sono la *disgiunzione logica* indicata con il simbolo  $\vee$  e la negazione, indicata con il simbolo  $\neg$ , e sono definiti rispettivamente come:

- Vero  $\vee$  Vero = Vero
- Vero  $\vee$  Falso = Vero
- Falso  $\vee$  Vero = Vero
- Falso  $\vee$  Falso = Falso

e

- $\neg$  Vero = Falso
- $\neg$  Falso = Vero

## Associazione tra logica e l'algebra

La logica è dotata di un apparato sintattico, cioè di regole per verificare che le espressioni siano ben formate e, con le regole viste sopra, di un apparato semantico, cioè di un modo automatico per stabilire il valore di una espressione. La cosa che rende particolarmente interessante questa logica è che usa due soli valori che si possono facilmente associare all'1 (Vero) e allo 0 (Falso), così come gli operatori  $\wedge$  e  $\vee$  si possono associare agli operatori algebrici  $\cdot$  e  $+$  (ora preferisco usare il simbolo  $\cdot$  per la moltiplicazione piuttosto che il simbolo  $\times$  che crea confusione con le variabili). Con queste

associazioni possiamo riscrivere le regole semantiche per la congiunzione ( $\wedge$ ) come segue:

$$\begin{aligned} 1 \cdot 1 &= 1 \\ 1 \cdot 0 &= 0 \\ 0 \cdot 1 &= 0 \\ 0 \cdot 0 &= 0 \end{aligned} \tag{2.1}$$

e quelle per la disgiunzione ( $\vee$ ):

$$\begin{aligned} 1 + 1 &= 1 \\ 1 + 0 &= 1 \\ 0 + 1 &= 1 \\ 0 + 0 &= 0 \end{aligned} \tag{2.2}$$

e quelle della negazione ( $\neg$ ):

$$\begin{aligned} 1 - 1 &= 0 \\ 1 - 0 &= 1 \end{aligned} \tag{2.3}$$

## Proprietà algebriche della logica di Boole

Con le (2.1), le (2.2) e le (2.3) abbiamo in realtà definito un'*algebra* con alcune importanti proprietà. Nel linguaggio matematico, un'algebra o più comunemente un *anello* è una terna costituita da un insieme  $K$  e due operazioni che si è soliti indicare con i due simboli  $\cdot$  e  $+$ . Nel caso dell'algebra di Boole,  $K$  è l'insieme costituito dai due numeri 0 e 1, quindi  $K = \{0, 1\}$  e, una volta definite le due operazioni  $\cdot$  e  $+$  attraverso la (2.1) e la (2.2) si ha che valgono le proprietà che seguono:

- Commutativa:  $x + y = y + x$ ,  $x \cdot y = y \cdot x$ , per esempio si ha:  
 $1 + 0 = 0 + 1$ ,  $1 \cdot 0 = 0 \cdot 1$
- Associativa:  $x + (y + z) = (x + y) + z$ ,  $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ ,  
per esempio si ha:  $1 + (1 + 1) = (1 + 1) + 1$  e  
 $1 \cdot (1 \cdot 1) = (1 \cdot 1) \cdot 1$

- Assorbimento:  $x + (x \cdot y) = x$ ,  $x \cdot (x + y) = x$ , per esempio si ha:  
 $0 + (0 \cdot 1) = 0$ ,  $0 \cdot (0 + 1) = 0$
- Distributiva:  $x \cdot (y + z) = x \cdot y + x \cdot z$ , per esempio si ha:  
 $1 \cdot (1 + 1) = 1 \cdot 1 + 1 \cdot 1$
- Idem-potenza:  $x + x = x$ ,  $x \cdot x = x$  per esempio si ha:  
 $1 \cdot 1$ ,  $1 + 1 = 1$
- Complemento:  $x \cdot \neg x = 0$ ,  $x + \neg x = 1$  per esempio si ha:  
 $1 \cdot 0 = 0$ ,  $1 + 0 = 1$
- Minimo e massimo: 1 e 0

Ora che abbiamo introdotto questa relazione tra logica e algebra possiamo concentrarci su alcune applicazioni di quest'algebra. Notiamo anzi tutto che possiamo definire delle funzioni basate sulle operazioni  $\cdot$  e  $+$  e dette funzioni possono essere di una sola variabile o di più variabili. Siccome una funzione è definita dal valore che assume in base al suo argomento, per definire le funzioni di tipo booleano possiamo usare una semplice tabella in cui riportiamo il valore dell'argomento e il valore della funzione. Detta  $x$  la variabile booleana (cioè una variabile che assumerà solo valori 0 o 1) e  $f(x)$  una funzione booleana, cioè basata solo su espressioni algebriche date da combinazioni di operatori booleani (quindi  $f(x) = \log(x)$  non va bene), possiamo definire  $f(x)$  per mezzo di una tabella:

*Tabella 1. Funzione booleana ad una variabile*

$x$	$f(x)$
1	$f(1)$
0	$f(0)$

La variabile  $x$  può assumere solo due valori e lo stesso vale per la funzione che, essendo costituita solo da espressioni booleane, assumerà valori nell'insieme  $K$  definito sopra. Pertanto, una funzione ad una sola variabile può essere completamente definita con sole due

righe di una tabella a due colonne. Possiamo anche facilmente tabellare tutte le possibili funzioni di una sola variabile come segue:

*Tabella II. Funzioni booleane di una variabile booleana.*

$x$	$f(x)$	$g(x)$	$h(x)$	$i(x)$
1	0	1	1	0
0	0	1	0	1

Le funzioni riportate in Tab. II sono tutte le possibili funzioni booleane di una sola variabile. La prima e la seconda sono due funzioni costanti mentre la terza e la quarta sono l'identità e il complemento (o negazione):

$$\begin{aligned} f(x) &= 0, \\ g(x) &= 1, \\ h(x) &= x, \\ i(x) &= 1 - x \end{aligned}$$

Sempre usando una tabella si può definire una funzione a più variabili. Per esempio la funzione ( $f(x, y, z)$ ) delle tre variabili  $x$ ,  $y$ , e  $z$  può essere definita come in tabella III: Se ora diamo una definizione della funzione  $f$  in termini di operatori booleani, per esempio:

$$f(x, y, z) = x \cdot z + y$$

otteniamo la tabella IV, dove abbiamo sostituito i valori  $f(x, y, z) = x \cdot z + y$  nell'ultima colonna di tabella III.

*Tabella III. Tabella booleana ad una funzione di tre variabili.*

$x$	$y$	$z$	$f(x, y, z)$
1	1	1	$f(1, 1, 1)$
1	1	0	$f(1, 1, 0)$
1	0	1	$f(1, 0, 1)$

1	0	0	$f(1, 0, 0)$
0	1	1	$f(0, 1, 1)$
0	1	0	$f(0, 1, 0)$
0	0	1	$f(0, 0, 1)$
0	0	0	$f(0, 0, 0)$

Tabella IV. Tabella booleana ad una funzione di tre variabili.

$x$	$y$	$z$	$f(x, y, z)$
1	1	1	1
1	1	0	1
1	0	1	1
1	0	0	0
0	1	1	1
0	1	0	1
0	0	1	0
0	0	0	0

## Sistema a due stati

La cosa davvero interessante è che avremmo anche potuto procedere al contrario, cioè avremmo potuto prima assegnare a piacere i valori della funzione  $f(x, y, z)$  per tutte le possibili combinazioni di valori che possono assumere le tre variabili booleane e, poi cercare la funzione booleana che li generava. Si può dimostrare che

**esiste sempre una espressione booleana che soddisfa qualsiasi tabella booleana.**

Questa proprietà, che discende direttamente dall'algebra, suggerisce l'uso delle funzioni booleane per rappresentare lo stato di sistemi non propriamente matematici. Proviamo a fare un esempio.

Si supponga di avere una porta che può aprirsi o chiudersi in base allo stato di tre interruttori. Le regole sono date dalla seguente tabella:

*Tabella V. Configurazione di apertura della porta in base allo stato degli interruttori I1, I2 e I3.*

I1	I2	I3	Porta
ON	ON	ON	CHIUSA
ON	ON	OFF	CHIUSA
ON	OFF	ON	CHIUSA
ON	OFF	OFF	CHIUSA
OFF	ON	ON	APERTA
OFF	ON	OFF	CHIUSA
OFF	OFF	ON	CHIUSA
OFF	OFF	OFF	CHIUSA

Il problema può essere facilmente trasformato in un problema booleano e si può ricavare la funzione  $f : \text{PORTA CHIUSA} \rightarrow f = 0$  e  $\text{PORTA APERTA} \rightarrow f = 1$ . La funzione  $f$  che soddisfa i criteri di Tabella V è:

$$f(x, y, z) = \neg x \cdot y \cdot z$$

## Sistema a più stati

Possiamo inoltre estendere l'applicazione delle funzioni booleane a sistemi che possono assumere anche più di due stati. Supponiamo per esempio di voler rappresentare gli stati di un semaforo stradale che è composto da tre luci indipendenti di colore diverso che possono essere accese o spente l'una indipendentemente dall'altra. Indichiamo con V, G e R le tre luci semaforiche e supponiamo, è solo un esempio, che il loro stato (acceso/spento) dipenda da tre variabili "stradali":

- P: indica se un pedone ha schiacciato il richiamo pedonale.
- D: indica se sono trascorsi cinque secondi da quando il pedone ha schiacciato il pulsante.
- T: indica se sono trascorsi 20 secondi da quando il pedone ha schiacciato il pulsante.

Il semaforo per le automobili deve essere verde se il pedone non ha premuto il richiamo pedonale. Se il pedone lo preme, il semaforo deve diventare prima giallo per cinque secondi poi rosso. Deve tornare verde dopo venti secondi. Queste richieste possono essere tabulate come segue:

*Tabella VI. Configurazioni di attivazione di un semaforo stradale. P, D e T sono variabili a due stati che possono assumere i valori ON e OFF, rappresentano il tasto di richiamo pedonale, e dei flag sull'intervallo di tempo passato dal richiamo. R, G e V rappresentano lo stato delle luci semaforiche.*

P	D	T	G	R	V
ON	ON	ON	OFF	OFF	ON
ON	ON	OFF	OFF	ON	OFF
ON	OFF	ON	ON	OFF	OFF
ON	OFF	OFF	ON	OFF	OFF
OFF	ON	ON	OFF	OFF	ON

OFF	ON	OFF	OFF	OFF	ON
OFF	OFF	ON	OFF	OFF	ON
OFF	OFF	OFF	OFF	OFF	ON

Alcune delle configurazioni previste in tabella VI dovrebbero essere impedisite dall'elettronica analogica che controlla il semaforo, ma, a parte ciò, vogliamo porre l'attenzione su come la tabella VI possa essere rappresentata da tre funzioni booleane ognuna in tre variabili. Se si associano P, D e T alle variabili booleane  $x$ ,  $y$  e  $z$  e G, R e V alle funzioni booleane  $f(x, y, z)$ ,  $g(x, y, z)$  e  $h(x, y, z)$  si ottiene subito l'associazione voluta, che può essere rappresentata come in tabella VII.

*Tabella VII. Trasposizione booleana della tabella VI.*

$x$	$y$	$z$	$f$	$g$	$h$
1	1	1	0	0	1
1	1	0	0	1	0
1	0	1	1	0	0
1	0	0	1	0	0
0	1	1	0	0	1
0	1	0	0	0	1
0	0	1	0	0	1
0	0	0	0	0	1

Proviamo a vedere l'esempio del semaforo da questa ottica:

*Come correlare le configurazioni di un interruttore e due timers alle configurazioni di accensione di tre luci colorate?*

La risposta è:

*Attraverso tre funzioni booleane ognuna delle quali ha tre variabili booleane.*

Quindi abbiamo associato delle configurazioni di 1 e 0 ad altre

configurazioni di 1 e 0 dimostrando che si possono usare delle configurazioni di stati logici o numerici (sequenze di 1 e 0) per controllare altre configurazioni di stati logici o numerici (sequenze di 1 e 0). Quindi:

**realizzare un sistema di calcolo capace di trasformare e "trasportare" delle configurazioni (informazioni), esattamente come un sistema ad ingranaggi è capace di trasformare e trasportare l'energia meccanica.**

Torneremo su questo tra poco, per ora vediamo di completare l'analisi del sistema semaforico. Quello che manca è dare la corretta espressione per le tre funzioni booleane  $f(x, y, z)$ ,  $g(x, y, z)$  e  $h(x, y, z)$ . Partiamo dalla  $f$  per la quale si ha:

$$f(x, y) = x \cdot \neg y$$

che come vediamo non ha una dipendenza esplicita da  $z$ . Poi la funzione  $g$  che è data da:

$$g(x, y, z) = \neg z \cdot x \cdot y$$

ed infine  $h$ :

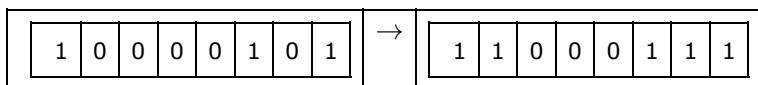
$$h(x, y, z) = \neg x + y \cdot z$$

Esistono dei metodi per ricavare l'espressione booleana di una funzione a partire dalla sua tabella, che, per inciso, si chiama *tabella di verità*. L'analisi dettagliata di detti metodi esula, cioè è al di fuori, dallo scopo di questo testo, mentre voglio porre l'attenzione ancora su due aspetti:

1. Preso un numero  $n$  di variabili booleane indipendenti, la tabella corrispondente a tutte le configurazioni di 1 e 0 possibili per dette variabili ha esattamente  $2^n$  righe (provare per credere!) e ovviamente  $n$  colonne.
2. Aggiunte  $m$  colonne alle  $n$  già esistenti e compilate con 0 e 1 a piacere le relative caselle, per ognuna delle nuove  $m$  colonne, si

troverà sempre una funzione booleana delle  $n$  colonne iniziali che per ogni riga dia come risultato esattamente il valore della casella.

Quanto visto finora ci suggerisce che possiamo usare l'algebra di Boole per creare una corrispondenza tra sequenze di 0 e 1 e altre sequenze di 0 e 1, per esempio:



L'obiettivo con cui sto introducendo questi concetti è quello di arrivare a comprendere come la matematica abbia aperto la strada alla realizzazione concreta di un elaboratore elettronico e quindi capire i fondamenti concettuali sui quali gli attuali calcolatori elettronici (non quantistici) sono costruiti. Per raggiungere tale obiettivo mancano ancora due passaggi che vediamo subito.

### Terzo passo: i numeri binari

Sarò breve, meglio, conciso. Consideriamo il numero centoventinove: 129. Esso è composto da tre cifre ordinate, se scrivessi 921 allora non sarebbe il centoventinove. Giusto? Bene, la posizione delle cifre è importante perché indica l'ordine di grandezza che le cifre rappresentano: il 9 in prima posizione ci dice che il nove si riferisce alle *unità*. Il 2 in seconda posizione si riferisce alle *decine*, mentre l'1 in terza posizione alle *centinaia*. Per questo si parla di sistema *decimale* e *posizionale*. Decimale perché ci sono dieci simboli (i.e. 0,1,2,3,4,5,6,7,8,9), posizionale perché la posizione di detti simboli in un numero specifica l'ordine di grandezza che il numero moltiplica. Ora torniamo alle nozioni di algebra viste nel primo capitolo e ricordiamoci che:

- $1 = 10^0$
- $10 = 10^1$
- $100 = 10^2$

- $1000 = 10^3$

- ecc.

quindi, scrivere 129 è come scrivere  $9 \times 10^0 + 2 \times 10^1 + 1 \times 10^2$  cioè scriverlo in potenze di dieci. Bene, posso fare la stessa cosa scrivendo lo stesso numero in potenze di due e avrò:  $129_{10} = 10000001_2$  dove il pedice indica la base in cui è espresso il numero. Sorpreso? Vediamo come ho fatto:

$$129 = 1 \times 2^0 + 0 \times 2^1 + 0 \times 2^2 + 0 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 0 \times 2^6 + 1 \times 2^7$$

. Quindi i numeri che siamo abituati a scrivere in notazione posizionale decimale possono essere scritti anche in formato binario come potenze del due. Ovviamente si noterà un'affinità con quanto visto nel paragrafo precedente. Possiamo sviluppare un'intera algebra basata su due sole cifre e scopriamo che possiamo rappresentare tutti i numeri usando le stesse due cifre. Sembra fatto apposta. Che il formato binario nascondesse in sé qualcosa di interessante se ne erano già accorti i popoli orientali circa tremila anni fa. Per chi non l'avesse già fatto, consiglio la lettura del classico dei mutamenti: I KING.

## Aritmetica dei numeri binari

Proviamo ora a fare un po' di aritmetica usando i numeri binari e cominciamo analizzando l'addizione tra due numeri binari ad una sola cifra. Ricordiamo però che ora siamo tornati nell'algebra ordinaria, quindi, per esempio se in decimale scrivo 1+1 questo fa 2, non 1 come nell'algebra di Boole. Allo stesso modo, se scrivo 1+1 in binario questo fa 10, cioè 0+2 e non 1. Bene, allora cominciamo con l'addizione.

### Somma di due numeri ad una sola cifra

Abbiamo:

$$\begin{aligned}
 1 + 1 &= 10, \\
 1 + 0 &= 1, \\
 0 + 1 &= 1, \\
 0 + 0 &= 0
 \end{aligned} \tag{2.4}$$

Se nelle (2.4) avessi voluto limitare l'analisi della somma alla prima cifra, cioè al coefficiente di  $2^0$  avrei potuto scrivere:  $1+1=0$  unità con resto 1, dove questo 1 è il coefficiente di  $2^1$ ,  $1+0=1$  con resto 0,  $0+1=1$  con resto 0 e  $0+0=0$  con resto 0. Posso anche tabulare queste proprietà della somma come illustrato in tabella VIII.

*Tabella VIII. Addizione binaria ad una cifra.*

<b>x</b>	<b>y</b>	<b>Somma</b>	<b>Resto</b>
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

Far notare l'analogia tra la tabella VIII e le tavole di verità per le funzione booleane (a volte si dice anche funzioni logiche) è quasi offensivo. Se rappresentiamo con  $f(x, y)$  la somma e con  $g(x, y)$  il resto, è immediato trovare:

$$\begin{aligned}
 f(x, y) &= \neg x \cdot y + \neg y \cdot x, \\
 g(x, y) &= x \cdot y
 \end{aligned} \tag{2.5}$$

visto però che stiamo operando anche sull'algebra ordinaria, ci potrebbe essere un po' di confusione riguardo al significato degli operatori  $\cdot$  e  $+$ . Per questo, da qui in poi, torniamo ad usare la forma "logica" degli operatori cioè  $\wedge$ ,  $\vee$  e  $\neg$ , solo che anziché usare questi simboli usiamo i loro "alter ego" (dal latino: altro io) AND, OR e NOT. Tanto per farla facile, quando inizieremo a programmare in C, questi operatori saranno rappresentati in modo ancora diverso: AND: `&&`, OR: `||` e NOT: `!`, coraggio, ci sarà di peggio! Bene, detto questo

riscriviamo le (2.5) come segue:

$$\begin{aligned}f(x, y) &= \text{NOT } x \text{ AND } y + \text{NOT } y \text{ AND } x, \\g(x, y) &= x \text{ AND } y\end{aligned}\quad (2.6)$$

Non è un brutto risultato, con le (2.6) abbiamo stabilito come eseguire una somma tra numeri binari usando delle funzioni logiche (booleane). E se adesso volessimo di più? Se volessimo sommare numeri a due cifre?

### **Somma di due numeri a più cifre**

Ci sono due modi: il primo consiste nello scrivere direttamente la tabella di verità. Siccome questa volta abbiamo due cifre per ogni numero binario, avremo in totale quattro variabili binarie che chiameremo  $x_1, x_2, y_1, y_2$ . Poi avremo tre funzioni delle quattro variabili:  $f(x_1, x_2, y_1, y_2)$  che rappresenta la cifra meno significativa (Less Significant) del risultato e  $g(x_1, x_2, y_1, y_2)$  che rappresenta la cifra più significativa (Most Significant) e infine  $h(x_1, x_2, y_1, y_2)$  che rappresenta il resto. Questo approccio ci porta a scrivere una tabella delle verità con 16 righe, che però sale rapidamente a 64 se vogliamo rappresentare la somma di numeri a tre cifre e a 128 per numeri a 4 per cui è meglio usare un approccio alternativo. In realtà basta usare un po' di buon senso. Prima abbiamo visto che la somma di un numero ad una cifra è data dalle due funzioni viste in (2.6). La somma di un numero a più cifre si ottiene ripetendo per ogni cifra del numero lo stesso algoritmo, quindi date le (2.6) dovremmo essere in grado di esplicitare la somma per un numero composto da cifre arbitrarie. Questo è quasi vero, infatti c'è solo un piccolo particolare di cui tener conto, cioè:

**quando si esegue una somma in colonna, a tutte le colonne tranne che alla prima potrebbe andar sommato il riporto della somma precedente.**

Quindi, quello presentato in (2.6) è l'algoritmo per sommare le prime due cifre ma non ancora quello per sommare due cifre e il riporto.

Rimediamo subito. Se consideriamo anche il riporto ( $r$ ), le variabili indipendenti non sono più due ma tre, quindi abbiamo  $2^3 = 8$  combinazioni come illustrato in tabella VIII bis.

Tabella VIII bis. Addizione binaria ad una cifra.

<b>x</b>	<b>y</b>	<b>r</b>	<b>Somma</b>	<b>Resto</b>
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

Le funzioni logiche Somma e Resto, si possono scrivere più facilmente se prima introduciamo un'altra funzione logica che si usa spesso, si tratta dell' OR esclusivo, anche scritto XOR. Questa funzione è simile agli operatori logici studiati in precedenza, ma in realtà non è una delle operazioni algebriche di base perché è definita attraverso esse come segue:

$$x \text{ } XOR \text{ } y = (x \text{ AND NOT } y) \text{ OR } (y \text{ AND NOT } x) \quad (2.7)$$

Bene, usando la (2.7) possiamo ora dare una espressione semplice per l'algoritmo di somma e resto relativo alla tabella VIII:

$$\begin{aligned} f(x, y) &= (x \text{ } XOR \text{ } y) \text{ } XOR \text{ } r, \\ g(x, y) &= (x \text{ AND } y) \text{ OR } (r \text{ AND } (x \text{ } XOR \text{ } y)) \end{aligned} \quad (2.8)$$

Quindi le due equazioni (2.8) mostrano l'algoritmo per sommare due generiche cifre di un numero binario. Volendo sommare tra loro due numeri binari a  $n$  cifre, sarà sufficiente usare prima le (2.6) per sommare le cifre della prima colonna, poi applicare le (2.8) dove la  $x$  e la  $y$  sono le due cifre in colonna da sommare e  $r$  è il risultato del riporto della somma precedente. Da qui in poi si procede sempre usando le (2.8) fino all'ultima colonna. Andiamo avanti con fede!

## Quarto passo: algebra di Boole e porte logiche

Le operazioni logiche, descritte in termini di espressioni algebriche, sono state anche rappresentate in modo grafico attraverso dei simboli che prendono il nome di *porte logiche*. La visualizzazione delle espressioni algebriche mediante tali simboli semplifica molto il loro uso nella progettazione dei circuiti elettronici, si perché dobbiamo dire, che l'algebra di Boole si è prestata particolarmente bene ad essere concretizzata.

### Introduzione del *bit*

Per capire, proviamo ad immaginare di avere un dispositivo magico, quindi solo immaginario, che esegue le moltiplicazioni degli oggetti. Una scatolina magica con tre porticine, in una inserisco due matite, nella seconda ne inserisco tre e magicamente, nella terza, ne compaiono sei! Fantastico, proverei subito con qualche biglietto da dieci euro! Scherzo.

Scatoline così se ne possono fare davvero, ma non funzionano né con le matite né con i biglietti da dieci, funzionano solo con i *bit*, cioè con i segnali elettrici che si usano per rappresentare le informazioni. Facciamo un passo per volta. Abbiamo detto che  $x \wedge y$  è una espressione logica che può essere rappresentata anche dall'espressione algebrica  $x \cdot y$ . Possiamo poi assegnare il risultato dell'espressione ad una terza variabile, quindi scrivere  $z = x \cdot y$  e possiamo anche dire che  $z$  è una funzione di  $x$  e  $y$ , cioè  $z = f(x, y)$ , ricordiamoci bene questo passaggio perché presto diventerà importante. Bene, queste sono solo formule. Affinché dette operazioni abbiano davvero luogo, dobbiamo agire con carta e penna e cervello, altrimenti le operazioni rimangono lì dove sono state scritte. Se ad esempio poniamo  $x = 1$  e  $y = 1$  avremo  $z = 1 \cdot 1 = 1$ , ma dobbiamo eseguire il calcolo e poi scriverlo.

### Le porte logiche

Esistono dei componenti elettronici capaci di eseguire alcune

operazioni elementari e che possono essere combinati insieme in modo arbitrariamente complesso. Il primo componente che vediamo si chiama *AND gate*, in italiano porta AND, che si indica con il simbolo seguente:

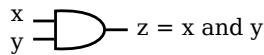
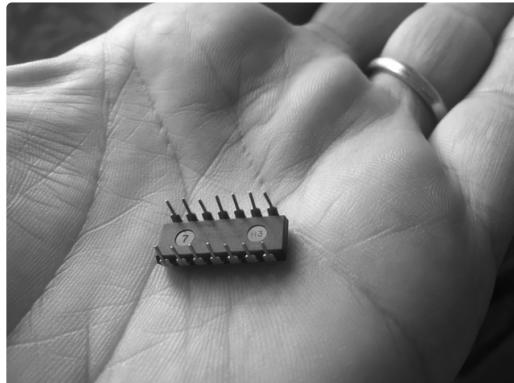


Fig. 2.1 - Simbolo usato per la porta AND.

che rappresenta appunto una porta AND (vedi Fig. 2.1). Come si vede dalla figura esso ha tre terminali, cioè fili o piedini metallici. Detti terminali sono conduttori elettrici. Se ad entrambi i terminali  $x$  e  $y$  si applica una tensione elettrica allora la stessa tensione sarà presente anche al terminale  $z$ , se invece anche ad uno solo dei terminali la tensione applicata è nulla, anche la tensione al terminale  $z$  risulterà nulla. Come ci si può rendere conto, questo componente realizza esattamente l'operazione algebrica  $z = x \cdot y$  dove si intenda:

- Tensione elettrica = 1
- Assenza di tensione = 0

Per essere rigorosi, si deve far presente che esistono dei precisi valori in Volt (V) che definiscono i valori di soglia della tensione elettrica associati agli stati logici 0 ed 1. Chi di elettronica già ne mastica capirà il perché tendo ad evitare i dettagli tecnici in questo contesto, per chi invece incontrasse per la prima volta questi argomenti, quanto detto sarà sufficiente per il proseguo del ragionamento e se sarà interessato potrà trovare tantissimo materiale in rete sulle tecnologie per la realizzazione dei *chip* che sono i componenti elettronici, materiali e tangibili, che realizzano le porte logiche in questione. Un esempio di chip è riportato in figura 2.2.

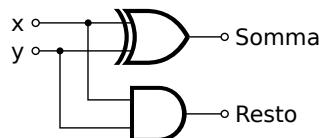


*Fig. 2.2 - La mia mano regge un tipico componente elettronico in cui sono presenti quattro porte AND.*

Dicevamo che la cosa davvero interessante è che tali porte possono essere connesse tra di loro, cioè si può portare l'uscita (che abbiamo indicato con  $z$ ) di una porta nell'ingresso di un'altra (per esempio alla  $x$ ) per formare circuiti logici più complessi, chiamate *reti logiche*.

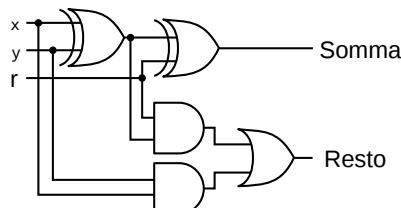
### Rete logica per i circuiti Half e Full Adder

Le funzioni presentate nelle (2.6) e nelle (2.8) possono essere rappresentate ora usando le porte logiche. Per la somma della prima cifra binaria (Eq. 2.6) abbiamo:



*Fig. 2.3 - Circuito Half Adder realizzato con una porta AND e una XOR.*

mentre per la somma delle restanti cifre (Eq. 2.8) si ha:

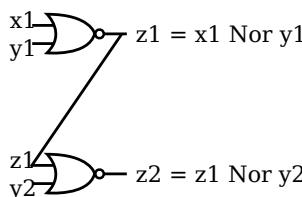


*Fig. 2.4 - Circuito Full Adder realizzato con porte AND e XOR.*

La tecnologia del silicio permette di *integrare* milioni di elementi circuitali, come quelli appena visti, in spazi ridottissimi. Per questo si può ben immaginare che collegare assieme un circuito Half Adder a sette circuiti Full Adder per ottenere un circuito che calcola la somma di numeri binari a otto cifre sia tecnologicamente un compito semplice. La somma, comunque, non è l'unica applicazione dell'algebra di Boole tra numeri binari, infatti sono di immediata realizzazione anche la differenza (si ottiene dalla somma attraverso il complemento a due che non trattiamo in questo contesto), il prodotto, la divisione e il confronto (maggiore o minore). Sono poi di immediata realizzazione anche le operazioni logiche AND, OR e NOT. Quando si integrano in un solo circuito tutti gli elementi per realizzare le suddette operazioni si parla di Unità Aritmetico Logica o ALU che come vedremo è un elemento della Central Processing Unit o CPU. Quindi abbiamo visto che con l'elettronica è possibile realizzare dei circuiti che operano sulle quantità elettriche (tensioni, correnti o anche cariche) come se fossero dei numeri, quello che dobbiamo vedere ora è dove memorizzare i numeri (sequenze binarie di 1 e 0) per permettere alla ALU di leggerli e dove poter scrivere i risultati delle operazioni.

### **Memorie e registri**

In questo paragrafo vedremo come si possano ottenere dei circuiti che possono memorizzare un valore logico o binario. Cominciamo analizzando lo schema riportato nella figura qui sotto che mostra come si possono collegare due porte NOR.



*Fig. 2.5 - Connessione tra due porte Nor.*

In figura 2.5 si è connessa l'uscita della prima porta NOR ad una delle entrate della seconda porta NOR. Come si può dedurre, una volta connesse le due porte, l'uscita della seconda porta (la  $z_2$ ) è funzione della variabile  $y_2$  e della' uscita della prima porta  $z_1$ . Voglio sottolineare che quando dico "connettiamo" intendo una connessione materiale, cioè si crea un collegamento elettrico tra i terminali delle porte, per esempio con una stagnatura. Ora connettiamo l'uscita della seconda porta ( $z_2$ ) all'ingresso  $y_1$ , come indicato nella figura seguente:

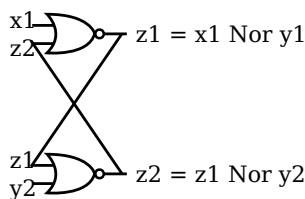


Fig. 2.6 - Mutua connessione tra due porte Nor.

Quello che si è ottenuto, mostrato schematicamente in figura 2.6, è un circuito detto *flip-flop*.

### Circuito flip-flop

Questo tipo di circuito ha molte applicazioni, ma una in particolare gioca il ruolo chiave nella comprensione di come si può passare dalla macchina di Turing, intesa come congegno meccanico, ad un moderno calcolatore elettronico, che ricordiamo ha gli stessi limiti concettuali di una macchina di Turing, cioè è in grado di computare gli stessi algoritmi, ma ha capacità di memoria e velocità di calcolo imparagonabili a qualsiasi macchina meccanica. Torniamo al flip-flop. La caratteristica di questo circuito elettronico è che può immagazzinare un'informazione di un bit, cioè uno 0 o un 1. Il circuito mantiene il valore memorizzato anche se ai suoi ingressi non viene più portato alcun segnale (ovviamente il circuito deve essere alimentato, cioè consuma energia elettrica!). Quindi basta immaginare di posizionare otto circuiti, analoghi a quello illustrato,

uno accanto all'altro, che sarà possibile memorizzare una "parola" composta da otto bit, quello che si chiama *byte*. Il circuito flip-flop fa parte dell'elettronica digitale (cioè numerica) detta *sequenziale*.

## Circuiti elettronici

I circuiti elettronici digitali si dividono in *combinatori* e, sequenziali. L'output di un circuito combinatorio si può calcolare immediatamente quando si conoscono i valori di input. I circuiti sequenziali invece sono diversi, lo si vede già se si cerca di capire come si comporta il flip-flop: proviamo ad analizzarlo insieme. Se poniamo:  $x_1 = 0$  e  $y_2 = 1$  abbiamo che l'uscita (output)  $z_1$  assume il valore 1. Se ora poniamo entrambi  $x_1$  e  $y_2$  a 0, l'uscita mantiene il valore 1, se da qui passiamo a  $x_1 = 1$  e  $y_2 = 0$ , l'uscita commuta a 0 e mantiene questo valore anche se di nuovo poniamo  $x_1$  e  $y_2$  a 0. Si ha che il valore di  $z_1$  dipende **non** solo dalla configurazione di ingresso (cioè l'insieme dei valori di input), ma anche dal valore precedentemente assunto dalla  $z_1$  stessa, in altre parole il circuito mantiene memoria delle configurazioni passate. In particolare, il flip-flop mantiene memoria di un solo passo!

## I registri e la memoria

Dopo la ALU, i flip-flop sono gli elementi base di un altro componente della CPU, i *registri*. Questi sono elementi capaci di memorizzare parole composte da 8, 16, 32 o 64 bit (il limite è solo tecnologico o economico). La CPU usa i registri come un "blocco appunti" per elaborare i dati che attinge dalla *memoria*. La memoria è un insieme molto vasto di byte ognuno dei quali ha un proprio *indirizzo* costituito da un numero progressivo. La memoria è un circuito elettronico digitale (quindi costruito sempre con porte logiche) esterno alla CPU. La memoria espone una propria interfaccia di connessione: una serie di contatti elettrici a cui va portato un segnale logico di tipo 0 o 1. La memoria interpreta i valori presentati alla propria interfaccia come numeri binari, compone l'indirizzo voluto e restituisce attraverso un'analogia interfaccia di *output* il valore memorizzato nel byte corrispondente all'indirizzo richiesto. La CPU memorizza i dati inviati

dalla memoria all'interno di uno dei propri registri, sul quale poi potrà operare le computazioni del caso. Ci si riferisce all'idea di separare la memoria dalla CPU con il termine *architettura di von Neumann* della quale riporto uno schema qui sotto.

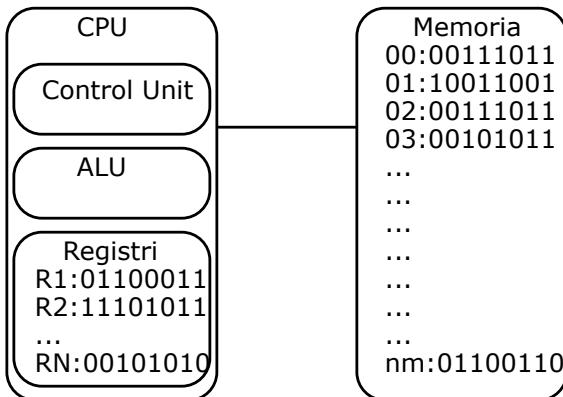


Fig. 2.7 - Architettura di von Neumann.

## Confronto con le reti neurali del cervello

Dopo aver introdotto il concetto di memoria, cioè di celle individuate da uno specifico indirizzo, è importante soffermarsi ad analizzare la profonda differenza che si ha tra il calcolatore, che appunto usa questo tipo di memoria, e il cervello la cui memoria è il risultato delle connessioni tra semplici cellule nervose, ognuna in sé eliminabile o sostituibile.

Fin'ora il concetto di interconnessione tra neuroni è stato solo accennato, ma quello che si è visto è sufficiente per apprezzare la differenza tra il funzionamento della memoria *biologica* e quella elettronica (meccanica). Questo è uno dei motivi per cui, se si vuole davvero apprezzare e capire come funzionano le reti neurali artificiali, è importante partire dall'hardware e da linguaggi molto vicini alla macchina, piuttosto che usare subito delle tecnologie che mascherano il funzionamento a basso livello del computer.

### L'unità di controllo CU

L'obiettivo di questo testo è di trattare le reti neurali. L'Excursus che

stiamo compiendo in questo campo dell'elettronica ci serve per capire come una macchina possa eseguire un processo "analogo" al processo che avviene in una rete neurale. Volendo sintetizzare, vogliamo vedere come un processo biologico possa essere astratto in forma algoritmica e come lo stesso algoritmo possa essere riprodotto da una macchina elettronica. Finora abbiamo visto come un dispositivo elettronico possa compiere delle operazioni logico-matematiche su dei numeri memorizzati all'interno di memorie binarie, quello che manca è vedere come tali operazioni logico-matematiche possano essere comandate mediante istruzioni memorizzate all'interno della stessa memoria.

Per comprendere bene la problematica che stiamo trattando proviamo ad immaginarci di voler ottenere (presto diremo programmare) che una macchina elettronica esegua un semplice algoritmo descritto dai passaggi che seguono:

1. Somma 1+2;
2. Prendi il risultato della somma al punto 1 e moltiplicalo per 2;
3. Scrivi nella memoria il risultato finale;

In linea di principio abbiamo già visto che tutte queste operazioni sono eseguibili da una macchina elettronica, ma se ci pensiamo bene in questo momento non abbiamo ancora sviluppato delle idee su come programmare la macchina per eseguirle. Un modo diretto sarebbe quello di impostare i flip-flop dei registri interni alla CPU in modo che contengano i due operandi 1 e 2, poi collegare tali registri ai piedini di entrata della ALU in modo che essa vi trovi le configurazioni binarie 00000001 e 00000010 in input ed esegua direttamente la somma. Se seguissimo questa procedura riusciremmo intanto a compiere il primo passo dell'algoritmo però ancora non sapremmo come andare al secondo. Intanto va detto che l'idea di codificare delle istruzioni agendo direttamente sui collegamenti elettrici dei circuiti integrati non è per niente "pellegrina" anzi, è usatissima tutt'ora. Quando si hanno delle

istruzioni codificate direttamente a livello hardware ci si riferisce ad esse come *firmware* anziché *software* e queste risiedono in memorie dette Memorie di Sola Lettura o in inglese ROM. A volte il firmware viene usato anche per scrivere piccoli programmi applicativi che devono avere prestazioni altissime.

## Idea di istruzione e codifica

Quindi da un problema siamo passati a due: abbiamo capito che dobbiamo trovare un modo di codificare e memorizzare le istruzioni che vogliamo vengano eseguite, poi dobbiamo trovare un modo perché le istruzioni vengano eseguite in successione una dopo l'altra.

Il primo problema lo possiamo risolvere così:

1. Assegnamo un codice numerico alle operazioni che vogliamo compiere.
2. Scriviamo nella memoria, non nei registri (vedi Fig. 2.7), la lista delle istruzioni con indirizzi crescenti.

Questa è una buona strategia che sta rendendo il nostro obiettivo da pura fantasia a qualcosa di attuabile, però ci sono ancora degli aspetti da sviscerare:

1. In che modo i codici delle istruzioni istruiscono la CPU sull'operazione da svolgere?
2. Con quale meccanismo la CPU sceglie l'istruzione da eseguire?  
Come fa a sapere "a chi tocca"?
3. Come facciamo a scrivere i codici delle istruzioni nella memoria?

Cominciamo dal primo punto. Assegnamo arbitrariamente il codice binario 00000001 alla istruzione di somma che chiamiamo ADD. Da qui in avanti faremo riferimento ai codici numerici delle istruzioni con il termine *opcode* che sta per operation-code. Ora dobbiamo fare in modo che al presentarsi dell'opcode 00000001 alla CPU, essa esegua un' addizione. Un' addizione tra chi? Ovviamente alla nostra istruzione manca qualcosa: dobbiamo specificare gli operandi. Potremmo pensare di costruire l'istruzione memorizzando oltre al

codice anche gli operandi ottenendo qualcosa come:

ADD 1,2 → 00000001 00000001 00000010

Per memorizzare una istruzione così però non è sufficiente un solo byte, ne servono tre. Siccome siamo previdenti possiamo pensare che quattro byte sia meglio: ci potrebbero venire in mente delle nuove istruzioni che prevedono più operandi! Possiamo quindi definire una specie di standard e stabilire che le istruzioni occupino sempre quattro byte. In questo caso, tra una istruzione e l'altra l'indirizzo di memoria varia di 4 e non di 1, quindi tanto vale rappresentare la memoria in modo che questo standard sia manifesto.

### Codifica esadecimale

Ecco, prima di proseguire è il momento di introdurre un nuovo modo di codificare i numeri, si chiama esadecimale e spesso si indica con EX. Quando siamo passati dai numeri decimali ai binari abbiamo ridotto da 10 a 2 il numero di cifre, ora facciamo il contrario e le aumentiamo, stabiliamo l'esistenza di 16 cifre. Le prime dieci le conosciamo: 0,1,2,3,4,5,6,7,8,9, le successive sei le aggiungiamo prendendo in prestito i simboli dall'alfabeto latino maiuscolo: A,B,C,D,E,F. Cosa significa quindi  $(1)_{16}$ ? Niente di nuovo, significa  $1 \times 16^0 = 1$ . Cosa significa invece la scrittura  $(10)_{16}$ ? Abbiamo:  $(10)_{16} = 0 \times 16^0 + 1 \times 16^1 = 16$ , così il numero AB corrisponde a:  $11 \times 16^0 + 10 \times 16^1 = 171$ .

### Istruzioni in memoria

Abbiamo introdotto la codifica esadecimale perché è quella che si usa per indicare gli indirizzi in memoria, ora possiamo passare a rappresentare la memoria con le istruzioni che ci scriveremo dentro.

Per convenzione la memoria si rappresenta come una tabella in cui ogni riga rappresenta una parola di quattro byte, e gli indirizzi sono crescenti dal basso verso l'alto:

0	00000001	00000001	00000010	-
<b>Indirizzo</b>	<b>Byte 0</b>	<b>Byte 1</b>	<b>Byte 2</b>	<b>Byte 3</b>

Abbiamo quindi scritto nella memoria la prima istruzione, scriviamo

anche le altre poi ci occuperemo dei punti che abbiamo sollevato. Il secondo passo dell'algoritmo scritto sopra consiste nel moltiplicare per 2 il risultato dell'addizione appena eseguita. A questo punto entrano in gioco i registri. Dobbiamo stabilire che la CPU scriva il risultato dell'addizione calcolata dalla ALU in uno dei registri interni. Stabiliamo che il registro R1 (vedi figura 2.5) sia usato proprio per questo, cioè per memorizzare i risultati delle operazioni aritmetiche svolte dalla ALU. Quindi nella prossima istruzione dovremo comunicare alla CPU 1) il tipo di operazione che vogliamo compiere (moltiplicazione), 2) il primo operando (memorizzato nel registro R1), 3) il secondo operando (il valore 2).

Quando in una istruzione si passa un valore numerico, per esempio nella ADD vista sopra abbiamo passato i valori 1 e 2, si dice che questi valori sono passati in modo immediato. Un'alternativa è passare gli stessi valori specificando anziché il valore stesso, l'indirizzo di memoria in cui essi sono memorizzati. Ovviamente, in base alla modalità con cui si passano i valori, cambia l'opcode dell'operazione, anche se si tratta della stessa operazione aritmetica o logica. Detto questo, stabiliamo che l'opcode per la moltiplicazione tra un valore presente in un registro ed un valore immediato sia il seguente: 00000010 e stabiliamo inoltre che la CPU faccia riferimento al registro R1 con il codice 00000001, allora l'istruzione per eseguire il punto due è la seguente:

MUL R1,2 → 00000010 00000001 00000010

che andremo a scrivere in memoria nell'indirizzo successivo alla prima:

04	00000010	00000001	00000010	-
00	00000001	00000001	00000010	-
Indirizzo	Byte 0	Byte 1	Byte 2	Byte 3

Il terzo passo è quello di prendere il risultato della moltiplicazione, che si trova in R1, e scriverlo nella memoria, in un dato indirizzo. Ora su quest'ultimo punto c'è un po' da ragionare. Abbiamo visto che la memoria serve per scriverci dentro le istruzioni da far eseguire alla CPU. Se lasciamo che la CPU la usi per scrivere i risultati delle operazioni che sta svolgendo, questi potrebbero andare sovrascritti alle istruzioni stesse, generando confusione.

### Suddivisione logica della memoria

Stabiliamo allora che la memoria sia divisa in due aree, un'area in cui è presente il *codice*, cioè la codifica in opcode e operandi del programma e un'area riservata ai dati, cioè ai valori che la CPU può scrivere e leggere senza che tali operazioni modifichino la *procedura*. In questo contesto stabiliamo in modo molto semplice che gli indirizzi di memoria riservati al codice vadano da 0 a 7F mentre quelli da 80 a FF siano riservati ai dati. Per la CPU un'operazione di scrittura corrisponde a spostare o copiare un dato da un indirizzo all'altro. L'istruzione per tale operazione si indica con MOV che sta per "move", cioè muovere. Se stabiliamo di voler scrivere il risultato nel primo indirizzo dell'area dati, quindi 80, l'istruzione da memorizzare è quindi:

MOV R1,128 → 00000011 00000001 10000000

dove si è stabilito che l'opcode di MOV è 00000011. Dopo aver scritto anche questa istruzione lo stato della memoria sarà il seguente:

08	00000011	00000001	10000000	-
04	00000010	00000001	00000010	-
00	00000001	00000001	00000010	-
Indirizzo	Byte 0	Byte 1	Byte 2	Byte 3

Fin qui abbiamo visto cosa scrivere nella memoria per ottenere l'esecuzione dell'algoritmo voluto. D'ora in avanti chiameremo la sequenza delle istruzioni presenti in memoria *programma*.

## **Pesca, decodifica ed esegui!**

Vediamo ora come ottenere che le istruzioni siano presentate alla CPU nella sequenza con cui sono state scritte. Anzi tutto stabiliamo che nella CPU sia presente un registro di 32 bit, quindi di 4 byte, che chiameremo IR (registro delle istruzioni) in cui verranno copiate le istruzioni presenti nella memoria, in questo modo la CPU si troverà sempre l'istruzione da decodificare ed eseguire pronta "per l'uso". Vediamo anzitutto come funziona il meccanismo che permette di copiare in sequenza le istruzioni dalla memoria all'IR. Introduciamo un altro registro che chiamiamo PC (contatore del programma o program counter) dove sarà scritto l'indirizzo in memoria della prossima istruzione che deve essere copiata all'interno dell'IR. Quando si avvia un programma, nell'PC viene scritto l'indirizzo 00 (questo è solo per semplificare) in modo che la *logica cablata*, che controlla il computer, acceda all'indirizzo iniziale del programma e copi la prima istruzione dalla memoria all'IR. (Nota: La logica cablata è il corrispondente degli ingranaggi della Macchina di Turing usati per far muovere il nastro e scriverci sopra.) Questa fase si chiama fase di fetch, come precedentemente anticipato. Fatto questo, ad opera della logica cablata, il contenuto del PC viene incrementato di 4, in questo modo, al prossimo ciclo, sarà "pescata" la seconda istruzione.

Dopo il "fetch" l'istruzione si trova nel registro IR che è direttamente collegato all'unità di controllo (CU, vedi figura 2.5) ed ha inizio la fase di decodifica e di esecuzione. In questa fase entra in gioco la logica cablata nella CU che ha il compito di produrre i segnali logici verso la ALU affinché essa svolga le operazioni codificate nelle istruzioni. Per approfondimenti su questo tema rimando ai due testi di Bucci e Zhirkov elencati nella bibliografia.

## **Memorizzare le istruzioni**

Dovrebbe esser chiaro che il problema di scrivere nella memoria le

istruzioni del programma è riconducibile al problema tecnico di impostare le celle di memoria (bit) della memoria ad 1 o 0 in base alle sequenza di bit prevista dalle istruzioni e dagli operandi stessi. In linea di principio si potrebbe anche pensare di disporre degli interruttori manuali per settare la memoria bit a bit, e in effetti agli albori della tecnologia informatica così è stato. Il problema di questa soluzione è che è altamente soggetta ad errori umani, infatti è molto facile scambiare un 1 con uno 0 inserendo centinaia di sequenze di 32 bit. In generale nei dispositivi commerciali in vendita oggi non è possibile accedere alla memoria direttamente per settare i valori dei bit a 1 o a 0, perciò questo avviene attraverso delle periferiche, cioè dispositivi elettronici esterni alla coppia CPU/memoria come le tastiere.

## **Input e output**

Senza pretesa di completezza cerchiamo di inquadrare il problema tecnico di portare dati dall'esterno (periferiche) verso la memoria e al contrario dalla memoria verso l'esterno. Anzi tutto dobbiamo pensare che per trasferire i dati è necessario prima generarli, per esempio dal dispositivo di input, poi disporre di un collegamento elettrico tra il dispositivo di input e la memoria. Quindi abbiamo due cose da fare:

1. Generare i dati.
2. Trasferirli.

## **Dispositivi a carattere**

Per esemplificare, prendiamo un dispositivo semplice come la tastiera meccanica di un personal computer, poi vedremo che lo stesso ragionamento si applica ad uno schermo touch screen. Essa è definita un *dispositivo a carattere* come il terminale, di cui tratteremo più avanti nella sezione dedicata alle reti neurali. La tastiera ha un compito abbastanza limitato, deve produrre un output digitale in base al carattere che viene premuto. Ogni carattere presente nella tastiera

ha una codifica numerica standard. Per esempio il carattere A corrisponde al codice decimale 65 dalla tabella ASCII (o 41 in esadecimale). Quindi se premo A sulla tastiera il numero 65 deve essere trasferito da essa alla memoria. Anzi tutto il numero dovrà essere generato, e questo significa che la tastiera dovrà avere una *porta* con almeno 8 collegamenti elettrici per presentare la configurazione 01000001 in termini di valori di tensione alti o bassi. Per farla semplice supponiamo che detti collegamenti siano forniti dal ben noto connettore USB.

## **BUS e USB, parentele?**

Abbiamo capito una cosa: la tastiera è un dispositivo elettronico che alla pressione di un tasto produce una configurazione di tensioni elettriche (alte o basse) applicate ad un insieme di contatti elettrici come quelli del connettore USB. Come si imposti la corretta configurazione di tensioni in base al tasto premuto è un problema squisitamente elettrotecnico che qui non trattiamo. Una volta che la A è presente sul connettore USB si tratta di portarla fino alla memoria. Per far questo è necessario che il connettore USB sia in collegamento elettrico con la memoria e questo lo otteniamo collegando il connettore alla porta USB che a sua volta è collegata ad un nastro di cavi che raggiungono sia la memoria che la CPU (vedi figura 2.8). Tale nastro si chiama BUS, in particolare BUS dei dati, non a caso, USB è l'acronimo di Universal Serial Bus. Dal momento che la memoria è collegata a questo BUS essa potrà attingere dati direttamente da esso, e va da sé che lo stesso ragionamento può essere applicato a qualsiasi altro dispositivo di input, come una webcam o altro. A questo punto però dovrebbe sorgere una domanda: a quale indirizzo di memoria va trasferito un dato presente sul BUS? Bene, questa è un'ottima domanda che merita una risposta. La memoria non ha autonomia in questa decisione: è infatti la CPU che ha il controllo attraverso il programma in esecuzione, quindi dalla CPU arriverà l'ordine di trasferire o meno e a quale indirizzo un dato presentato sul BUS. Quando dico che il compito spetta alla CPU

intendo dire che è il programmatore che dovrà scrivere le giuste istruzioni perché il dato presentato sul BUS finisce nel giusto indirizzo di memoria. Il compito non è semplice, ma vedremo almeno di delineare gli aspetti principali.

## **Iterare per riuscire**

Supponiamo di star scrivendo un programma semplicissimo il cui scopo è quello di leggere i dati dalla tastiera e trasferirli in sequenza alla memoria. Il programma avrà la responsabilità di stabilire qual è il primo indirizzo in cui si inizierà a memorizzare l'input della tastiera, poi, per ogni nuovo codice ASCII inviato dalla tastiera dovrà calcolare il nuovo indirizzo. Rispetto al semplice algoritmo visto prima, questo programma prevede di ripetere il *flusso* delle istruzioni più volte, generando quella che viene detta *iterazione*, quindi un ciclo di azioni sempre uguali. Per fare questo, come vedremo nei prossimi paragrafi, le CPU hanno una tecnica che si basa sul registro PC, praticamente dopo aver eseguito una sequenza di istruzioni, se vogliamo ripeterle dobbiamo scrivere nel PC non l'indirizzo della memoria dove ci sarebbe la prossima istruzione, cioè PC+4, ma dobbiamo scrivere l'indirizzo dell'istruzione da cui vogliamo ripartire. Poi il programma dovrà stabilire in quale indirizzo della memoria spostare il dato presente nel BUS. Per fare questo dovrà scrivere o in memoria o su un registro l'ultimo indirizzo in cui ha copiato il dato del BUS e, ad ogni ciclo di lettura aggiornarlo aumentandolo di 1, in modo che tutti i codici ASCII dei tasti premuti dalla tastiera siano correttamente memorizzati e non sovrascritti gli uni sugli altri. Ci sono tante difficoltà tecniche in questo, che come dicevo sono bene affrontate nei testi che trovate in bibliografia.

## **BUS degli indirizzi**

Non abbiamo ancora visto come sull'indirizzo di memoria in cui il programma ha deciso di memorizzare il dato del BUS venga oggettivamente copiato il dato. Questo avviene per mezzo di un altro BUS, detto appunto BUS degli indirizzi. Il programma che stiamo ipoteticamente scrivendo dovrà presentare sul BUS degli indirizzi

l'indirizzo di memoria in cui si vuole sia copiato il dato presente sul BUS dei dati. Come fa? Semplice, nella CPU c'è un registro specifico che si chiama MAR (registro di accesso alla memoria) che è collegato al BUS indirizzi. Quello che scriviamo lì, è l'indirizzo che viene presentato sul BUS. Anche la memoria è collegata a quel BUS, quindi il compito del programma è solo quello di scrivere sul MAR l'indirizzo calcolato ad ogni ciclo, poi di dare alla memoria il comando di lettura. La stessa analisi vale per uno schermo touch. Generalmente un touch screen non è parte integrante di un'architettura, ma è visto come una periferica anche se non USB. Sugli smartphone, normalmente, tutte le periferiche sono già collegate al chipset insieme alla CPU e si scambiano dati mediante un BUS interno, quindi, sebbene cambi il tipo di dispositivo, la problematica di portare i dati acquisiti da una tastiera, meccanica o touch, alla memoria rimane la stessa.

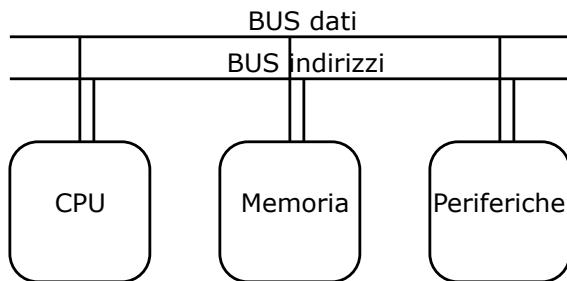


Fig. 2.8 - Collegamento a BUS tra memoria, CPU e periferiche.

## 2.2 Cenni sul linguaggio macchina e sul linguaggio Assembly

Assembler o assembly? Il termine corretto in effetti è assembly, ma lo abbiamo chiamato tutti assembler per tanti anni e mi sono affezionato a questo nome. Il linguaggio assembly è il parente più vicino del linguaggio macchina, cioè della programmazione basata solo su 1 e 0. Come sottolinea Bucci nel suo testo (vedi bibliografia), non si tratta propriamente di un linguaggio ma più di un sistema per

scrivere programmi in linguaggio macchina evitando alcuni aspetti noiosi, come doversi ricordare a memoria il codice binario di ogni istruzione. Il motivo per cui non si può parlare propriamente di un linguaggio quando ci si riferisce all'assembler è perché la caratteristica principale che deve avere un linguaggio di programmazione è quella di "nascondere" al programmatore i dettagli architettonici della macchina su cui sta programmando. Solo per fare un esempio, le architetture dei processori si dividono storicamente in RISC e CISC che sono due acronimi. Il primo si riferisce a CPU che si basano su alcuni principi progettuali quali:

1. Tutte le istruzioni (macchina) devono avere le stesse dimensioni fisse (per esempio 4 byte).
2. Il campo opcode ha una dimensione predefinita (per esempio 1 byte).
3. I formati delle istruzioni devono essere in numero molto limitato (per esempio la ADD deve limitarsi a sommare addendi caricati in due registri e non prevedere anche la somma di valori immediati).

In questo modo le architetture RISC riescono a ridurre "all'osso" il loro repertorio delle istruzioni ed avere certi vantaggi. Al contrario le architetture CISC "rinnegano" detti principi e hanno altri vantaggi. Noi non vogliamo discutere le scelte architettoniche, sottolineiamo invece che se un programmatore deve eseguire una somma tra 1 e 2 in assembly, dovrà sapere se l'operazione ADD sulla architettura che ha scelto, gli permette di scrivere:

ADD R3,1,2 ;

oppure se deve prima caricare gli operandi su due registri e poi eseguire la somma:

MOV R1,1 ;

MOV R2,2 ;

ADD R3,R1,R2 ;

## **Assembly e architettura hardware**

Quindi, l'assembly è uno specchio dell'architettura su cui si sta lavorando. Se si vuole scrivere un programma che sfrutti a pieno l'architettura che su cui si sta lavorando, l'assembler è spesso una buona soluzione. Questo argomento è particolarmente importante oggi che si vogliono utilizzare al massimo le potenzialità dell'hardware per eseguire i calcoli paralleli degli algoritmi di machine learning. Uno dei motivi per cui dal 2012 le reti neurali sono tornate in voga dopo le precedenti "bolle" è stata la disponibilità di PC con GPU a basso costo. L'uso delle GPU permette di accelerare le computazioni algebriche che abbiamo visto nel primo capitolo, ma come si chiede ad una macchina di usare la GPU anziché la CPU? Ovviamente è necessario conoscerne a fondo le caratteristiche dell'hardware (quindi sapere che quella macchina dispone di GPU) e il repertorio delle istruzioni per poterle usare.

Tutto questo per dire che una volta che si è scritto un programma in assembler, pensando ad una certa architettura (per esempio l'AMD64) non si può sperare di prendere lo stesso codice e *assemblarlo* per un'altra architettura. Certo, la logica con cui si programma a basso livello (cioè assembler) rimane circa la stessa su ogni macchina, ma ogni architettura ha le proprie peculiarità. Per questo motivo sono stati introdotti i linguaggi come il C che invece permettono di scrivere il codice di un programma senza preoccuparsi dell'architettura sottostante. Ad esempio, le istruzioni per eseguire la somma di prima, potrebbero essere le seguenti:

```
int a=1;  
int b=2;  
int c;  
c=a+b;
```

su tutte le possibili architetture. Ovviamente questo è possibile perché c'è uno "strato" software che si occupa di trasformare il codice C nelle opportune istruzioni assembler in base all'architettura sottostante. Questo strato si chiama *compilatore* e lo vederemo un po' più avanti quando inizieremo l'analisi del linguaggio C.

## Codifica delle istruzioni

La cosa migliore per prendere confidenza con l'assembly e con l'architettura è provare a scrivere un programma. In questo paragrafo vediamo di scrivere insieme un semplice programma e di assemblarlo. Per seguire completamente bene l'esempio vi invito a lavorare con il seguente assetto:

- Architettura Intel®64 o AMD64.
- Sistema operativo GNU/Linux per x86\_64, anche il Mac OSX dovrebbe andare bene, per Windows leggi sotto.
- Assemblatore NASM.
- Compilatore GCC.
- Editor di testo EMACS o uno a piacere.

Visto la rapidità con cui evolve l'informazione sul Web, preferisco non sbilanciarmi nello scrivere che potete trovare Linux per Windows a questo indirizzo: [https://docs.microsoft.com/en-us/windows/wsl/install\\_guide](https://docs.microsoft.com/en-us/windows/wsl/install_guide) o che per installare nasm basta digitare \$sudo apt-get install nasm al prompt dei comandi, oppure ancora che il gcc si installa come: \$ sudo apt install gcc, siete baldi e giovani, datevi da fare!

## Programmi applicativi e funzioni di sistema

Bene, possiamo scrivere il primo programma, che nella tradizione dei buoni programmatore è sempre un Hello World!. Prima di procedere devo fare un'avvertenza. I programmi che scriveremo qui, sia quelli di esempio in assembler che i modelli di reti neurali in C, sono programmi applicativi, cioè non di sistema. I programmi di sistema sono parte del sistema operativo, oppure i così detti *driver* e sono gli unici autorizzati ad accedere direttamente alle risorse hardware del computer. Per questo motivo i programmi che scriveremo in questi esempi non possono accedere alle risorse del computer direttamente, quindi ad esempio non possono leggere direttamente la porta USB o mandare un output al video. Per farlo usano un intermediario, cioè una chiamata ad una *funzione di sistema*, cioè una procedura che ha

scritto Linus Torvalds per chi usa Linux, Bill Gates per chi usa Windows o ancora Steve Wozniak per chi usa il Mac. Se si desidera scrivere tutto il codice, anche quello che accede all'hardware ci sono due strade, una è quella di incorporare il proprio codice nel kernel, l'altra è quella di rinunciare al sistema operativo e scrivere un programma *stand alone* che giri sul computer senza sistema operativo. Per farlo si deve scrivere anche un proprio boot loader. Si può fare, si fa, ma è oltre gli obiettivi di questo libro.

### **Il primo programma: Ciao Mondo!**

Questo forse non è il programma più istruttivo con cui cominciare, ma è una tradizione ed è sempre meglio rispettare le tradizioni. Lo scopo del programma è quello di scrivere Ciao Mondo! sul monitor. Anzi tutto dobbiamo memorizzare la scritta 'Ciao Mondo!' nella memoria, questo lo facciamo come segue:

```
section .data  
messaggio: db 'Ciao Mondo!'
```

L'istruzione section avverte l'assemblatore (nasm, in questo caso) che i dati inseriti devono essere scritti in memoria in una sezione dedicata che stiamo chiamando appunto .data.

### **sezioni .data e .text**

Le informazioni scritte in detta sezione verranno tenute separate da quelle inserite in memoria nella sezione .text dove invece verranno scritte le istruzioni del programma. Abbiamo già affrontato questo argomento qualche paragrafo fa quando, semplificando, abbiamo ipotizzato che dall'indirizzo 0 al 7F fossero memorizzate le istruzioni e dal 80 al FF i dati. Qui vediamo che questa divisione logica (e non fisica) della memoria la otteniamo specificando, prima di inserire un dato o un'istruzione, la sezione .data o .text. Con divisione logica si intende dire che non c'è nessuna barriera reale che separa la memoria dove sono scritti i dati da quella dove sono scritte le istruzioni, è una separazione che otteniamo facendo attenzione a condurre il flusso di esecuzione del programma (quindi gli indirizzi scritti nel PC) solo attraverso gli indirizzi dove sono memorizzate le

istruzioni e non i dati. Il modo più semplice di farlo e di non "mischiare" dati e istruzioni.

L'istruzione *messaggio*: è solo una *label*, un' etichetta che si usa per non dover inserire un indirizzo numerico di memoria. In pratica quello che stiamo facendo è di indicare un indirizzo nella sezione di memoria .data e specificare che da lì in avanti saranno occupati 11 byte con i caratteri ASCII della scritta 'Ciao Mondo!'. Che si parla di byte lo abbiamo specificato scrivendo db che appunto avverte l'assemblatore che ogni dato, in questo caso ogni lettera, occupa un solo byte. Come dicevo non vogliamo specificare un indirizzo numerico della memoria perché, sebbene questo sia possibile, creerebbe confusione ed errori al momento del caricamento del programma in memoria per essere eseguito. Apriamo una piccola parentesi.

### **Indirizzi hardware ed etichette assembly**

Affinché una sequenza di istruzioni venga eseguita dalla macchina, essa deve essere caricata nella memoria. Come è noto, nei computer che usiamo attualmente, siano essi personal computer, smartphone o altri sistemi, i programmi, cioè il software, è normalmente caricato nella *memoria di massa*, cioè in un'unità di memorizzazione vista dal sistema come una periferica e non come la memoria di lavoro. La CPU per eseguire un programma necessita quindi che esso sia in memoria e che sul PC sia scritto l'indirizzo della prima istruzione da eseguire. Questo compito (caricare il programma e impostare il PC) spetta ad un altro programma, il *loader*. Chi è il loader? Se è presente il sistema operativo, come appunto accade nelle esperienze comuni (chi usa il personal computer, chi lo smartphone ecc) allora il loader è un modulo del sistema operativo stesso. Il fatto è che purché non si abbia veramente controllo su ciò che si sta facendo (cioè se siete esperti davvero) non avete modo di controllare a che indirizzo il loader caricherà il programma che state scrivendo, quindi, se non si è a conoscenza dell'indirizzo di memoria in cui il programma è caricato, diventa complesso e pericoloso far riferimento dall'interno del programma ad indirizzi numerici perché per esempio si

correrebbe il rischio di scrivere un dato dove il loader ha già caricato le istruzioni. Quindi, per evitare questi problemi, si lascia che la trasformazione in indirizzi numerici avvenga al momento dell'esecuzione in memoria, mentre nella fase di programmazione si usano delle etichette che si riferiscono all'indirizzo in cui poi verrà fisicamente memorizzato il dato.

## Chiamate di sistema

Ora che abbiamo scritto il codice per porre il messaggio in memoria, dobbiamo stamparlo sul video, quindi interagire con una periferica hardware. Come anticipato questo non è possibile se è il sistema operativo ad avere il controllo, allora per ottenere lo stesso effetto si usa una procedura del sistema operativo a cui appunto ci si riferisce con "chiamata di sistema". Linux ha una lista di funzioni (procedure) offerte al programmatore. Esse hanno un codice, un nome ed una serie di parametri che devono essere specificati al momento della chiamata. In tabella IX sono riportate le prime quattro chiamate di sistema di Linux.

*Tabella IX. Elenco delle prime quattro funzioni di sistema di Linux x86\_64.*

%rax	System call	Uso
0	sys_read	Legge fino al numero di byte specificati da un file, di cui deve essere fornito il descrittore, e le scrive in memoria all'indirizzo che viene specificato
1	sys_write	Scrive sul file specificato, fino al numero di byte richiesto, attingendo i dati dall'indirizzo specificato.
2	sys_open	Apre il file specificato.
3	sys_close	Chiude il file specificato.

In ambiente Linux una chiamata ad una funzione di sistema si fa come segue:

- Si memorizza il numero della chiamata che si vuole eseguire nel registro rax.

- Si memorizzano gli argomenti della chiamata nei registri rdi, rsi, rdx, r10, r8 e r9.
- Si esegue l'istruzione syscall

Il primo parametro della chiamata specifica cosa stiamo chiedendo al sistema operativo, e nel caso specifico stiamo chiedendo di eseguire la procedura `sys_write` che chiede a Linux di inviare verso un dispositivo di output una sequenza di byte. Linux ha bisogno di conoscere:

- A quale dispositivo inviare i byte.
- Qual è il primo indirizzo in memoria dove iniziano i byte da inviare.
- Quanti sono i byte.

quindi dovremo inserire tutte queste informazioni nei registri visti prima e visto che ci sono solo tre informazioni useremo solo i primi tre registri. La prima informazione è che vogliamo inviare i byte (cioè i codici ASCII della scritta 'Ciao Mondo!') al dispositivo che il sistema ritiene l'output di default (standard output). Se state usando un normale PC, questo è il vostro video ed è identificato dal codice 1. Bene, cominciamo a vedere un po' di codice, poi scriveremo per bene tutto il programma in ordine:

```
section .text  
_start:  
    mov rax,1 ;
```

L'istruzione `mov` l'abbiamo già incontrata e abbiamo visto che serve per copiare dei dati tra indirizzi di memoria o registri. In questo caso stiamo copiando il valore immediato 1 nel registro `rax`. Ora dobbiamo specificare che vogliano scrivere il messaggio sul video, questo per Linux è visto come un file il cui descrittore ha valore 1. Per farlo dovremo solo copiare il valore 1 nel registro `rdi`, quindi:

```
    mov rdi,1 ;
```

Ora dobbiamo specificare qual è l'indirizzo del primo byte in cui abbiamo memorizzato il messaggio da scrivere. Come precedentemente chiarito, non è conveniente specificare nel

programma l'indirizzo numerico, è meglio usare un etichetta, quindi scriveremo:

```
mov rsi,messaggio ;
```

L'ultimo parametro che deve ancora essere specificato è il numero di byte da scrivere sull'output. La scritta 'Ciao Mondo!' sono 11 caratteri quindi:

```
mov rdx,11 ;
```

Sarebbe stato meglio aggiungere un ultimo carattere in coda alla scritta in modo da mandare a capo il cursore... lo facciamo? Si dai, lo facciamo. Si deve sapere che il ritorno a capo è gestito anche dalla tabella ASCII, infatti il codice decimale 10 (esadecimale A) corrisponde esattamente ad un comando di ritorno a capo (questo argomento viene trattato esaustivamente più avanti). Se aggiungiamo al message il codice 10, dopo aver stampato Ciao Mondo! il cursore sarà riportato a capo, dando un effetto più ordinato:

```
section .data  
messaggio: db 'Ciao Mondo!',10  
...  
mov rdx,12 ;
```

Ovviamente oltre ad aver accodato il codice 10 nel messaggio abbiamo incrementato di 1 il valore nel registro rdx. Fatto questo non resta che chiamare l'esecuzione della chiamata di sistema:

```
syscall ;
```

## **Terminare l'esecuzione**

Per completare davvero tutto e provare ad editare, assemblare, linkare e eseguire il codice manca un' ultimo aspetto: terminare il programma. Se non informiamo il sistema operativo che il programma ha terminato il suo compito, esso lascerà che il meccanismo di Fetch/Decode/Execute che anima il computer prosegua il suo corso anche oltre l'ultimo indirizzo di memoria in cui era caricato il programma. Negli indirizzi successivi ci saranno dei valori che potrebbero essere codici di altri programmi o dati, in tutti i

casi, non sono istruzioni del programma in esecuzione. Per questo motivo, quando il programmatore giunge alla fine del programma deve lanciare una chiamata di sistema che arresti l'esecuzione del programma e liberi la memoria che esso ha occupato. La procedura da chiamare ha il codice 60 e si chiama sys\_exit. Il codice per chiamarla è:

```
mov rax,60 ;  
syscall
```

quindi, mettendo tutto insieme abbiamo:

**Listato 2.1** ciao.asm

```
global _start  
section .data  
messaggio: db 'Ciao Mondo!', 10  
section .text  
_start:  
    mov rax,1 ;  
    mov rdi,1 ;  
    mov rsi, messaggio ;  
    mov rdx,12 ;  
    syscall ;  
    mov rax,60 ;  
    syscall ;
```

Ora che abbiamo il codice salviamolo con il nome ciao.asm in una cartella, per esempio potremmo chiamarla nn. Poi usando una shell (una finestra del terminale), andiamo nella cartella in cui abbiamo salvato ciao.asm. Nel mio caso per andarci uso i seguenti comandi:

**Compilazione** ciao.asm

```
>~cd ~/Documents/nn  
~/Documents/nn> nasm -felf64 ciao.asm -o ciao.o  
~/Documents/nn> ld -o ciao ciao.o  
~/Documents/nn> chmod u+x ciao
```

```
~/Documents/nn> ./ciao
```

Il risultato dovrebbe essere la stampa a video del messaggio di saluto! Prima di procedere con altri esempi in assembler poniamo l'attenzione sulla scritta `_start` che abbiamo inserito senza commentare. Questa indica qual è la prima istruzione che deve essere eseguita quando il programma entra in esecuzione.

### Un esempio di somma

Proviamo ora ad eseguire la somma tra due numeri, 1 e 2, che è un esempio concreto dei concetti introdotti qualche paragrafo fa. Come abbiamo già visto la ALU è capace di eseguire la somma tra numeri interi (non lo abbiamo visto ma può farlo anche con numeri frazionari). L'operazione per eseguire una somma sull'architettura Intel®64 o sulla AMD64 è la ADD che però si presta a varie interpretazioni, infatti questa architettura è una cosiddetta CISC, cioè ricca di istruzioni, e anche la semplice ADD può essere intesa in diversi modi. La sintassi per usarla è la seguente:

```
ADD op_1, op_2;
```

l'azione è di sommare i due operandi e memorizzare il risultato nell' operando `op_1`. Gli operandi `op_1` e `op_2` possono però essere:

- indirizzi di memoria,
- registri,
- valori immediati

e non sono ammesse le configurazioni memoria/memoria e ovviamente `op_1` di tipo immediato.

In questo esempio useremo la *operation encoding* della ADD di tipo RM la quale indica che l' operando di destinazione è di tipo registro (R) mentre quello sorgente è di tipo memoria (M). Come prima cosa dobbiamo definire la label per memorizzare i dati in memoria. Useremo `a`, `b` per memorizzare i due addendi e `c` il

risultato, infine d per memorizzare il carattere 10 del ritorno a capo.

```
section .data
```

```
a: db 1
```

```
b: db 2
```

```
c: db 0
```

```
d: db 10
```

Per eseguire la somma dobbiamo caricare uno degli addendi in un registro e poi sommare l'altro addendo. Il risultato della somma sarà caricato nel registro stesso, sovrascrivendo il precedente valore. Siccome abbiamo dichiarato che gli addendi sono dati che occupano un solo byte, è necessario che anche il registro occupi un solo byte. In realtà i registri dell'architettura 64 hanno 8 byte (8 byte per 8 bit = 64! ecco perché si chiama 64bit!!!) però in base a come vengono chiamati nel codice assembler è possibile usarli come registri a 8, 16, 32 o 64 bit. Per usare il registro a come un registro ad 8 bit ci si deve riferire ad esso come al:

```
global _start
```

```
_start:
```

```
    mov al,[a] ;
```

```
    add al,[b] ;
```

```
    add al,48 ;
```

```
    mov [c],al ;
```

Esaminiamo il codice. Come prima cosa copiamo il valore della a nel registro al. Le parentesi quadre servono per specificare che è il contenuto della memoria dell'indirizzo di etichetta a che deve essere copiato e non il valore dell'indirizzo stesso (vedi tabella X).

*Tabella X. La riga centrale mostra la memoria. La prima colonna indica l'indirizzo del primo byte, nelle successive quattro colonne è riportato il contenuto degli indirizzi di memoria. I punti interrogativi significano che le rispettive celle di memoria non sono state settate esplicitamente dal programma. La riga sopra indica la relazione tra la label, l'indirizzo e il contenuto della memoria.*

a	[a]	[a+1]	[a+2]	[a+3]
400	1	?	?	?

Indirizzo	Byte 0	Byte 1	Byte 2	Byte 2
-----------	--------	--------	--------	--------

Poi, con l'istruzione successiva, si somma il contenuto della memoria etichettata da b all'indirizzo al che quindi al termine della somma conterrà la sequenza binaria 00000011, cioè il numero 3.

La strategia per stampare il risultato a video è quella di copiarlo in un indirizzo della memoria, poi invocare la funzione di sistema `sys_write` esattamente come abbiamo fatto prima. Prima di farlo però dobbiamo ragionare bene sul contenuto del registro al. In al c'è proprio il numero 3. Se inviamo allo standard output il numero 3 esso lo interpreterà come il codice ASCII di ciò che vogliamo visualizzare. Il codice ASCII 3 si riferisce al comando ETX che informa il video che la trasmissione dei dati è finita e non produce alcuna stampa. Per stampare il risultato della somma dobbiamo sommare al risultato il codice ASCII dello 0 che è il 48, come si vede nella riga successiva. In questo modo a video sarà stampato il numero 3. Diciamo subito che il codice così come è impostato può gestire solo delle somme che non superino il numero 9, altrimenti che succede? Provate, trovate una tabella ASCII e fate qualche prova!

Fatto questo copiamo il contenuto del registro al all'indirizzo etichettato c poi eseguiamo la chiamata di sistema. Di seguito il codice completo:

### Listato 2.2 somma.asm

```
section .data
a: db 1
b: db 2
c: db 0
d: db 10
section .text
global _start
_start:
```

```
mov al,[a] ;
add al,[b] ;
add al,48 ;
mov [c],al ;
mov rax,1 ;
mov rdi,1 ;
mov rsi,c ;
mov rdx,2 ;
syscall ;
mov rax,60 ;
syscall ;
```

Per provare il codice si procede esattamente come prima solo che invece di ciao.asm suggerisco di chiamare il file somma.asm, quindi:

### **Compilazione** somma.asm

```
~/Documents/nn> nasm -felf64 somma.asm -o somma.o
~/Documents/nn> ld -o somma somma.o
~/Documents/nn> chmod u+x somma
~/Documents/nn> ./somma
```

Ancora un particolare su questo codice. Se siete programmatori attenti, vi sarete chiesti a che cosa serva la d.

In d abbiamo memorizzato l'ASCII del ritorno a capo, ma perché viene stampato anche esso? La ragione è che il valore decimale 10, binario 00001010, memorizzato in d è dichiarato subito dopo la c, quindi all'indirizzo successivo. Siccome nel registro rdx (usato per specificare quanti byte inviare all'output) abbiamo caricato 2 anziché 1, la stampa proseguirà attingendo la memoria anche dal secondo indirizzo! Non è una buona pratica di programmazione, ma rende l'idea di come funziona la memoria e dell'attenzione che si deve usare quando si programma a basso livello.

## Controllo del flusso

Fino qui abbiamo visto l'esecuzione di programmi che si è svolta sempre secondo lo schema sequenziale  $PC \leftarrow PC + 4$  (nell'architettura Intel® il PC si chiama rip) che impone ad ogni ciclo di esecuzione che il program counter venga incrementato di 4 in modo da passare l'istruzione successiva, fino alla chiamata sys\_exit. Ora vogliamo provare a controllare il flusso di esecuzione del codice generando una biforcazione nella sequenza di esecuzione delle istruzioni. Proveremo quindi a eseguire la somma tra due numeri precedentemente memorizzati. Se la somma è minore di 10 produrremmo la scritta 'Somma minore di 10', altrimenti 'Somma maggiore di 10'.

### Program counter e jump

Come precedentemente accennato, per modificare il flusso di esecuzione del programma è necessario agire sul valore del registro PC. Alla pratica non si può scrivere direttamente sul registro ma si può usare l'istruzione "jump" che significa salta specificando l'indirizzo a cui saltare il quale verrà scritto nel PC al posto dell'indirizzo già presente. Il salto (jump) può essere *incondizionato*, cioè deve essere eseguito senz'altro, oppure *condizionato*, cioè deve essere eseguito al verificarsi di una certa condizione. Spesso la condizione è espressa attraverso il valore di un registro. Prima di vedere il codice, cerchiamo di capire meglio l'architettura dei salti.

Consideriamo il seguente codice:

```
inizio:  
    mov rbx,0;  
    jmp inizio
```

Durante l'esecuzione del programma, cioè dopo la fase di load, l'etichetta `inizio` ha il valore dell'indirizzo in cui è memorizzata l'istruzione `mov rbx,0`. L'istruzione seguente, `jmp` è un salto del flusso di istruzioni fino all'etichetta. In pratica scrive nel PC (registro rip) l'indirizzo indicato dalla etichetta `inizio`, cioè

dell'istruzione `mov`. È chiaro che l'esecuzione di tale codice porterebbe a quello che si chiama *ciclo infinito* o in inglese infinity loop. L'istruzione `jmp` però può essere usata anche in modo costruttivo per ottenere il controllo del flusso di esecuzione delle istruzioni in tre diversi modi:

- esecuzione condizionata: si esegue una o più istruzioni solo se una certa condizione è rispettata;
- ripetizione di un blocco di istruzioni per più volte: si ripete un insieme di istruzioni per un numero definito di volte (iterazione);
- richiamo di una procedura: si interrompe la linea di esecuzione del codice per saltare più avanti ad eseguire un certo gruppo di istruzioni, poi si riprende il codice dall'istruzione successiva all'ultima eseguita.

In questo esempio implementeremo il primo modo, cioè l'esecuzione condizionata, quello che presto in C chiameremo il *costrutto sintattico if* o più semplicemente l'`if`.

### L'istruzione `compare`

Prendiamo il codice di prima e dopo la somma confrontiamo il risultato con il valore immediato 10 usando l'operazione compare (`cmp`) che permette di confrontare il contenuto di un registro con un valore immediato:

```
section .data
a: db 1
b: db 2
c: db 0
d: db 10
section .text
global _start
_start:
    mov al,[a] ;
    add al,[b] ;
```

```
add al,48 ;  
mov [c],al ;  
cmp al,58;
```

Che effetto ha la *compare* sulla macchina? Come abbiamo iniziato a capire, le operazioni che svolgiamo attraverso l'assembler hanno sempre come unico effetto quello di modificare lo *stato* dei registri, cioè i valori che vi sono memorizzati. Può sorgere spontanea la domanda su come queste due sole azioni portino poi alle varie prestazioni a cui siamo abituati dal personal computer o da uno smartphone o ancora da un altro dispositivo. Qualcuno lo avrà già capito, perché in parte ne abbiamo già parlato, per qualcun altro invece "repetita iuvant". Le operazioni della CPU agiscono solo modificando lo stato dei registri i quali però sono collegati elettricamente ai BUS e quindi i valori interni di alcuni registri sono trasmessi sulla linea del BUS e, a quel punto, le periferiche esterne che vi si collegano ci fan quel che vogliono! Immaginiamo una youtuber che si sta prodigando nel mostrare la procedura per realizzare una torta in un video in streamimg. Quello che lei può fare è limitato all'uso degli utensili e ingredienti che ha in cucina, eppure, se ha molti ascoltatori affezionati, le sue azioni saranno colte e replicate anche in altre case a chilometri di distanza. Lo stesso accade nel personal computer, dove le distanze tra i componenti, se paragonate alle dimensioni dei transistor sono enormi. La CPU modifica il suo stato interno e la memoria e le periferiche agiscono di conseguenza. Detto questo, torniamo al punto. L'istruzione cmp agisce sul registro EFLAGS che, come suggerito dal nome, è il registro dei *flag*.

Il concetto di flag, che si può tradurre con bandierina (la stessa che sventola il guardalinee quando un calciatore va in fuorigioco per intenderci, non la bandiera della patria!) è molto importante nella tecnologia informatica. L'uso è davvero simile a quello dei guardalinee. L'arbitro, che in questo caso sarebbe la CPU, non può avere la testa su tutto mentre segue il gioco, così se ad esempio è

impegnata ad eseguire una sottrazione non si accorge se il minuendo è minore del sottraendo. Per questo ci sono i flag che sono singoli bit all'interno di un registro. Normalmente i flag hanno valore 0, se capita qualcosa, per esempio un sottraendo che supera un minuendo o che il risultato di una somma genera un numero che ha più cifre del numero di bit del registro che lo deve memorizzare (*overflow*), il bit viene posto ad 1.

### **Salto condizionato: compare + jump = if**

L'istruzione `cmp op1_1, op2` agisce sottraendo il secondo operando dal primo e modificando i registri di conseguenza. Risulta chiaro allora che essa non è sufficiente per controllare il flusso del programma, dopo il confronto è necessario eseguire un'altra istruzione il cui comportamento si basi sullo stato dei registri. E così siamo tornati al punto! L'istruzione che ci serve è appunto il `jump indirizzo` che ha l'effetto di copiare l'indirizzo specificato dall'etichetta nel PC (che ora sappiamo essere il registro `rip` nell'architettura `86_64`). Ci sono diversi tipi di jump, quello che useremo noi è il codice `jb` che sta per `jump if below`, quindi esegui il "salto" solo se il primo operando è minore del secondo. Dobbiamo aggiungere al codice l'istruzione di salto e anche un indirizzo a cui saltare:

```
cmp al,58;
```

```
ja maggiore;
```

dove `maggiori` è un etichetta che mettiamo al primo indirizzo di memoria a cui vogliamo salti l'esecuzione del programma se il contenuto del registro `al` è maggiore di 58. Perché 58 e non 10? Troppo facile, pensateci da soli! Di seguito mostro il codice completo. Salvatelo con il nome di `condizione.asm`, assemblatelo con `nasm`, linkatelo con `ld` e assegnetegli i permessi di esecuzione con `chmod u+x if`. Ecco il codice, fate un po' di prove cambiando i valori di `a` e `b` ma che siano sempre minori o uguali a 255.

### **Listato 2.3 condizione.asm**

```
section .data
a: db 1
b: db 21
c: db 0
d: db 10
msg1: db 'Somma minore di 10',10
msg2: db 'Somma maggiore di 10',10
section .text
global _start
_start:
    mov al,[a] ;
    add al,[b] ;
    add al,48 ;
    mov [c],al ;
    cmp al,58 ;
    mov rax,1 ;
    ja maggiore ;
    mov rdi,1 ;
    mov rsi,msg1 ;
    mov rdx,19 ;
    syscall ;
    mov rax,60 ;
    syscall ;
maggiori:  mov rdi,1 ;
    mov rsi,msg2 ;
    mov rdx,21 ;
    syscall ;
    mov rax,60 ;
    syscall ;
```

### **Compilazione** condizione.asm

```
~/Documents/n> nasm -felf64 condizione.asm -o
condizione.o
```

```
~/Documents/nn> ld -o condizione condizione.o  
~/Documents/nn> chmod u+x condizione  
~/Documents/nn> ./condizione
```

Il codice mostrato non presenta grosse difficoltà, l'unico aspetto che può dar da pensare è come si realizzzi la mutua esclusione dei due messaggi (cioè: o uno o l'altro). Basta vedere che se il contenuto di `al` è minore di 58, il flusso di esecuzione continua progredendo normalmente agli indirizzi successivi. Incontra quindi la chiamata a `sys_write` con il messaggio `msg1`, poi l'esecuzione termina per via della chiamata a `sys_exit`, quindi il secondo messaggio ('Somma maggiore di 10') non viene stampato. Se invece vale il contrario, tutte le istruzioni necessarie a stampare il primo messaggio sono saltate (`jump`) e quindi il messaggio non viene stampato, sono eseguite invece le istruzioni per il secondo poi, come sopra abbiamo l'uscita dal programma.

## Iterare un blocco di istruzioni

L'assembly è un po' ostico all'inizio, ma pian piano ci si prende gusto e il piacere di capire come funziona la macchina e la consapevolezza che la comprensione acquisita permette di concretizzare delle idee dà una vera soddisfazione. L'importante è non prendersi troppo sul serio.

Vediamo un esempio relativo al secondo punto dell'ultimo elenco sopra, quello riguardante la ripetizione di un blocco di istruzioni usando il jump.

### **compare + jump = iterazione**

L'idea è semplice. Prima del gruppo (ma chiamiamolo blocco) di istruzioni che vogliamo ripetere (su cui vogliamo iterare), mettiamo un' etichetta, per esempio **ciclo** per indicare che lì inizia l'iterazione. Poi, alla fine mettiamo un jump al ciclo, così l'effetto sarà quello di iterare sul blocco di istruzioni. Fin qui va bene, ma come abbiamo già visto questo porta ad un ciclo che non termina, il che in alcuni casi è proprio quello che vogliamo, ma in questo caso vogliamo provare un ciclo che termini al verificarsi di una data condizione. Per esempio vogliamo provare a stampare a video una a una tutte le lettere di cui è composto un messaggio di testo finché non incontriamo il carattere ASCII NULL, che corrisponde al codice 0. Il codice è il seguente e potete salvarlo e assemblarlo chiamandolo **ciclo.asm**

#### **Listato 2.4** **ciclo.asm**

```
section .data
end: db 10
messaggio: db
67,105,97,111,32,109,111,110,100,111,33,0
section .text
global _start
_start:
    mov rax,1 ;
    mov rdi,1 ;
```

```
    mov r9,messaggio ;  
ciclo:  
    mov rsi,r9;  
    mov rdx,1 ;  
    syscall ;  
    mov rsi,end ;  
    syscall;  
    add r9,1 ;  
    mov r10,[r9];  
    cmp r10,0 ;  
    jne ciclo ;  
    mov rax,60 ;  
    syscall ;
```

### **Compilazione** ciclo.asm

```
~/Documents/nn> nasm -felf64 ciclo.asm -o ciclo.o  
~/Documents/nn> ld -o ciclo ciclo.o  
~/Documents/nn> chmod u+x ciclo  
~/Documents/nn> ./ciclo
```

Questa volta c'è un po' da ragionare. Anzitutto vediamo che a partire dall'indirizzo etichettato con `messaggio` memorizziamo undici byte (uno contiguo all'altro) poi uno finale dal valore 0. I byte sono appunto contigui, quindi partendo dall'indirizzo del primo, si può far riferimento al secondo aumentando tale indirizzo di 1. Vogliamo usare questa idea per stampare uno alla volta i caratteri relativi ai codici memorizzati in `messaggio`. Quindi, anziché stampare una *stringa* unica, stampare singolarmente le lettere che la compongono. Avete già provato a leggere il messaggio? Provate! Bene, questa volta l'operazione è una vera sfida e quel che c'è da fare è un algoritmo iterativo, cioè un algoritmo che ripete una certa operazione tante volete finché ne raggiunge la fine. L'operazione da ripetere è la chiamata di sistema `sys_write` per stampare una lettera, poi, a ruota una

seconda chiamata per stampare un ritorno a capo.

## Stampare una stringa

La difficoltà è che per ogni chiamata dobbiamo passare alla lettera successiva della stringa. Disponiamo dell'indirizzo della prima lettera, quindi possiamo incrementarlo di uno ad ogni iterazione! Il meccanismo per farlo è mostrato nel codice, ma vediamolo insieme. Prima dell'inizio del blocco di istruzioni che sarà iterato (in gergo si dice anche ciclare), copiamo con una `mov` l'indirizzo nel registro ad uso generale (general purpose register) `r9`. Subito dopo comincia il blocco su cui avverrà l'iterazione, esso è indicato dalla label `ciclo` che serve anche a memorizzare l'indirizzo della prima istruzione del blocco stesso. All'interno del blocco si copia il contenuto di `r9` in `rsi` e si setta `rdx` per stampare un solo byte, poi si esegue una chiamata di sistema alla `sys_write` il cui codice era stato inserito in `rax` poche righe sopra. L'effetto a *runtime* è quello di stampare la prima lettera del messaggio, quella che ha codice ASCII 67. Dopo questo, il contenuto di `r9` viene aumentato di 1, quindi ora è l'indirizzo del secondo byte del messaggio. Subito dopo, il contenuto della memoria (non del registro) il cui indirizzo è memorizzato in `r9` viene copiato in `r10` e confrontato con il valore 0. Di seguito abbiamo un `jne` a `ciclo`, quindi l'esecuzione riprende da `ciclo` se il contenuto della memoria *puntato* da `r9` è diverso da 0. Questa non è una mia invenzione, come vedrete presto, le stringhe anche in C sono terminate dal codice 0. Quando infine `r9` punterà allo 0, il jump non verrà eseguito e il flusso delle istruzioni continuerà con l'istruzione successiva al jump, cioè la chiamata di sistema alla `sys_exit`.

## Chiamare una procedura

Le procedure in Assembler sono la base delle funzioni del linguaggio C, in altre parole, una funzione in C viene compilata in una procedura, per questo avere almeno un'idea intuitiva su come opera la macchina a fronte di una codifica C di una funzione, aiuta

ad aumentare la comprensione e la consapevolezza di ciò che si sta facendo nella programmazione. Detto questo, vediamo anzitutto che le procedure sono un meccanismo che permette di isolare un pezzo di "logica" ed usarla a "scatola chiusa" o come si dice in informatica in una *black box*. A scatola chiusa significa che si ignorano i dettagli di come la logica della procedura è stata implementata limitandosi a conoscere le convenzioni per richiamare la procedura e accedere al risultato.

Prima di fare altre considerazioni, vediamo un esempio molto semplice che chiarirà i concetti esposti. Costruiamo una procedura che esegue la somma dei quadrati di due parametri. In termini matematici possiamo vedere la procedura come una funzione di due variabili definita come segue:

$$f(x, y) = x^2 + y^2$$

Alle procedure deve essere fornito un nome, che in realtà altro non è che un' etichetta per l'indirizzo di memoria da cui parte la prima istruzione del loro codice. Ho chiamato questa procedura `somma_quadrati`.

## **Stack e procedure**

Per eseguire il codice della procedura si usa l'istruzione macchina `call somma_quadrati` che ha l'effetto di "salvare" sullo stack l'indirizzo della prossima istruzione (prossima rispetto alla `call`, cioè l'istruzione dopo la `call`) e poi di eseguire il `jmp` all'indirizzo specificato dalla `call`. A questo punto viene eseguito il codice della procedura che termina quando incontra l'istruzione `ret`. Lo scopo di questa istruzione è quello di eseguire il "ritorno" alla funzione che ha chiamato (caller procedure). Come si può intuire, il ritorno al chiamante consiste semplicemente nell'impostare sul PC (registro `rip`) l'indirizzo precedentemente salvato sullo stack. Per questo la `ret` è equivalente all'operazione `pop rip`, cioè all'estrazione dallo stack dell'indirizzo salvato e alla sua memorizzazione sul `rip` cioè il PC.

## I parametri della procedura

I parametri su cui opera la procedura (l'argomento della funzione) sono passati dalla chiamante alla chiamata per mezzo dei due registri edi ed esi. Il passaggio dei parametri per mezzo di registri è usato da diversi linguaggi di programmazione e anche dal C. I parametri però possono anche essere passati sullo stack.

## Il risultato della procedura

La procedura chiamata esegue le proprie elaborazioni e memorizza il risultato nel registro eax rendendolo in questo modo disponibile alla procedura chiamante al ritorno. La chiamante dopo aver prelevato il valore di ritorno provvede a stamparlo. Per ottenere il codice ASCII del numero da stampare somma il valore ritornato a 48, appunto il codice ASCII dello 0. Purtroppo questa semplice tecnica è efficace solo per numeri minori di 9, se provate ad eseguire il codice sostituendo ai valori impostati per la x e per la y dei numeri maggiori, vi accorgerete che invece di un numero verrà stampato un carattere. Cercate il codice ASCII di quel carattere su una tabella ASCII in rete. Sottraete 48 e troverete che è proprio il risultato atteso!

Di seguito riporto il codice dell'esempio descritto.

### Listato 2.5 procedura.asm

```
section .data
a: dd 48
x: dd 2
y: dd 2

section .text
global _start

_start:
;; *** Procedura Caller o chiamante***
;; chiama la funzione somma dei quadrati
```

```

;; i due parametri della somma sono memorizzati in
edi e esi
    mov edi,[x] ;
    mov esi,[y] ;
;; Chiamata alla procedura e attende il risultato in
eax
    call somma_quadrati ;
;; Dopo la chiamata accede al risultato
    add [a],eax ;
    mov rax,1 ;
    mov rdi,1 ;
    mov rsi,a ;
    mov rdx,1 ;
    syscall
    mov rax,60 ;Fine dell'esecuzione
    syscall ;

somma_quadrati:
;; *** Procedura Callee o chiamata***
;; calcola x^2
    mov eax,esi ;
    mul esi ;
    mov esi,eax ;
;; calcola y^2
    mov eax,edi ;
    mul edi ;
    mov edi,eax ;
;; somma x^2 e y^2
    add edi,esi ;
;; copia la somma in eax come accordato con il
chiamante
    mov eax,edi ;
    ret ;

```

### **Compilazione** procedura.asm

```
~/Documents/nn> nasm -felf64 procedura.asm -o  
procedura.o  
~/Documents/nn> ld -o procedura procedura.o  
~/Documents/nn> chmod u+x procedura  
~/Documents/nn> ./procedura
```

## 2.3 Il compilatore C

Le istruzioni dell'assembly sono in relazione uno a uno con l'insieme delle operazioni che l'architettura è capace di compiere e per tanto permettono di codificare qualsiasi algoritmo eseguibile dalla macchina quindi l'uso dell'assembly per scrivere programmi "sarebbe" consigliato. Come si è avuto modo di vedere però, codificare un algoritmo in termini di "operazioni macchina" è piuttosto laborioso o almeno così sembra a noi esseri umani, perché la meccanica della macchina, intesa come l'insieme di regole che ne governa il funzionamento, sembra essere differente dalla meccanica del cervello umano che si basa principalmente sulla capacità di esprimersi con il linguaggio. Ciò di cui manca la macchina è un linguaggio e questo nella storia passata è stato sentito come una limitazione alle capacità di produrre programmi in tempi "ragionevoli", dove si intenda, e questo non è facile da comprendere subito, che ragionevole è ciò che ci aspettiamo di riuscire a fare, non ciò che davvero riusciamo a fare. Se un programmatore si aspetta di riuscire a scrivere un programma funzionante in sessanta ore e ne impiega trecento a causa di difficoltà tecniche che nascono dalla non intuitività dell'assembly, troverà le trecento ore un tempo non ragionevole.

Per detti motivi, l'assembly non soddisfaceva e per questo, usando l'assembly si sono scritti dei programmi detti compilatori. Un programma compilatore è un programma eseguibile, che accetta come input un file (anche più di uno) contenente del

codice *sorgente* scritto in un linguaggio non assembly e restituisce in output l'"equivalente" scritto in assembly.

Spesso le opzioni del compilatore comprendono anche le operazioni di assemblaggio e linkaggio viste prima, quindi compilando un file sorgente si ottiene direttamente un eseguibile. Studiare bene come è scritto un compilatore è un'esperienza formativa e interessante, che consiglio caldamente. In questo contesto però ci limitiamo a descriverne l'uso. Prima di procedere con i primi programmi in C e l'uso del compilatore per compilarli, chiamiamo solo cosa significa che il programma prodotto dal compilatore è equivalente a quello scritto dal programmatore in un certo linguaggio che, d'ora in avanti, supporremo essere il C.

Abbiamo detto che l'esecuzione di un programma ha come effetto la variazione dello stato della CPU (qualche paragrafo qua sopra...) e della memoria. L'esistenza di periferiche collegate ai BUS o al chipset, permette che lo stato della CPU/memoria si rifletta in *trasduzioni* dei segnali elettrici ricevuti in input dalla periferica. Per fare un esempio, nel codice *ciao.asm* visto sopra, abbiamo visto come lo stato interno della CPU/memoria sia trasdotto nella stampa a video della scritta 'Ciao Mondo!'. Quando scriviamo un programma in assembly controlliamo sia lo stato della CPU che quello della memoria, quindi possiamo dire che ciò che identifica un programma e lo differenzia da un altro è come evolve lo stato CPU/memoria nel tempo. Due programmi (assembly) che, a parità di input, producano la stessa evoluzione dello stato sono equivalenti. Questa definizione è corretta, ma è anche troppo "stringente" se il quesito è posto dal punto di vista degli effetti sulle periferiche. Tornando al nostro esempio precedente, si può ottenere un programma che scriva 'Hello World!' in molti modi, ognuno con una propria evoluzione dello stato ma ognuno con lo stesso risultato finale. Tra due programmi che realizzano lo stesso output ma hanno una diversa evoluzione

dello stato, probabilmente, uno è più efficiente in termini di energia e tempo. Bene, chiarito questo diciamo che lo scopo del compilatore è di ricevere in input un programma scritto in C (perché qui parliamo di C) e trasformarlo in un programma equivalente in assembly, dove per equivalente intendiamo solo rispetto all'input e all'output. Quindi il compilatore ha abbastanza discrezionalità su come impiegare le risorse della CPU per ottenere il risultato voluto (cioè input e output). Esistono comunque delle regole, per esempio le *variabili* dichiarate nel prototipo di una funzione C sono allocate in memoria nello stack (visto poco sopra) dal compilatore. Questo significa che il compilatore scriverà il codice assembly affinché i valori che nel programma in C sono gestiti attraverso l'uso di variabili, nel programma in assembly siano memorizzati in memoria nello stack e non, per fare un esempio, in un registro. Al contrario, se dichiariamo una variabile di tipo `register`, indichiamo al compilatore che può anche memorizzarla in un registro.

Quindi l'uso consapevole del compilatore porta ad ottenere programmi efficienti risparmiandoci i dettagli hardware del sistema su cui stiamo lavorando. Questo appena citato è un altro punto importante. Supponiamo infatti di voler scrivere comunque in assembly perché ci troviamo bene così, abbiamo un problema, che l'assembly è specifico di ogni architettura. Se avessimo passato venti mesi per produrre un buon assembly per l'AMD64, per poi scoprire che per un cambio di tecnologia il programma dovrà girare su un architettura PA-RISC, saremmo spacciati. Usando un linguaggio di più alto livello, come il C, non avremmo conseguenze da un cambio di specifiche hardware, semplicemente useremmo il compilatore dell'architettura in questione. È chiaro che le cose non sono sempre così semplici, a volte si devono scrivere comunque delle librerie specifiche per un'architettura, però per cominciare, quello che abbiamo detto è sufficiente.

## 2.4 Il flusso delle istruzioni

La caratteristica principale del C è che ogni programma consiste in una funzione: ogni programma è una funzione che prevede un argomento e un risultato, ricordate  $z = f(x, y)$ ? Il C è fatto proprio così.

Scrivere un programma in C significa anzi tutto scrivere una funzione che deve avere il nome prestabilito `main`, cioè principale. Questo è il punto di ingresso del programma in C e, quando compilato, la prima istruzione della `main` diventerà la prima istruzione dopo lo `_start`. Come nella matematica, anche nella programmazione esistono delle regole da seguire per rendere i nostri scritti chiari e inequivocabili.

Se in matematica io scrivessi:  $f(x) = 3x$  avrei scritto qualcosa di ambiguo, se non addirittura scorretto. Fare errori quando si parla è normale, come è normale correggerli di fronte alla perplessità dell'interlocutore. Nel linguaggio scritto la cosa è più complessa perché non c'è un rapporto diretto e la comunicazione è mediata dallo scritto stesso. Se ci pensiamo è la differenza tra un compito in classe e un'interrogazione. Quindi è necessario che i programmi che vengono scritti comunichino dei contenuti (istruzioni) inequivocabili. Per ottenere questo risultato ci si deve concentrare su due aspetti: *sintassi* (e sì... sempre lei, come a scuola!) e *semantica*.

Cosa sia la sintassi è un concetto ben noto: è l'insieme delle regole per scrivere in modo formalmente corretto, ma senza giudizio sul significato del contenuto. Per esempi la frase "la scatoletta ha mangiato il gatto" è sintatticamente corretta, perché articolo, soggetto, predicato e complemento sono tutti a posto, però il significato trasmesso non è quello desiderato, quindi c'è un errore di significato, cioè semantico. Vediamo anzi tutto la sintassi per scrivere la funzione `main`:

```
int main()
{
}
```

La prima cosa da notare è la costruzione sintattica (detta anche costrutto sintattico). Anzi tutto abbiamo la *keyword* int che descrive il tipo di ritorno della funzione e in matematica rappresenterebbe il *codominio* della funzione: l'esecuzione di una funzione produce sempre un risultato che si chiama valore di ritorno, in questo caso il valore di ritorno sarà un tipo *int* che è un sottoinsieme finito dei numeri interi (**I**). Poi abbiamo il nome della funzione che si può assegnare liberamente seguendo alcune regole che poi vedremo, in particolare questa funzione deve sempre essere presente in ogni programma eseguibile, cioè che possa essere lanciato in esecuzione. Esistono programmi, o meglio *codici oggetto* che si possono scrivere in C e compilare correttamente ma non sono eseguibili. Essi contengono le cosiddette librerie di funzioni e possono essere linkati da altri programmi eseguibili. Ora limitiamoci ai programmi eseguibili, per essi la main è obbligatoria. Poi, tra parentesi tonde, obbligatorie, possiamo avere una lista di parametri, lo vediamo tra poco. Dopo le parentesi tonde abbiamo l'apertura e la chiusura delle parentesi graffe e in mezzo si scriverà il codice C che si desidera.

Qualsiasi cosa che scriviamo nel codice sorgente quando programmiamo deve sempre appartenere ad una delle seguenti categorie:

- parola chiave (keyword),
- identificatore,
- costante,
- stringa letterale,
- simbolo.

Rispetto a questa classificazione abbiamo che: `int` è una parola chiave, `main` è un identificatore, `(`, `)`, `{` e `}` sono simboli. Tutti questi si chiamano *token* o in italiano *lessemi*. Nella teoria dei linguaggi, per la quale consiglio il libro di Carlucci Aiello e Pirri che trovate in bibliografia (un po' tosto ma completo!), il concetto di lessema è fondamentale, approfonditelo se volete capire bene come lavora un compilatore. Per ora è importante capire che esiste questa categorizzazione e che le regole sintattiche si basano su di essa. Faccio notare che questa è una caratteristica di tutti i linguaggi generabili da una *grammatica ricorsivamente numerabile* e sono quelli che possono essere riconosciuti da una Macchina di Turing, ricordate? Bene, detto questo proviamo a scrivere il primo programma che, come nella migliore tradizione, sarà il noto 'Hello World!' o per gli italianifoni come me, 'Ciao Mondo!'. Vediamo prima il codice poi come compilarlo:

### Listato 2.6 ciaomondo.c

```
#include <stdio.h>
int main()
{
    printf("Ciao Mondo!");
    return 0;
}
```

La direttiva `#include <stdio.h>` è una istruzione per il *preprocessore* cioè un analizzatore che agisce prima del compilatore per svolgere alcuni compiti "di servizio". Nel caso che stiamo vedendo, il preprocessore si occuperà di andare a cercare nel filesystem il file `stdio.h` a cui ci si riferisce come *header file*, cioè un file che contiene le intestazioni di funzioni. A noi serve includere detto file perché contiene le intestazioni delle funzioni per accedere ai dispositivi di input e output; queste sono funzioni della libreria standard del C, significa che ogni compilatore ANSI C

deve metterle a disposizione del programmatore, quindi funzioni già scritte pronte all'uso, un po' come il take-away. Segue poi la funzione `main` che abbiamo già visto ma questa volta al suo interno abbiamo l'istruzione `printf("Ciao Mondo!")`. Essa è costituita da un token identificatore `printf` e dal token stringa letterale "Ciao Mondo!", il cui valore è: **Ciao Mondo!**, senza i doppi apici che si usano per delimitare il valore della stringa stessa e che fanno parte della stringa letterale ma non del valore. L'istruzione rappresenta una funzione e si dice che è una chiamata alla funzione, le parentesi tonde sono token simboli che servono per identificare l'argomento della funzione. Ogni istruzione termina con il token simbolo punto e virgola: `;`. Dopo la chiamata alla `printf` segue l'istruzione `return` che è una keyword token del linguaggio. Essa corrisponde alla `ret` vista nell'assembly.

## 2.5 Compilazione

Proviamo adesso a editare, compilare ed eseguire il programma del listato appena visto. Con l'editor di testo usato per i codici assembly, editiamo le istruzioni C viste sopra e salviamo il file nella solita cartella chiamandolo `ciaomondo.c`. Ora dobbiamo compilarlo. In questo testo farò sempre riferimento al compilatore GNU/GCC che è disponibile presso la fondazione GNU (<https://gcc.gnu.org/>) che è disponibile anche per Windows (<http://www.mingw.org/>) e Mac una volta installato Xcode. Comunque in questo testo a parte la specifica sintassi di compilazione (cioè i comandi per compilare) non faremo uso di caratteristiche specifiche del compilatore GCC, per cui qualsiasi altro compilatore C va benissimo.

La compilazione avviene in più fasi:

1. preprocessig,
2. compilazione,
3. assemblaggio,

## 4. linking

e il GCC permette, volendo di separare queste fasi, ma noi non lo faremo, ricordiamoci che il nostro scopo è imparare i fondamenti delle reti neurali, poi chi è interessato troverà tempo e modo di approfondire questi argomenti.

Vediamo ora come compilare:  
Apriamo una shell dei comandi e portiamoci sulla cartella di lavoro e da qui digitiamo:

### **Compilazione ciaomondo.c**

```
~/Documents/nn> gcc -o ciaomondo ciaomondo.c  
~/Documents/nn> chmod u+x ciaomondo  
~/Documents/nn>./ciaomondo
```

Il risultato sullo schermo deve essere Ciao Mondo!

Adesso è interessante andare a vedere come il compilatore ha compilato in assembly il nostro programma in C.

Come abbiamo visto l'output a video del programma in C è equivalente a quello dell'output di ciao.asm del listato 2.1, ma ci chiediamo cosa si possa dire dell'assembly. Il compilatore GCC possiede l'opzione `gcc -S <filename.c>` che permette di produrre solo il file assembly che troveremo nella stessa cartella come `<filename.s>`. Visto che è capitata l'occasione, spieghiamo la sintassi usata qua sopra. Se si scrive un comando e si lascia un token tra parentesi angolari `<mytoken>` è perché indica che il valore preciso può cambiare in base ad un scelta dell'utente: in questo caso il comando per compilare va bene con qualsiasi nome di file, per questo ho scritto `<filename>`. Ciò detto vediamo qui sotto il codice assembly di ciaomondo.c:

```
.file "ciaomondo.c"  
.text  
.section .rodata
```

```

.LC0:
.string "Ciao Mondo!"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
leaq .LC0(%rip), %rdi
movl $0, %eax
call printf@PLT
movl $0, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (Ubuntu 7.3.0-16ubuntu3) 7.3.0"
.section .note.GNU-stack,"",@progbits

```

I due codici sono senza dubbio diversi: `ciao.asm` è più semplice e conciso di `ciaomondo.s` e per questo ci sono due ragioni e un'osservazione. Partiamo dall'osservazione. Nella programmazione il dono della sintesi non è sempre una buona cosa, infatti ci sono programmi che si possono scrivere in pochissime righe, ma che poi causeranno un consumo importante di risorse della CPU durante la loro esecuzione, mentre altri,

implementando un algoritmo più complesso e lungo da scrivere risultano più economici in fase di esecuzione.

Veniamo alle ragioni delle differenze tra i codici. La prima è che i due assemblatori GCC e NASM usano due sintassi leggermente diverse per l'assembly. GCC è in realtà solo un compilatore e usa al suo interno un assemblatore, sempre del progetto GNU, che si chiama GAS (da GNU e Assembly). GAS, in origine, supportava solo la sintassi detta di AT&T mentre NASM nasce sulla sintassi Intel®. Quindi per confrontarle propriamente bisogna studiarsi bene entrambe le sintassi, ma sintassi a parte, la differenza che ci preme notare è la chiamata alla funzione `printf` che vediamo nell'istruzione `call printf@PLT`. Nel codice `ciao.asm` non abbiamo usato nessuna funzione, tranne la `sys_write` finale, inevitabile per accedere ai *device* di output (device è un termine molto usato per riferirsi alle periferiche). A questo uso continuo di funzioni ci dobbiamo abituare e anzi dobbiamo sfruttarlo al meglio perché sono state introdotte nel linguaggio C per renderlo compatto e altamente comprensibile.

## 2.6 La memoria e le variabili

Nei codici assembly abbiamo usato la memoria per memorizzare i dati su cui l'algoritmo doveva operare. Per far riferimento ad essi, abbiamo visto che si deve usare l'indirizzo del byte di memoria in cui sono memorizzati e, nel caso il dato richieda più di un byte di memoria, esso sarà disposto su celle (byte) contigue l'una all'altra. In questo caso, per accedervi bisogna usare l'indirizzo della prima. Abbiamo detto che inserire in un programma assembly l'indirizzo numerico di un byte di memoria può essere sconveniente. Per questo motivo, le celle si etichettano con delle label (che poi in italiano suonerebbe come: "le celle si etichettano con delle etichette!") a cui si fa riferimento dal codice. Esse vengono poi trasformate in indirizzi numerici per accedere alla

memoria.

### Nomi e variabili

Nel C, l'esigenza è la stessa: usare la memoria per memorizzare dati, ma la strategia è diversa. Nel C si usa dare un nome ai byte di memoria che verranno usati per memorizzare un certo dato, e tale nome si chiama variabile. La sintassi per dichiarare l'uso di una variabile nel C è la seguente:

```
type variable_name;
```

type è una keyword del C e serve a stabilire il tipo di variabile che si vuole dichiarare, essa va scelta tra i tipi disponibili che sono:

1. char
2. int
3. float
4. double

### Il tipo della variabile

La scelta del tipo comporta due cose. La prima è il numero di byte (ad indirizzi contigui) che sono riservati per memorizzare la variabile: un char occupa un byte, un int ne occupa almeno due ma normalmente quattro, un float almeno quattro, un double fino ad otto. I tipi char e int servono a memorizzare dati di tipo numerico intero, cioè senza virgola, mentre il float e il double quelli con la virgola, definiti "in virgola mobile". Facciamo un esempio: supponiamo di dover scrivere un programma che realizza una rete neurale artificiale composta da 10 neuroni di ingresso e 8 neuroni di uscita. Anche se ancora non sappiamo nulla delle reti neurali, è intuitivo che il numero di neuroni servirà a descrivere e parametrizzare l'algoritmo del programma per cui è necessario memorizzare questi numeri in due variabili. Un neurone o c'è o non c'è, non ce ne può essere metà (almeno a livello informatico) per cui non avrebbe senso scrivere qualcosa come: numero\_neuroni = 3.2 pertanto, quello che serve è di memorizzare questi numeri in variabili di tipo intero e la scelta si

strige ad usare un tipo `char` o `int`. In più i neuroni non possono essere in numero negativo, allora si può modificare il tipo usando un *modificatore* che, a livello di linguaggio è un token di tipo keyword, che specifica che la variabile non ha segno (quindi segno positivo), la keyword è: `unsigned`. La scelta tra `char` e `int` dipende dal numero massimo di neuroni che penso di dover usare. Se sono pochi e credo che rimangano pochi potrei usare anche un `char`, che di solito è un solo byte, quindi può essere usato per rappresentare numeri da 0 a  $2^8 - 1$  (255), altrimenti userò un `int` che mi assicura di poter memorizzare fino a  $2^{16} - 1$  (65535).

### Tipi e dimensioni

Ci si domanderà cosa è questa incertezza riguardo alla dimensione in memoria dei tipi numerici. Questo è proprio uno degli aspetti relativi alla capacità di un linguaggio di programmazione di "astrarsi" rispetto ai dettagli architetturali della piattaforma su cui opera. Come abbiamo visto la ALU usa i registri per eseguire le operazioni aritmetiche. Se il registro più capiente di una data architettura, facciamo l'esempio di un micro-controller, è di 16 bit, eseguire somme tra operandi maggiori di 16 bit diventa oneroso in termini di operazioni della CPU, per cui il C, su tale architettura potrebbe essere implementato in modo da limitare l'uso degli interi a valori memorizzabili direttamente nei registri, quindi a 16 bit o appunto due byte. Lo scopo dei linguaggi è proprio quello di farci prendere distanza dall'architettura, ma il C lo fa mantenendo una porta aperta, infatti il C prevede l'operatore `sizeof(<type>)` che permette di conoscere la dimensione in byte di un dato tipo di dato sull'architettura in esecuzione. Anche su questo ci sarebbe da approfondire, il `sizeof` restituisce la dimensione del tipo dato in unità di `CHAR_BIT`, che è ovunque 8, quindi in unità di byte, ma lascia una porta aperta ad architetture che usino un differente numero di bit per rappresentare un `char`. Questi aspetti del

linguaggio sono molto interessanti perché ci fanno capire quanto esso ci permetta di programmare in modo efficiente e veloce pur senza rinunciare al controllo dell'architettura. Comunque non possono essere trattati qui nel dettaglio e, come già suggerito, rimando ai testi specifici indicati in bibliografia.

#### Dichiarazione delle variabili

Tornando ai neuroni, nel programma dovremo dichiarare due variabili per memorizzarne il numero e lo faremo come segue:

```
#include <stdio.h>
```

```
/* ANN: Rete Neurale Artificiale */  
int main()  
{  
    unsigned int n_in; //Numero di neuroni di input  
    unsigned int n_out; //Numero di neuroni di output  
    return 0;  
}
```

Prima della funzione `main` e a fianco delle dichiarazioni delle variabili ho aggiunto dei commenti, in C ci sono due modi diversi di fare commenti e qui li ho mostrati entrambi. Dopo aver dichiarato le variabili è il momento di assegnare ad esse dei valori. L'assegnamento di un valore ad una variabile può essere fatto in qualsiasi momento, sia in fase di dichiarazione: `unsigned int n_in=10;` che successivamente dopo la dichiarazione: `n_in=10;`, ma, come si nota, dopo aver dichiarato una variabile, il suo tipo non va più ripetuto. Vediamo un esempio di codice completo e compilabile in cui dichiariamo, assegnamo e usiamo le variabili:

```
#include <stdio.h>
```

```
/*  
 * ANN: Rete Neurale Artificiale  
 */
```

```
int main()
{
    unsigned int n_in; //Numero di neuroni di input
    unsigned int n_out; //Numero di neuroni di output
    n_in=10;
    n_out=8;
    unsigned int n_tot;//Numero totale di neuroni
    n_tot=n_in+n_out;
    return 0;
}
```

Suggerisco di dare dei nomi "temporanei" a questi frammenti di codice e provare a compilarli come abbiamo visto prima. Leggere è importante, ma provare a scrivere il codice, trovare errori di battitura, correggere e ricompilare, ha un altro gusto.

Assegnamento di un valore ad una variabile

Prima della return abbiamo scritto un'espressione aritmetica:  $n\_in+n\_out$  e ne abbiamo assegnato il risultato alla variabile  $n\_tot$ . È importante notare che si tratta di un assegnamento e **non** di una equazione. Nel C, il simbolo `=` è usato per assegnare un valore ad un'espressione, per esempio: `n_tot=n_in+n_out;` mentre il simbolo `==` è usato per verificare l'uguaglianza tra due espressioni. Ricordate l'operazione `cmp` dell'assembly? Bene, l'operatore `==` del C si basa proprio su questa operazione, esegue il confronto tra due espressioni e verifica il flag opportuno nel registro dei falg (quale di preciso dipende dall'architettura, noi abbiamo visto quello per la Intel®64). Anche l'operazione di confronto ha un valore in C, esso è un valore booleano, cioè 0 oppure 1. Se confrontiamo due espressioni i cui valori sono uguali l'espressione vale 1, se invece sono diversi l'espressione vale 0. Per esempio in C, l'espressione:

```
1==2; //vale 0
```

come suggerito nel commento ha valore 0 (booleano FALSE), mentre l'espressione:

```
2==2; //vale 1
```

Questa caratteristica ci slega notevolmente dall'uso del registro dei flags, che come si sarà capito non è molto intuitivo.

Per procedere oltre e vedere un po' di esempi che si possano compilare, completiamo l'analisi della funzione printf vista prima e scopriamo che possiamo anche stampare il valore delle variabili (ma in fondo lo avevamo già visto esaminando la sys\_call). Scriviamo allora il codice seguente:

### **Listato 2.7** variabili.c

```
#include <stdio.h>
/*
 * ANN: Rete Neurale Artificiale
 */
int main()
{
    unsigned int n_in; //Numero di neuroni di input
    unsigned int n_out; //Numero di neuroni di
    output
    n_in=10;
    n_out=8;
    unsigned int n_tot;//Numero totale di neuroni
    n_tot=n_in+n_out;
    printf("N. di input %u, N. di output %u, totale
    %u\n",
    n_in,n_out,n_tot);
    return 0;
}
```

### **Compilazione** variabili.c

```
~/Documents/nn> gcc -o variabili variabili.c
~/Documents/nn> chmod u+x variabili
~/Documents/nn>./variabili
```

Provate a editare e compilare il codice, poi lanciatelo e dovreste ottenere questo output a video:

N. di input 10, N. di output 8, totale 18

Analizziamo come è stato cambiato l'argomento della funzione per giungere a questo risultato. La funzione printf usa una caratteristica del C che noi non indagheremo, ed è la possibilità di definire delle funzioni che accettano un numero variabile di argomenti. Già, di questo non abbiamo ancora parlato: prima di usare una funzione, questa deve essere stata definita in un codice, quindi:

- deve essere specificato il tipo del valore di ritorno,
- il nome della funzione stessa (che in questo caso è printf),
- la lista dei parametri, detti parametri formali (in matematica si chiamano l'argomento della funzione).

La funzione può essere definita in un codice (programma) che sia scritto da noi, oppure che faccia parte di una libreria.

Parametri formali e parametri *attuali*

La maggior parte delle funzioni viene definita assegnando esattamente la lista dei parametri formali, ad esempio posso definire la funzione int somma\_tre\_addendi(int a, int b, int c); dove le tre variabili intere a, b e c sono proprio i parametri formali della funzione. Come dicevamo, la maggior parte delle funzioni presenta una lista dei parametri rigida che deve essere rispettata nel momento in cui si chiama la funzione altrimenti si ottiene un errore di compilazione; però il C permette anche di specificare che una funzione verrà chiamata con un numero arbitrario di parametri, questo è appunto il caso della printf. Esaminiamo quindi con quali parametri "attuali" viene chiamata la funzione, essi sono:

1. "N. di input %u, N. di output %u, totale %u\n"
2. n\_in

3. n\_out

4. n\_tot

Il primo è una stringa letterale che però segue un formato preciso, cioè alterna del testo a dei simboli costituiti dal simbolo di percentuale concatenato ad una lettera. Questi sono chiamati specificatori di conversione e hanno il ruolo di "segna posto". Se osservate l'output del programma vedrete che al loro posto ci sono i valori che nel programma abbiamo assegnato alle variabili n\_in, n\_out e n\_tot infatti la funzione printf sostituisce agli specificatori il valore delle variabili passate come argomento. Questi sono inseriti di seguito dopo la stringa tra apici che è detta stringa *di formato*. La regola è semplice: il numero di variabili passate dopo la stringa di formato deve corrispondere al numero di specificatori di conversione. Le variabili vengono stampate nell'ordine in cui sono inserite nella chiamata, quindi in questo caso n\_in, n\_out e n\_tot. C'è ancora una cosa da sapere riguardo agli specificatori, essi non solo specificano la posizione in cui deve essere stampato il valore della variabile, ma anche il tipo di dato, per esempio il formato %u informa la funzione che si tratta di un intero senza segno.

Ora che abbiamo introdotto la printf possiamo iniziare a usarla per spiegare meglio la logica in un programma scritto in C. Torniamo anzi tutto alle espressioni booleane e proviamo a fare un esempio. Aggiungiamo un carattere di conversione alla stringa di formato e modifichiamo la chiamata di stampa per ottenere che il programma stampi a video un valore che indichi se il numero di neuroni di input è uguale o diverso dal numero di neuroni di output. Modifichiamo come segue la chiamata alla funzione:

```
printf("N. di input %u, N. di output %u, totale %u,  
uguali:%i\n",n_in,n_out,n_tot,n_in==n_out);
```

Provate a ricompilare e dovete vedere l'output seguente:

```
N. di input 10, N. di output 8, totale 18, uguali:0
```

dove lo 0 è il valore calcolato della espressione booleana `n_in==n_out`.

### Espressioni booleane

Le espressioni booleane sono fondamentali nel C perché è su di esse che si basano i costrutti sintattici `if`, `if-else`, `for`, `while` e `do-while` che servono a controllare il flusso.

Abbiamo visto che nel linguaggio C le espressioni matematiche e logiche (booleane) sono costruite con l'uso delle variabili, che in termini di token sono degli identificatori, degli operatori matematici come il + e quelli logici come l'operatore di confronto ==, questi ultimi due in termini di token sono simboli, poi ancora le espressioni si costruiscono usando i numeri 8, 110, 121.12 ecc, questi in termini di token sono costanti.

Abbiamo chiarito che salvo casi particolari (cioè usando il modificatore `register`) il compilatore associa alle variabili una precisa locazione di memoria. I parametri formali delle funzioni e tutte le variabili dichiarate all'interno di una funzione sono memorizzate nel segmento stack della memoria o nei registri, così come abbiamo visto nella sezione dedicata all'assembly. Esse risultano accessibili e *visibili* solo dal codice scritto nella funzione stessa e, un'altra funzione può usare una variabile con lo stesso nome senza che si crei ambiguità. Si chiamano queste *variabili locali*. Le variabili locali si differenziano dalle variabili *globali* che si possono dichiarare all'esterno di ogni funzione, normalmente prima della funzione `main`. Queste ultime (globali) sono allocate nel segmento `data`, lo stesso in cui abbiamo definito l'etichetta `messaggio` nel primo esempio in assembly. Tale segmento è visibile a tutte le funzioni e quindi se una variabile è dichiarata in modo globale essa potrà essere utilizzata da ogni funzione.

Va detto che in generale l'uso di variabili globali nel C non è una

buona pratica perché porta facilmente ad errori o comunque a scrivere codice scarsamente comprensibile.

### Indirizzo di una variabile

Il valore di una variabile è memorizzato in modo binario nella memoria ad un preciso indirizzo. Il C permette di ottenere tale indirizzo e quindi di accedere al medesimo valore anche senza l'uso della variabile. Facciamo subito un esempio:

#### Listato 2.8 puntatore.c

```
#include <stdio.h>
/*
 * Variabili e indirizzi di memoria
 */
int main()
{
    int n_in=8;
    int* p;
    p=&n_in;
    *p=10;
    printf("Neuroni di input=%d\n",n_in);
}
```

#### Compilazione puntatore.c

```
~/Documents/nn> gcc -o puntatore puntatore.c
~/Documents/nn> chmod u+x puntatore
~/Documents/nn>./puntatore
```

Nel codice qui sopra, viene dichiarata una variabile intera (`n_in`) e contestualmente le viene assegnato il valore 8. Nella riga sotto, viene dichiarata una variabile *puntatore* (`p`), o più semplicemente un puntatore di tipo intero. Volendo si potrebbe valutare l'analogia dei puntatori con le label usate nell'assembly ma non è proprio

corretto, il concetto di puntatore si appoggia ovviamente a tutte le strutture dell'assembly ma è un concetto proprio del C. Nella riga successiva si assegna al puntatore p l'indirizzo della variabile n\_in.

L'indirizzo assegnato corrisponde al vero indirizzo in memoria del primo byte della variabile n\_in e può essere usato per accedervi sia in lettura che in scrittura. Si deve notare che un indirizzo è relativo ad un solo byte, quindi è naturale chiedersi se le operazioni di lettura e scrittura della memoria coinvolgano solo quel dato byte o anche i successivi. La risposta è che questo dipende dal tipo di puntatore con cui si accede alla memoria. Se, come nel caso in questione, il puntatore p è di tipo int, le operazioni sulla memoria usando p coinvolgeranno i primi quattro byte (quattro su un architettura Intel®64). Se invece avessimo dichiarato p di tipo char allora avrebbero coinvolto solo il primo. Questo è una delle caratteristiche che va sotto il nome di *aritmetica dei puntatori*.

Vediamo alla riga successiva che alla memoria puntata da p viene assegnato il valore 10. Questa operazione modifica realmente il contenuto della memoria a cui si riferisce la variabile n\_in, infatti, come si può provare compilando ed eseguendo il codice, il valore della variabile essa risulta 10 e non più 8.

## 2.7 Strutture iterative e di controllo

### nel linguaggio C

Finora abbiamo trattato piccole frazioni di codice in cui il flusso di esecuzione è stato lineare, in termini di PC potremmo dire che è stato sempre e solo incrementato di 4 passi alla volta! Come abbiamo già visto nella sezione dell'assembly, la maggior parte degli algoritmi prevede istruzioni che possono essere eseguite in base a precise condizioni oppure istruzioni che devono essere

ripetute per un determinato numero di volte.

Nell'assembly il controllo del flusso si basa sull'uso dell'istruzione `cmp` che modifica il registro dei flags e poi sul salto condizionato `jcc` che modifica il contenuto del PC. Nel C si usano dei costrutti sintattici "più leggibili" ma che ovviamente vengono poi compilati in istruzioni assembly `cmp` e `jcc`.

### La condizione if

il costrutto `if` serve a condizionare l'esecuzione di un blocco di codice in base al valore assunto da una condizione booleana. Di seguito vediamo un diagramma che rappresenta il flusso di esecuzione delle istruzioni in base al valore logico (booleano) assunto da una generica condizione. Il simbolo a rombo, o a losanga, si usa per rappresentare appunto le condizioni, le frecce invece rappresentano il flusso del codice mentre i rettangoli rappresentano le istruzioni.

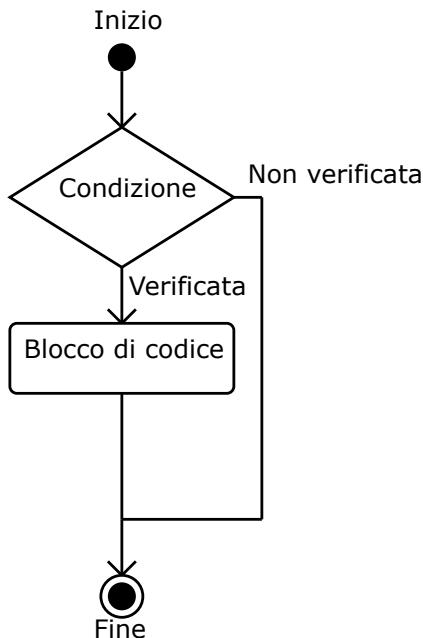


Fig. 2.9 - Diagramma di flusso del costrutto `if`.

Analizziamo il diagramma di figura 2.9. Il flusso di esecuzione delle istruzioni parte dal cerchio con l'etichetta inizio. Sia chiaro che quello è un inizio relativo, non significa che prima non c'è stato flusso, solo lo consideriamo da quel punto perché quanto avvenuto prima non ci interessa ai fini di esemplificare il costrutto if. Il flusso prosegue al blocco condizionale. Da lì, se la condizione risulta vera si scende e si esegue il blocco di istruzioni e si giunge al simbolo di fine, per il quale vale la stessa considerazione fatta per il simbolo di inizio. Se invece la condizione è falsa, il flusso salta il blocco di istruzioni e giunge direttamente a fine.

Nel linguaggio C, questo controllo si ottiene appunto attraverso il costrutto if che ha la sintassi seguente:

```
/*Inizio: flusso prima dell'if*/  
  
if(condizione)  
{  
    ...;//istruzioni  
}  
/*Fine: flusso dopo l'if*/
```

dove *condizione* è una qualsiasi espressione booleana e *istruzioni* sono un numero arbitrario di istruzioni scritte sempre in linguaggio C. Quando si programma in C, bisogna ragionare come se venissero eseguite le istruzioni del linguaggio C. Come ormai chiarito, questo non è vero, il compilatore trasforma le istruzioni C in istruzioni assembly poi le assembla in linguaggio macchina, ma per ragionare sul C, noi penseremo che sono le istruzioni C ad essere eseguite e ci concentreremo sulla loro controparte assembly solo se si dimostra necessario.

Analizziamo quindi il flusso di istruzioni C. Esso parte dal punto commentato come /\*Inizio\*/, poi viene valutata la condizione

che deve essere una espressione booleana. In realtà per il C va bene qualsiasi espressione numerica. Il C tratterà come vera ogni espressione che risulti diversa da 0 e falsa se è uguale a 0. Se la condizione è vera allora verranno eseguite in successione tutte le istruzioni racchiuse tra le graffe, poi (importante) il flusso riprenderà dal punto commentato come `/*Fine*/`, altrimenti, se la condizione è false, il flusso passerà direttamente al punto `/*Fine*/` saltando il blocco di istruzioni.

Vediamo un esempio compilabile. Partendo dal codice precedente, vogliamo che il programma ci avverte nel caso in cui il numero di neuroni di input sia diverso da quello di neuroni di output, altrimenti, in caso contrario, proceda con il flusso di esecuzione. L'espressione che verifica questa condizione è `n_in!=n_out` e va inserita tra parentesi tonde dopo la keyword `if`. Vediamo il codice:

**Listato 2.9** condizione.c

```
#include <stdio.h>
/*
 * ANN: Rete Neurale Artificiale
 */
int main()
{
    unsigned int n_in; //Numero di neuroni di input
    unsigned int n_out; //Numero di neuroni di
    output
    n_in=10;
    n_out=8;
    if(n_in!=n_out)
    {
        printf("N. di input diverso da N. di
        output\n");
    }
}
```

```

unsigned int n_tot;//Numero totale di neuroni
n_tot=n_in+n_out;
printf("N. di input %u, N. di output %u, totale
%u\n",n_in,n_out,n_tot);
return 0;
}

```

### **Compilazione condizione.c**

```

~/Documents/nn> gcc -o condizione condizione.c
~/Documents/nn> chmod u+x condizione
~/Documents/nn>./condizione

```

L'espressione `n_in!=n_out` vale 1 se è vero che `n_in` è "non uguale" (diverso) a `n_out` e vale zero se invece le due variabili sono uguali. Per costruire questa espressione abbiamo usato l'operatore C corrispondente al simbolo: `!=` che si ottiene combinando l'operatore logico NOT (in C è dato dal simbolo `!`) e l'operatore relazionale uguale `==`, la loro combinazione è appunto il simbolo `!=`.

Il costrutto `if` può essere completato con l'opzione `else` che come `if` è una keyword del C. Essa serve per specificare un blocco di codice che deve essere eseguito solo se la condizione risulta falsa. Vediamo sempre lo stesso esempio in C:

### **Listato 2.10 alternativa.c**

```

#include <stdio.h>
/*
 * ANN: Rete Neurale Artificiale
 */
int main()
{
    unsigned int n_in; //Numero di neuroni di input

```

```

unsigned int n_out; //Numero di neuroni di
output
n_in=10;
n_out=8;
if(n_in!=n_out)
{
    printf("N. di input diverso da N. di
output\n");
}
else {
    printf("N. di input uguale al N. di output\n");
}
unsigned int n_tot;//Numero totale di neuroni
n_tot=n_in+n_out;
printf("N. di input %u, N. di output %u, totale
%u\n",n_in,n_out,n_tot);
return 0;
}

```

### **Compilazione alternativa.c**

```

~/Documents/nn> gcc -o alternativa alternativa.c
~/Documents/nn> chmod u+x alternativa
~/Documents/nn>./alternativa

```

Consiglio di provare a modificare il valore assegnato alle variabili e valutare gli effetti sul codice, ovviamente dopo averlo ricompilato.

### **Ciclo for**

Per introdurre il costrutto `for` facciamo un piccolo esempio introduttivo di un aspetto delle reti neurali (niente paura però, questo argomento verrà trattato nel prossimo capitolo).

I neuroni, le cellule che costituiscono il tessuto del cervello, sono

collegati tra loro attraverso le sinapsi. L'attività cerebrale consiste nel passaggio di stimoli elettrici da neurone a neurone. Ogni neurone può essere collegato a migliaia di altri neuroni e, il suo "attivarsi" durante una funzione cerebrale dipende dallo stato di attivazione dei neuroni che hanno un collegamento sinaptico con esso. Nel nostro esempio prendiamo un neurone di output e supponiamo sia collegato a tutti i neuroni di input e calcoliamo il suo stato di attivazione semplicemente come la somma di tutti gli stati di attivazione dei neuroni che gli sono collegati. Per rappresentare uno ad uno lo stato di attivazione di ogni neurone di input dovremmo usare tante variabili quanti essi sono, per esempio dichiarando le seguenti variabili:

```
float in_1=0.1;  
float in_2=0.5  
...;  
float in_10=0.3;
```

Come si capirà con il tempo questo approccio non è funzionale. Quando si devono creare un numero  $n$  di variabili omogenee sia nel tipo di dato che nella loro funzione logica rispetto a ciò che rappresentano (in questo caso lo stato di attivazione di un neurone), si usa un *array* che è l'esatto equivalente delle matrici e dei vettori visti nella prima sezione del libro. Gli array possono avere più indici ma noi ci occuperemo solo di array ad un indice, che sono analoghi ai vettori o alle matrici colonna) e di array a due indici, che sono analoghi ad una matrice  $n \times m$ . Il modo di dichiarare un array è analogo alla dichiarazione di una variabile solo che si aggiunge una parentesi quadra per indicare il numero di elementi dell'array.

riscriviamo il codice, visto sopra, come segue:

```
float in[10];  
in[0]=0.1;
```

```
in[1]=0.5;  
....;  
in[9]=0.3;
```

La sintassi per assegnare un dato valore ad un elemento dell'array ricorda quella dell'algebra, dove, dato il vettore **n**, si "assegna" un valore alle componenti usando l'indice che specifica la componente del vettore a cui ci si riferisce. Tra gli array e i vettori c'è quindi una relazione che possiamo indicare come segue:  $\text{in}[1]=0.5 \rightarrow n_2 = 0.5$ .

Come si vede l'indice degli array parte da 0 e non da 1. Il motivo di questo è che il nome dell'array senza indice, in questo caso **in** è un puntatore all'area di memoria in cui vengono memorizzati in modo contiguo (ad indirizzi crescenti) tutti gli elementi dell'array. L'operatore indice, cioè la coppia di parentesi quadre **[]**, serve a far riferimento agli elementi dell'array indicando lo scostamento dall'indirizzo iniziale memorizzato nella variabile array, quindi per accedere al primo elemento dovremo scostarci di 0 byte dall'indirizzo di base, si ha:

$\text{in}[0] \leftrightarrow n_1$

Lo scostamento dall'indirizzo di base viene calcolato moltiplicando l'indice dell'array per il numero di byte occupato dal tipo base, in questo caso il **float**. Segue che la scrittura **in[1]** fa riferimento all'area di memoria di indirizzo

$\text{in}+1 \times \text{sizeof}(\text{float})$

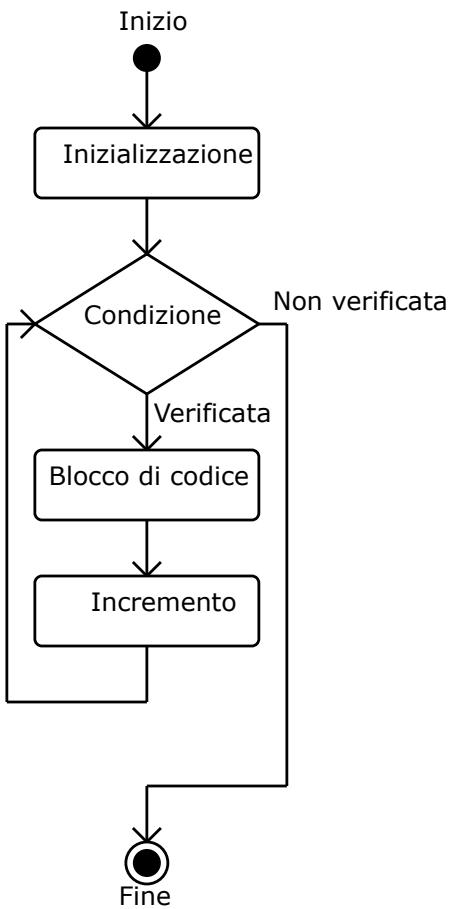
normalmente quattro byte dopo il primo. Bene, chiarito questo torniamo al programma. Vogliamo calcolare il valore di attivazione del neurone di output.

Introduciamo la variabile **float att=0;** e calcoliamola come la somma dei valori di attivazione di tutti i neuroni collegati al neurone di output. Per farlo dobbiamo sommare tutti gli elementi dell'array **in**. A prima vista potrebbe sembrare comodo scrivere la

seguente espressione (già, in C anche gli assegnamenti sono espressioni e hanno un valore):

`att=in[0]+in[1]+in[2]+in[3]+in[4]+in[5]+in[6]+in[7]+in[8]+in[9];`  
però cosa ne pensate di scriverla per una rete che prevede diecimila neuroni di input e diecimila di output? Poi ancora: se il numero di neuroni dovesse diventare variabile, come si potrebbe gestire questa somma non conoscendone l'estensione nel momento in cui si scrive il codice?

La soluzione è quella di iterare la somma usando un ciclo, nel caso specifico useremo un ciclo `for`. Il flusso di questo costrutto è descritto nel diagramma di figura 2.10 che segue qui sotto.



*Fig. 2.10 - Diagramma di flusso del costrutto for.*

Il primo rettangolo corrisponde all'azione di "inizializzazione". Il ciclo for ripete il blocco di istruzioni per un numero di volte che è dato dalla differenza di due numeri: il numero a cui ci si vuole fermare e quello da cui si vuole partire. L'inizializzazione permette (ma non è obbligatorio) di assegnare il numero di partenza. È costume farlo assegnando ad una variabile di tipo int un valore iniziale che dipende dal contesto del programma, diciamo che di solito si parte da 0.

La sintassi è la seguente: `for(int i=0;...;...)`. Si noti che ho dichiarato e inizializzato l'indice `i` all'interno del `for`, questo è

ammesso ma non è obbligatorio, si può usare anche una variabile già dichiarata precedentemente. Poi incontriamo la losanga (rombo) che rappresenta la condizione. Qui è dove stabiliamo il secondo numero con la sintassi seguente: `for(int i=0; i<10; ...)`. Qui viene valutata l'espressione relazionale `i<10`, se essa è verificata, il flusso procede verso il "blocco di codice" altrimenti va alla fine, cioè alla prima istruzione successiva al `for`. A questo punto vengono eseguite le istruzioni che sono inserite tra le parentesi graffe:

```
for(int i=0;i<10;...)
{
    att=att+in[i];
}
/*Fine: flusso dopo il for*/
```

Al termine del blocco (in questo caso una singola istruzione) il flusso passa per il rettangolo "di incremento", qui normalmente viene incrementata la stessa variabile che è stata inizializzata e che si usa nella condizione, però questo non è obbligatorio. Il tipico incremento consiste nell'aumentare di un'unità la variabile usata come contatore, quindi ad esempio: `i++` che è una forma contratta per scrivere `i=i+1` con la quale assegnamo alla variabile `i` il valore che ha aumentato di uno quindi: `for(int i=0;i<10; i++)`.

Come si vede nel diagramma, dopo l'incremento il flusso è condotto ancora verso la losanga e da lì il ciclo riparte. Vediamo adesso un esempio completo e compilabile:

### Listato 2.11 ciclofor.c

```
#include <stdio.h>
/*
```

```

* ANN: Calcolo dell'attivazione
*/
int main()
{
    unsigned int n_in; //Numero di neuroni di input
    unsigned int n_out; //Numero di neuroni di
output
    n_in=10;
    n_out=8;
    float att=0;
    float in[10];
    in[0]=0.1;
    in[1]=0.5;
    in[2]=0.6;
    in[3]=0.11;
    in[4]=0.15;
    in[5]=0.25;
    in[6]=0.98;
    in[7]=0.5;
    in[8]=0.22;
    in[9]=0.3;
    for(int i=0;i<10;i++)
    {
        att=att+in[i];
    }
    printf("Attivazione di output %f\n",att);
    return 0;
}

```

### **Compilazione** ciclofor.c

```

~/Documents/nn> gcc -o ciclofor ciclofor.c
~/Documents/nn> chmod u+x ciclofor
~/Documents/nn>./ciclofor

```

L'inizializzazione, la condizione e l'incremento possono essere manipolati in molti modi, ma non approfondiremo questo aspetto. La cosa che invece voglio sviscerare subito è la possibilità di "annidare" un ciclo dentro l'altro, o come si dice in inglese di scrivere dei *nested loop*.

Proviamo a fare un esempio semplice ma efficace: proviamo a stampare l'intera tavola pitagorica. La ricordate? Bene, dobbiamo scrivere il codice per eseguire 144 moltiplicazioni. Il sistema è semplice, ed è lo stesso che poi useremo per eseguire il prodotto di una matrice per un vettore. Prima scriviamo un ciclo che iteri i cicli su tutte le righe, poi, al suo interno cicleremo su tutte le colonne e per ogni moltiplicazione produrremo la stampa a video. Quindi la costruzione da imparare è quella mostrata di seguito:

```
for(int i=1,i<=12;i++)
{
    for(int j=1;i<=12;j++)
        ...;//il codice
}
```

Il ciclo interno, verrà ripetuto dodici volte, così il blocco di istruzioni all'interno del ciclo più interno verrà ripetuto centoquarantaquattro volte. Di seguito riporto il codice completo per la stampa della tavola pitagorica. Mi sembra già di sentirvi: "Ma cosa è quel \t? Ma cosa è quel \n?" Ah! provate a toglierli, così lo capite da soli! Coraggio, bisogna provare per imparare!

### Listato 2.12 tavolapitagorica.c

```
#include <stdio.h>
/*
 * Stampa della tavola pitagorica
 */
int main()
```

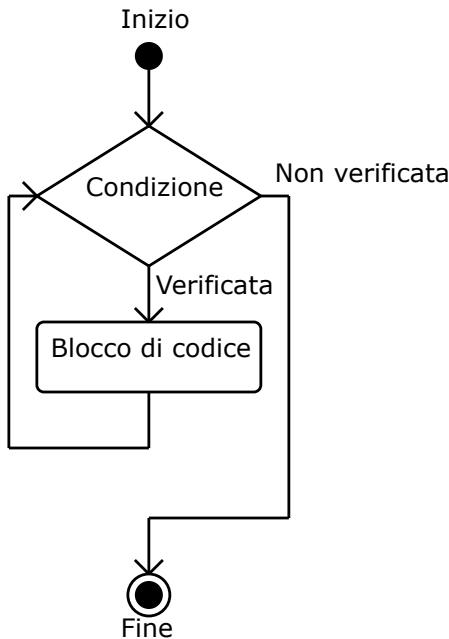
```
{  
    for(int i=1;i<=12;i++)  
    {  
        for(int j=1;j<=12;j++)  
        {  
            printf("%d\t",i*j);  
        }  
        printf("\n");  
    }  
}
```

### **Compilazione** tavolapitagorica.c

```
~/Documents/nn> gcc -o tavolapitagorica  
tavolapitagorica.c  
~/Documents/nn> chmod u+x tavolapitagorica  
~/Documents/nn>./tavolapitagorica
```

### **Ciclo while**

I cicli while e for potrebbero in realtà sostituirsi l'un l'altro, se non che il loro uso appropriato serve a chiarire le intenzioni di chi programma e quindi a rendere il codice leggibile e comprensibile. Il diagramma di flusso è mostrato qui sotto in figura 2.11.



*Fig. 2.11 - Diagramma di flusso del costrutto while.*

Rispetto al ciclo for vediamo che il flusso del while manca dell'inizializzazione e dell'incremento. Si ripete il blocco di codice finché (appunto while in inglese) la condizione è vera. Il costrutto sintattico è molto semplice, si ha: `while(condizione)` dove di nuovo condizione è una espressione booleana. Come prima il blocco di codice da iterare va inserito tra le graffe. Raggiunta l'ultima istruzione viene valutata di nuovo la condizione e, se è vera, il ciclo riparte. Vediamo un esempio semplice:

### Listato 2.13 ciclowhile.c

```

#include <time.h>
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int i;
    time_t t;

```

```
 srand((unsigned) time(&t));
while((i=rand())>RAND_MAX/4)
{
    printf("%d è maggiore di %d\n",i,RAND_MAX/4);
}
return 0;
}
```

### **Compilazione** ciclowhile.c

```
~/Documents/nn> gcc -o ciclowhile ciclowhile.c
~/Documents/nn> chmod u+x ciclowhile
~/Documents/nn>./ciclwhile
```

Provate a compilare il programma e ad eseguirlo un po' di volte. Cosa succede? Vedete che il numero di iterazioni cambia di volta in volta, certo perché stiamo valutando la condizione del while usando una funzione che genera dei numeri *pseudo casuali*. La funzione in questione si chiama rand e si trova nella libreria standard stdlib.h di cui infatti viene incluso l'header nella seconda riga. La funzione srand serve a "rigenerare" la sequenza di numeri casuali, che se no si presenterebbe sempre uguale, e per farlo usa il tempo corrente in millisecondi, che è fornito dalla funzione time. La cosa più interessante di questo codice è come ho mischiato assegnamento e valutazione dell'espressione. La morale della favola è che i cicli while si usano quando a *compile time* non si conosce a priori il numero di iterazioni che saranno eseguite perché dipende da valori che saranno calcolati a runtime. In questo si differenzia dal ciclo for.

## 2.8 Astrazione: le funzioni

Nei paragrafi precedenti abbiamo introdotto i tipi di dato, le variabili, i puntatori, gli array, le espressioni e i costrutti sintattici per il controllo del flusso. Abbiamo usato diverse funzioni della libreria standard, ma a parte la `main` ancora non abbiamo provato a scrivere una *funzione utente* cioè una di quelle funzioni che ogni programmatore deve scriversi "inventandole" da sé per implementare il proprio codice. Fatto salvo per la `main`, l'uso di funzioni utente non è obbligatorio.

Si può scrivere un programma di qualsiasi complessità senza usare altre funzioni. Il motivo per cui nella programmazione furono introdotte le funzioni (già dall'assembly) è di migliorare l'efficienza della programmazione permettendo al programmatore di dividere il programma in più programmi di minor complessità. La divisione del codice in funzioni permette poi di separare le funzioni in moduli diversi e di salvarle su file diversi. Questo a sua volta permette sia di riusare le stesse funzioni per programmi diversi tra loro, sia di non dover ricompilare un intero progetto a causa della modifica di una singola funzione.

Per illustrare l'uso delle funzioni nella programmazione C, riprendiamo il codice con cui abbiamo calcolato lo stato di attivazione e spostiamo il calcolo esplicito all'interno di una funzione utente a cui assegnamo il nome `calcola_attivazione`. Il nome di una funzione è un identificatore e, per essere corretto, deve sottostare ad alcune regole. Deve cominciare con un carattere alfabetico (maiuscolo o minuscolo) o con il segno under score (`_`), può poi continuare sia con caratteri alfabetici che con numeri, o ancora con il segno under score. Esistono diversi sistemi per costruire il nome di una funzione. Io cerco di chiarire sempre l'azione svolta dalla funzione specificando, quando è possibile, un verbo e un complemento oggetto, come in questo caso.

La funzione riceve un array di valori di attivazione di tipo float e restituisce un solo valore di tipo float, per cui il suo *prototipo* sarà il seguente:

```
float calcola_attivazione(float att_neur[]);
```

### Funzioni e prototipi

Il prototipo delle funzioni serve al compilatore per conoscere il tipo e la lista dei parametri prima che esse vengano invocate, per questo, come vedremo ora, il prototipo va inserito prima della chiamata della funzione, oppure, come nel caso delle funzioni definite nella libreria standard, il prototipo può essere definito in un header file che viene incluso nel file principale. Questi aspetti più "ingegneristici" del linguaggio sono molto belli e importanti, ma la loro analisi esaustiva ci porterebbe troppo al di fuori del nostro obiettivo primario, per cui qui ci limitiamo a quegli aspetti del linguaggio che ci permetteranno di scrivere degli esempi concreti di reti neurali artificiali.

Nella definizione della funzione, cioè nel codice in cui scriviamo concretamente cosa fa la funzione, inseriremo la logica con cui deve essere calcolata l'attivazione; di seguito il codice:

```
float calcola_attivazione(float att_neur[]) {  
    float att=0;  for(int i=0;i<10;i++)  
    {  
        att=att+att_neur[i];  
    }  
    return att;  
}
```

Ci sono due aspetti molto importanti del codice che abbiamo appena scritto. Il primo da osservare è l'uso dell'istruzione `return`. Il valore che viene "ritornato" dalla funzione (l'uso di "ritornato" è un po' italiese ma in informatica non si va troppo per

il sottile...) è il valore che assume l'espressione chiamante. Immaginiamo che all'interno della `main` sia presente il codice seguente:

```
...;  
calcola_attivazione(in);  
//Chiamata ...;
```

allora, l'istruzione che chiama la funzione è una espressione, e il suo valore è il valore ritornato dalla funzione. È importante capire bene questo concetto perché le funzioni possono essere usate in molti modi, per esempio all'interno di una condizione. Il secondo aspetto fa parte della programmazione avanzata, ma siccome lo useremo molto nella programmazione delle reti neurali è necessario analizzarlo, si tratta di passare uno o più array come parametri di una funzione.

### Funzioni e stack

Abbiamo chiarito che sia le variabili dichiarate all'interno di una funzione che i parametri della funzione stessa sono allocati nello stack e quindi non sono visibili dalle altre funzioni, in pratica ogni funzione è isolata dalle altre e la comunicazione avviene copiando i valori dei parametri nello stack, ma la modifica di detti valori da parte della funzione chiamata non influisce sui valori delle variabili della funzione chiamante

Supponiamo di avere la seguente situazione: nella funzione `main` abbiamo definito le variabili `a` e `b`, e, senza necessità di farlo, abbiamo assegnato lo stesso nome ai parametri della funzione `foo` (se vi chiedete perchè `foo`, sappiate che `foo`, `bar`, `poo` ecc. sono tipici nomi usati per definire delle funzioni che hanno solo scopo esemplificativo). La funzione `foo` modifica il valore del parametro `a`, ma come si vede compilando ed eseguendo il programma, questo non comporta la modifica del valore della variabile `a` definita nel corpo della `main`. Il motivo, torno a

spiegare, è che al momento della chiamata di foo, sullo stack viene copiato il valore memorizzato dentro la variabile a che si trova sempre nello stack ma ad un altro indirizzo, pertanto, la modifica operata da foo sulla variabile locale a (cioè il suo parametro formale) non influisce sulla variabile locale a dichiarata dentro la main. Segue l'esempio:

**Listato 2.14** foo.c

```
#include <stdio.h>
#include <stdlib.h>

float foo(float a, float b);

int main ()
{
    float a,b,r;
    a=5.1;
    b=4.9;
    r=foo(a,b);
    printf("a=%f, b=%f, r=%f\n",a,b,r);
}

float foo(float a, float b)
{
    a=-b+1;
    return 2.1*a;
}
```

**Compilazione** foo.c

```
~/Documents/nn> gcc -o foo foo.c
~/Documents/nn> chmod u+x foo
~/Documents/nn>./foo
```

Ciò detto, le cose si complicano quando ad una funzione si passa

un array come parametro, infatti sebbene quanto detto rimanga assolutamente vero, bisogna notare che passando un array, si passa di fatto non una copia di tutti gli elementi di cui l'array è composto, ma l'indirizzo del primo elemento. Attenzione, non il valore del primo elemento, il suo indirizzo. La funzione chiamata riceverà quindi una copia dell'indirizzo ma, agendo sugli elementi dell'array potrebbe modificare gli stessi elementi a cui fa riferimento l'array della funzione chiamante. Questo è sia un aspetto positivo che negativo, dipende da quel che si deve fare, la cosa importante è averlo chiaro

Nel caso in questione, la funzione `calcola_attivazione` non modifica il valore degli elementi dell'array, quindi per ora il problema non si pone, ma si porrà presto nel prossimo capitolo. Vediamo l'esempio completo del codice:

### **Listato 2.15** attivazione.c

```
#include <stdio.h>

float calcola_attivazione(float att_neur[]);

/*
 * ANN: Calcolo dell'attivazione
 */
int main()
{
    unsigned int n_in; //Numero di neuroni di input
    unsigned int n_out; //Numero di neuroni di
    output
    n_in=10;
    n_out=8;
    float att=0;
    float in[10];
```

```

in[0]=0.1;
in[1]=0.5;
in[2]=0.6;
in[3]=0.11;
in[4]=0.15;
in[5]=0.25;
in[6]=0.98;
in[7]=0.5;
in[8]=0.22;
in[9]=0.3;

att=calcola_attivazione(in);

printf("Attivazione di output %f\n",att);
return 0;
}

float calcola_attivazione(float att_neur[]) {
    float att=0;
    for(int i=0;i<10;i++)
    {
        att=att+att_neur[i];
    }
    return att;
}

```

### **Compilazione** attivazione.c

```

~/Documents/nn> gcc -o attivazione attivazione.c
~/Documents/nn> chmod u+x attivazione
~/Documents/nn>./attivazione

```

Strutturare il codice in funzioni permette di astrarre i dettagli delle operazioni che si devono compiere sui dati lasciando così alla main

solo l'organizzazione logica del flusso principale dell'algoritmo che stiamo implementando.

Con questo abbiamo visto i fondamenti dell'informatica che sono sufficienti per addentrarci con coraggio nel mondo delle reti neurali artificiali. Dal prossimo capitolo cominceremo a mischiare insieme nozioni di matematica, informatica e neurologia: comincia il divertimento.

### 3. Parte terza: fondamenti di reti neurali

Nei primi paragrafi di questo libro, sono stati introdotti i concetti base per iniziare l'analisi del funzionamento di una rete neurale, prendendo ad ispirazione gli studi fatti sul funzionamento del cervello umano. In questa terza sezione, useremo gli elementi di algebra e di informatica appresi nelle due precedenti per modellare in termini matematici ed informatici una rete neurale che risponda agli stimoli in modo *analogo* a quanto fa una rete neurale biologica.

Arrivati qui, potrebbe essere il caso di rileggere detti paragrafi introduttivi in modo da focalizzare l'attenzione sul concetto di cellula nervosa (neurone) sull'idea che essa possa essere *attivata* in conseguenza di un impulso elettrico e sul concetto di rete, intesa come l'insieme delle interconnessioni tra i neuroni di un certo sistema (per esempio un area cerebrale).

D'ora in avanti dovremo sfruttare questi concetti per creare degli automi di calcolo (programmi) che realizzeranno delle funzionalità concrete, come ad esempio il riconoscimento di un numero manoscritto, per questo è necessario che non esistano dubbi sui concetti di fondo.

#### 3.1 Il modello matematico

Consideriamo due insiemi distinti di  $n$  neuroni. Nel primo insieme indichiamo la configurazione della attività neurale con  $f$ , nel secondo con  $g$ . Supponiamo che ogni neurone del primo insieme sia connesso ad ogni neurone del secondo insieme con una sinapsi di una certa intensità data dal "peso" della connessione. Usiamo la lettera  $j$  per indicare il  $j$ -esimo neurone della configurazione  $f$  e la lettera  $i$  per indicare l' $i$ -esimo neurone della configurazione  $g$ , e  $\alpha(i, j)$  per indicare il peso della sinapsi tra  $i$  e  $j$ .

Il modello di memoria associativa si può ora rappresentare in

termini algebrici come segue:

$$\Delta \mathbf{A} \mathbf{f} = \eta \mathbf{g} \quad (3.1)$$

Nell'equazione (3.1) i vettori  $\mathbf{f}$  e  $\mathbf{g}$  rappresentano le configurazioni  $f$  e  $g$  dei neuroni, mentre la matrice  $\Delta \mathbf{A}$  rappresenta i pesi delle connessioni sinaptiche, mentre  $\eta$  è una quantità scalare. Rispetto al modello astratto della memoria associativa, la (3.1) può essere letta come segue:

- Ci sono due insiemi di neuroni uno di input e l'altro di output.
- L'insieme dei pesi delle connessioni sinaptiche tra i due insiemi è dato dalla matrice  $\Delta \mathbf{A}$
- Lo stato di attivazione dei neuroni di input è dato dal vettore  $\mathbf{f}$ .
- Presentando in input la configurazione  $f$  data dal vettore  $\mathbf{f}$  si ottiene in output la configurazione  $g$  data dal vettore  $\mathbf{g}$ .

Per fissare le idee, facciamo un esempio semplice in cui analizziamo solo l'aspetto algebrico di quanto appena introdotto, poi procederemo vedendo come implementare gli algoritmi algebrici in linguaggio C e, solo allora, passeremo ad un esempio che abbia attinenza con l'elaborazione delle immagini, che è un ambito classico di applicazione delle reti neurali.

Consideriamo allora i due vettori colonna

$$\mathbf{f}^{(1)} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

$$\mathbf{g}^{(1)} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

dove il numero uno tra parentesi (i.e.  $^{(1)}$ ) è un'etichetta per indicare che questa è la prima configurazione di input/output che considereremo (poi ne considereremo altre due) e costruiamo la

matrice  $\Delta \mathbf{A}^{(1)}$  come il prodotto del vettore  $\mathbf{g}^{(1)}$  per il trasposto di  $\mathbf{f}^{(1)}$ :

$$\Delta \mathbf{A}^{(1)} = \mathbf{g}^{(1)} \mathbf{f}^{(1)T} = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad (3.2)$$

Eseguendo il prodotto riga per colonna, si ottiene immediatamente:

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = 2 \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

cioè lo scalare  $\eta = 2$  per il vettore  $\mathbf{g}^{(1)}$ , che appunto può essere interpretato come la risposta del sistema allo stimolo identificato dal vettore  $f^{(1)}$ . Abbiamo quindi costruito un semplice sistema basato sull'idea di memoria associativa che a fronte di un dato input produce un determinato output. La cosa interessante che notiamo è che l'equazione (3.1) non richiede che l'output sia esattamente il vettore  $\mathbf{g}^{(1)}$  ma un qualsiasi vettore che abbia la stessa direzione di  $\mathbf{g}^{(1)}$ . Come si vedrà più avanti, nel secondo paragrafo, questo costituisce il limite principale di questo modello. È naturale chiedersi se il sistema che abbiamo costruito si limiti ad associare i vettori  $f^{(1)}$  e  $g^{(1)}$  o se sia possibile estendere le sue capacità associative. Per rispondere definiamo la coppia  $f^{(2)}$  e  $g^{(2)}$  come segue:

$$\mathbf{f}^{(2)} = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}$$

$$\mathbf{g}^{(2)} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

e, di nuovo, costruiamo la matrice  $\Delta \mathbf{A}^{(2)}$  data dal prodotto del vettore  $\mathbf{g}^{(2)}$  per il trasposto di  $\mathbf{f}^{(2)}$ :

$$\Delta \mathbf{A}^{(2)} = \mathbf{g}^{(2)} \mathbf{f}^{(2)T} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{bmatrix}$$

con

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} = 2 \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

e facciamo lo stesso per i due vettori  $f^{(3)}$  e  $g^{(3)}$  che definiamo come segue:

$$\mathbf{f}^{(3)} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

$$\mathbf{g}^{(3)} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$\Delta \mathbf{A}^{(3)} = \mathbf{g}^{(3)} \mathbf{f}^{(3)T} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

con

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = 1 \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Adesso costruiamo la matrice  $\mathbf{A}$  data dalla somma di  $\Delta \mathbf{A}^{(1)}$ ,  $\Delta \mathbf{A}^{(2)}$  e  $\Delta \mathbf{A}^{(3)}$ :

$$\mathbf{A} = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & -1 \end{bmatrix} \quad (3.3)$$

e verifichiamo subito che:

$$\begin{bmatrix} 2 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & -1 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 0 \end{pmatrix} = 2 \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} \quad (3.4)$$

$$\begin{bmatrix} 2 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & -1 \end{bmatrix} \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix} = 2 \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

$$\begin{bmatrix} 2 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & -1 \end{bmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = 1 \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

quindi abbiamo costruito una matrice ( $A$ ) tale che il suo prodotto per  $\mathbf{f}^{(1)}$  è proporzionale a  $\mathbf{g}^{(1)}$ , per  $\mathbf{f}^{(2)}$  è proporzionale a  $\mathbf{g}^{(2)}$  e infine, per  $\mathbf{f}^{(3)}$  è uguale a  $\mathbf{g}^{(3)}$ . Ovviamente non siamo interessati solo all'aspetto algebrico di quello che abbiamo costruito, ma anche e soprattutto a quello che i vettori e le matrici che abbiamo introdotto stanno rappresentando. Il risultato che abbiamo ottenuto, se riportato nel contesto della memoria associativa da cui siamo partiti, ci dice che è possibile costruire una rete di sinapsi che collega due insiemi di neuroni tali che presentando una tra tre possibili configurazioni di input produce una specifica configurazione di output.

Il risultato ottenuto è senz'altro interessante e incoraggiante, ma prima viene spontanea la domanda al riguardo di quante possibili relazioni input/output possiamo rappresentare con la matrice che abbiamo introdotto. In altre parole qual è il limite di informazioni che possiamo memorizzare correttamente all'interno della matrice. Un'altra domanda che viene sempre spontanea è cosa succede se all'ingresso viene presentata una configurazione di input che non corrisponde esattamente a  $\mathbf{f}^{(1)}$  o a  $\mathbf{f}^{(2)}$  o a  $\mathbf{f}^{(3)}$ ? Bene, daremo delle risposte chiare a queste domande, ma non prima di aver visto come realizzare una implementazione pratica del sistema appena visto usando il linguaggio C.

## 3.2 Il modello in C

Per ora quello che dobbiamo fare è scrivere una procedura che esegua gli stessi calcoli che abbiamo visto nel paragrafo precedente in modo automatico.

Comprendere a fondo i calcoli algebrici è fondamentale, altrimenti l'idea che ci si formerà delle reti neurali rimarrà vaga e non applicabile.

Come prima cosa scriviamo la funzione `main()` che, come abbiamo visto, contiene il codice che sarà eseguito quando attraverso il sistema operativo lanceremo l'esecuzione del programma che stiamo scrivendo.

A questo punto definiamo le variabili per i dati con cui dobbiamo lavorare che sono i sei vettori  $\mathbf{f}^{(i)}$  e  $\mathbf{g}^{(i)}$ , le tre matrici  $\Delta \mathbf{A}^{(i)}$  e la matrice  $\mathbf{A}$ .

Per rappresentare i vettori e le matrici, il tipo di dato che si usa più comunemente nel C è l'array di un tipo primitivo, per esempio nel codice in questione useremo degli array di double.

```
int main()
{
    double f1[3]={1,0,1};
    double g1[3]={1,1,0};
    double DA1[3][3];
    double f2[3]={1,0,-1};
    double g2[3]={1,0,1};
    double DA2[3][3];
    double f3[3]={0,1,0};
    double g3[3]={0,0,1};
    double DA3[3][3];
}
```

Fino qui modo abbiamo dichiarato e inizializzato gli array con i valori di esempio usati nel paragrafo precedente.

Seguendo i passaggi algebrici, ora si deve implementare la (3.2),

quindi il prodotto tensoriale di un vettore colonna per un vettore riga (vedi anche equazione (1.22)).

Sebbene possa sembrare complicato, il codice per eseguire questo calcolo è abbastanza semplice e può naturalmente essere incapsulato all'interno di una funzione.

Come prima cosa, costruiamo il prototipo della funzione come segue:

```
void vector_dot_Tvector (double res[][3],double  
a[3],double Tb[3]);
```

e lo scriviamo prima della main dove scriveremo anche tutti gli altri prototipi necessari.

Analizziamo i parametri di invocazione della funzione per poi passare alla sua definizione.

I due parametri a e Tb rappresentano due vettori che devono essere moltiplicati.

All'interno della funzione, il vettore **a** deve essere moltiplicato per il trasposto del vettore **b** e il risultato del prodotto è memorizzato nell'array a due dimensioni res, che appunto rappresenta la matrice risultato della moltiplicazione tensoriale. Quindi il risultato delle operazioni eseguite da questa funzione non arriva dalla istruzione return, ma si trova memorizzato dentro all'array res dopo l'esecuzione della funzione stessa.

Ora che abbiamo analizzato il prototipo veniamo alla sua definizione. Il compito della funzione è quello di fare esattamente la stessa procedura che si eseguirebbe nel fare a mano la moltiplicazione tensoriale, ovvero, prendere le componenti di a e moltiplicarle per le componenti trasposte di Tb. Vediamo il codice:

```
void vector_dot_Tvector  
(double res[][3],double a[3],double Tb[3])  
{
```

```

for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
        res[i][j]=a[i]*Tb[j];
    }
}

```

Il codice è molto semplice, ma vale sempre la pena di analizzarlo nel dettaglio.

L'implementazione della funzione consiste in due cicli uno innestato dentro l'altro, questo significa che l'operazione interna al secondo ciclo sarà ripetuta tante volte quante il prodotto del numero di esecuzioni del primo ciclo per quelle del secondo, in questo caso  $3 \times 3 = 9$  volte. Ad ogni ripetizione (iterazione) viene eseguito un calcolo, cioè il prodotto dell' $i$ -esimo elemento dell'array  $a$  per il  $j$ -esimo elemento dell'array  $Tb$  e il risultato viene assegnato all'elemento di indice  $i$  e  $j$  dell'array  $res$ . Riassumendo, il ciclo più esterno scorre ad una ad una le righe della matrice risultato e per ogni riga, il ciclo interno calcola gli elementi di ogni colonna.

Quando il ciclo più esterno avrà completato la terza iterazione, il ciclo finirà e, come si vede dal codice, la funzione non tornerà nulla. Bisogna però ricordarsi che quando si passa un array come argomento di una funzione, si sta passando l'indirizzo di memoria in cui tale array è allocato, quindi al termine dell'esecuzione della funzione, il risultato, cioè la matrice  $res$ , sarà accessibile a chi ha chiamato la funzione come vedremo tra breve. Ora che è stata definita la funzione per il prodotto tensoriale, vediamo come usarla all'interno del codice eseguendo il prodotto  $\mathbf{g}^{(1)}\mathbf{f}^{(1)T}$ .

Inseriamo la seguente

```
vector_dot_Tvector(DA1,g1,f1);
```

dopo la dichiarazione delle variabili.

Analizziamo ora cosa succede quando viene eseguito il codice

della main. Non dimentichiamo che le istruzioni C vengono compilate in linguaggio macchina e sono queste che vengono eseguite, ma sussiste una relazione molto stretta tra l'astrazione del linguaggio C e il codice macchina, talmente stretta che per semplicità possiamo immaginare che siano le stesse istruzioni C ad essere eseguite. Detto questo torniamo alla funzione main e vediamo che quando viene eseguita, dopo la dichiarazione delle variabili, il controllo del flusso viene passato alla funzione vector\_dot\_Tvector. Il parametro DA1 è l'indirizzo in RAM del primo dei 72 byte a partire dal quale è memorizzato l'array di dimensione 9xsizeof(double) byte (72 byte sulle architetture attuali). L'esecuzione dell'istruzione

```
res[i][j]=a[i]*Tb[j];
```

assegna il risultato del prodotto all'array res, ma tale array si riferisce esattamente alle stesse celle di memoria indirizzate dall'array DA1, appunto perché nel linguaggio C quando si passa un array come argomento di una funzione, si sta in realtà passando l'indirizzo del primo byte indirizzato dall'array stesso. Al termine dell'esecuzione della funzione, il controllo torna alla main.

Il codice che abbiamo scritto fino qui è il seguente:

```
void vector_dot_Tvector  
(double res[][3],double a[3],double Tb[3]);  
int main()  
{  
    double f1[3]={1,0,1};  
    double g1[3]={1,1,0};  
    double DA1[3][3];  
    double f2[3]={1,0,-1};  
    double g2[3]={1,0,1};  
    double DA2[3][3];  
    double f3[3]={0,1,0};  
    double g3[3]={0,0,1};
```

```

double DA3[3][3];

vector_dot_Tvector(DA1,g1,f1);
}

void vector_dot_Tvector
(double res[][3],double a[3],double Tb[3])
{
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            res[i][j]=a[i]*Tb[j];
        }
    }
}

```

Prima di procedere è opportuno scrivere una funzione per visualizzare gli array con cui stiamo operando. L'idea è quella di stampare il contenuto di ogni elemento dell'array a video usando la funzione `printf` che è stata introdotta precedentemente.

Stamperemo gli array che rappresentano dei vettori allineandone gli elementi verticalmente e gli array che rappresentano delle matrici allineando gli elementi in righe e colonne. Per fare questo è molto comodo poter indicare alla funzione `printf` le coordinate dello schermo in cui stampare. Il linguaggio C non prevede un meccanismo per stampare un carattere in una data posizione nel monitor. Quando si tratta di un'applicazione "a terminale", è conveniente usare delle sequenze di escape che impartiscono al terminale delle istruzioni su come elaborare il testo che si vuole stampare.

Non ho intenzione di dilungarmi troppo su questo argomento, giusto due parole per inquadrarlo.

## **ANSI ed ASCII**

I primi computer disponibili negli anni '70 del ventesimo secolo utilizzavano come monitor un terminale elettronico a caratteri

dotato di una propria logica. La comunicazione tra computer e terminale era limitata ad un set ristretto di centoventotto caratteri detto ASCII (ancora in uso) dei quali i primi trentadue (dal 0 al 31) sono caratteri di controllo non stampabili mentre i restanti novantasei sono stampabili. In particolare il carattere di escape, il cui codice corrisponde al decimale 27 o esadecimale 1b, veniva usato per inviare dal computer al terminale una sequenza per impartire una istruzione al monitor. Per esempio se si desiderava che il monitor stampasse la stringa Hello World! a partire dalla riga 10 e colonna 10 dello schermo, si inviava al monitor la sequenza di caratteri: ESC[10;10HHello World! da notare che la doppia H non è un errore, infatti la prima H è il carattere che conclude la sequenza di escape prima della stringa da stampare mentre la seconda è la H di Hello. In linguaggio C, per inviare al terminale una sequenza come quella appena vista, si può usare la funzione printf che, con una chiamata di sistema, invia al terminale la sequenza ASCII voluta. Per esempio se scriviamo:

```
printf("\x1b[10;10;HHello World!");
```

abbiamo come effetto di stampare Hello World! alla riga 10 e colonna 10 del terminale.

Prima di vedere come le sequenze di escape sono state usate per stampare gli array del programma esempio1.c, manca ancora da spiegare un ultimo concetto.

Con ogni probabilità chi sta usando questo manuale in anni vicini a quello di edizione (seconda edizione 2019) noterà che il suo computer non è collegato ad un terminale, quindi potrebbe chiedersi come il discorso appena fatto trovi una sua applicazione. Il fatto è che sia i sistemi operativi Unix-like, come Linux e Mac OSX, che il sistema Windows, usano degli applicativi di sistema per l'emulazione del terminale.

La Linux console del kernel Linux, il Terminal del Mac OSX e la Win32 console ne sono un esempio. Questi programmi sono dei software che emulano la risposta di un video terminale

dall'interno di un'altra architettura video, come ad esempio X Window. La loro caratteristica è quella di supportare gli standard che sono stati sviluppati per i terminali che emulano, come ad esempio l'ANSI X3.64 che definisce le sequenze di escape (sequenze che iniziano con il carattere ESC) usate, per esempio, per definire la posizione e il colore dei caratteri da stampare.

Vediamo ora come implementare la funzione per il display a video degli array. Tale funzione dovrà permetterci di specificare:

- l'array da stampare
- il numero di righe e di colonne dell'array
- la riga e la colonna corrispondenti al primo elemento dell'array

Il prototipo che usiamo in questa implementazione è il seguente:

```
void print_matrix(double x[],int r, int c,int R,int C)
```

Prima di vedere la definizione della funzione `print_matrix` analizziamo il significato dei parametri del suo prototipo.

Il primo parametro è un array ad un solo indice che usiamo per passare i dati da stampare. I secondi due parametri: `r` e `c`, rappresentano il numero di righe e di colonne con cui vogliamo che sia stampato l'array, mentre gli ultimi due: `R` e `C`, rappresentano la posizione (riga e colonna) nel terminale in cui vogliamo che sia stampato l'array.

Tornando ora al primo parametro, notiamo che, se vogliamo stampare a video un array a due indici, dobbiamo prima *stenderlo*, cioè trasformarlo in un array ad un solo indice. Stenderlo significa mettere in fila l'una dopo l'altra tutte le sue righe in modo da ottenerne una sola. Quindi, se si ha un array di  $n$  righe per  $m$  colonne, il risultato di *stendere* è un array ad un solo indice di  $n \times m$  elementi.

Dobbiamo introdurre una funzione che stenda gli array. È un detto comune che non si possa mai iniziare niente senza dover prima fare qualcos'altro, e vale anche nella programmazione. Il prototipo della funzione che ci serve è la seguente:

```
void matrix_to_vector  
(double w[][3],double v[]);
```

che ci permette di passare alla funzione l'array a due dimensioni e un array (vuoto) ad una dimensione, dove sarà memorizzato il risultato. La definizione della funzione è la seguente:

```
void matrix_to_vector  
(double w[][3],double v[])  
{  
    int r,c;  
    int k;  
    r=c=3;  
    for(int i=0;i<r;i++)  
        for(int j=0;j<c;j++)  
    {  
        k=i*3+j;  
        v[k]=w[i][j];  
    }  
}
```

L'utilizzo di due cicli annidati per accedere a tutti gli elementi di un array bidimensionale è già stato analizzato poco sopra, mentre ciò che abbiamo di nuovo in questo codice, è l'algoritmo che mappa la coppia di indici  $i,j$  nell'indice  $k$  che usiamo per l'array monodimensionale.

Come si vede l'idea è semplice e diretta: per provarla basta disegnare un rettangolo di  $3 \times 5$  quadretti su un quaderno e iniziare a contarli tenendo presente che il primo quadretto, in alto a destra, ha riga e colonna uguale a zero. Da lì in poi, spostandoci verso destra aumentiamo sempre di uno e quando arriviamo al

bordo scendiamo di una riga e torniamo a sinistra (colonna zero). Se si fa la prova è facile vedere che tutti i quadretti della prima riga sono numerati con un indice che corrisponde alla loro colonna, poi, scendendo alla seconda riga (cioè indice di riga uguale ad uno) l'indice di ogni quadretto corrisponde all'indice della sua colonna sommato al numero di quadretti presenti in ogni riga (quindi cinque). In questo modo, quando saremo sul secondo quadretto della seconda riga, il valore dell'indice sarà  $5+1=6$ .

Continuando in questo modo, arrivati alla fine della seconda riga l'indice del quadretto deve essere nove. Scendendo ora nella riga successiva si ripete la procedura di calcolo e, anziché sommare all'indice della colonna il numero di elementi presenti in ogni riga, sommeremo il numero di elementi presenti nelle due righe precedenti, quindi  $5+5=10$ . In questo modo, quando saremo, per esempio sul terzo quadretto, il valore del suo indice sarà  $5\times 2 + 2 = 12$ .

Per stendere l'array abbiamo bisogno di un altro array, vuoto, in cui copiare il risultato: ovviamente non possiamo usare uno degli array che abbiamo già definito per rappresentare i dati del nostro problema, perché altrimenti lo sovrascriveremo.

Quello che dobbiamo fare è allora dichiarare un nuovo array che useremo come *appoggio*. Questo significa che l'array che introduceremo non ha un significato nel modello matematico/neurale che stiamo rappresentando, ma ci serve per scrivere l'algoritmo.

Chiamiamo l'array *v* e allochiamo per lui la memoria corrispondente all'array bidimensionale che dobbiamo stenderci sopra, quindi  $3\times 3 = 9$  double. Fatto questo eseguiamo una chiamata alla funzione `matrix_to_vector`. Il codice si dovrebbe presentare come segue:

```
void matrix_to_vector  
(double w[][3],double v[]);
```

```

void vector_dot_Tvector
(double res[][3],double a[3],double Tb[3]);
int main()
{
    double f1[3]={1,0,1};
    double g1[3]={1,1,0};
    double DA1[3][3];
    double f2[3]={1,0,-1};
    double g2[3]={1,0,1};
    double DA2[3][3];
    double f3[3]={0,1,0};
    double g3[3]={0,0,1};
    double DA3[3][3];
    double v[3*3];

    vector_dot_Tvector(DA1,g1,f1);
    matrix_to_vector(DA1,v);
}

void vector_dot_Tvector
(double res[][3],double a[3],double Tb[3])
{
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            res[i][j]=a[i]*Tb[j];
        }
    }
}

void matrix_to_vector
(double w[][3],double v[])
{
    int r,c;
    int k;
    r=c=3;
}

```

```

for(int i=0;i<r;i++)
    for(int j=0;j<c;j++)
{
    k=i*3+j;
    v[k]=w[i][j];
}

```

Fatto questo, possiamo passare senz'altro a vedere l'implementazione della funzione per stampare gli array:

```

void print_matrix
(double x[],int r, int c,int R,int C)
{
    for(int i=0;i<r;i++)
        for(int j=0;j<c;j++)
    {
        double gl;
        gl=x[i*c+j];
        if(gl>=0)
            printf("\x1b[%d;%dH%0.1lf \x1b[0m ",
            i*2+R,6*j+C,gl);
        else
            printf("\x1b[%d;%dH%0.1lf \x1b[0m ",
            i*2+R,6*j+C-1,gl);
    }
    fflush(stdout);
}

```

In questo codice non c'è più nulla di nuovo, bisogna comunque prestare attenzione a come sono stati calcolati gli indici di riga e colonna.

La riga dell'elemento  $i, j$  è calcolata come:

$i*2+R$

dove l'indice  $i$  viene moltiplicato per due per lasciare una riga di spazio tra gli elementi (che non stiano fitti!), mentre la colonna è calcolata come segue:

$6*j+C$

dove l'indice  $j$  viene moltiplicato per sei perché ogni elemento dell'array occupa orizzontalmente fino a quattro colonne: una colonna occupata dalla cifra per la parte intera, una per il punto decimale, una per la cifra decimale e una per il segno meno, quando è il caso.

Un secondo aspetto che potrebbe incuriosire è la struttura condizionale `if-else`, il cui scopo è di allineare correttamente le colonne dell'array quando i valori sono negativi e quindi sono preceduti dal segno meno. Per capire meglio, consiglio di rimuovere il costrutto e vedere cosa succede quando si presentano dei numeri negativi.

Vista l'implementazione della funzione, non rimane altro che invocarla con la chiamata:

```
print_matrix(v,3,3,10, 1);
```

Nel codice si deve includere l'header della libreria `stdio.h` dove è definito il prototipo della funzione `printf`.

Dopo le modifiche apportate ecco il codice fin qui prodotto:

**Attenzione:** rispetto ai precedenti, questi listati propongono delle funzioni e delle linee di codice che possono estendersi per diversi caratteri. Per mantenerne una buona leggibilità, ho spezzato quasi sempre le dichiarazioni e le chiamate alle funzioni su più righe. Ricordiamoci che in C si può andare a capo e che ogni istruzione termina con il punto e virgola!

### Listato 3.1 `modello_associativo_parziale.c`

```
#include <stdio.h>

void matrix_to_vector
(double w[][3],double v[]);
void vector_dot_Tvector
```

```

(double res[][3],double a[3],double Tb[3]);
void print_matrix
(double x[],int r, int c,int R,int C);
int main()
{
    double f1[3]={1,0,1};
    double g1[3]={1,1,0};
    double DA1[3][3];
    double f2[3]={1,0,-1};
    double g2[3]={1,0,1};
    double DA2[3][3];
    double f3[3]={0,1,0};
    double g3[3]={0,0,1};
    double DA3[3][3];
    double v[3*3];

    vector_dot_Tvector(DA1,g1,f1);
    matrix_to_vector(DA1,v);
    print_matrix(v,3,3,10, 1);
}

void vector_dot_Tvector
(double res[][3],double a[3],double Tb[3])
{
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            res[i][j]=a[i]*Tb[j];
        }
    }
}

void matrix_to_vector
(double w[][3],double v[])
{
    int r,c;
    int k;
    r=c=3;
    for(int i=0;i<r;i++)
        for(int j=0;j<c;j++)
    {
        k=i*3+j;
        v[k]=w[i][j];
    }
}

void print_matrix
(double x[],int r, int c,int R,int C)
{
    for(int i=0;i<r;i++)
        for(int j=0;j<c;j++)
    {
        double gl;
        gl=x[i*c+j];

```

```

if(gl>=0)
    printf("\x1b[%d;%dH%0.1lf \x1b[0m ",i*2+R,6*j+C,gl);
else
    printf("\x1b[%d;%dH%0.1lf \x1b[0m ",i*2+R,6*j+C-1,gl);
}
fflush(stdout);
}

```

**Compilazione** modello\_associativo\_parziale.c

```

~/Documents/nn> gcc -o map
modello_associativo_parziale.c
~/Documents/nn> chmod u+x map
~/Documents/nn>./map

```

L'esecuzione del programma genera l'output seguente:

1.0 0.0 1.0

1.0 0.0 1.0

0.0 0.0 0.0 ~/Documents/nn>

che corrisponde alla matrice ottenuta in equazione (3.2).

Lo stesso codice sviluppato fin'ora può essere ripetuto per ottenere gli array DA2 e DA3.

Rispetto al modello matematico, precedentemente sviluppato, mancano ancora un paio di passaggi.

Il primo consiste nell'implementare una funzione per sommare gli array come richiesto in equazione (3.3). Il prototipo potrebbe essere il seguente:

```

void matrix_plus_matrix
(double res[][3],double m1[][3],double m2[][3]);

```

L'implementazione della `matrix_plus_matrix` non presenta novità: si tratta di iterare su righe e colonne ed eseguire la somma degli elementi dei due array `m1` e `m2` memorizzandone il risultato nell'array `res`.

Il secondo passaggio da completare è quello presentato nella (3.4), cioè il prodotto di una matrice per un vettore, che, in termini di C, si ridurrà al prodotto di un array bidimensionale per un array monodimensionale.

Il prototipo della funzione dovrà prevedere un parametro che rappresenti il risultato del prodotto, uno che rappresenti la matrice ed uno il vettore:

```
void matrix_dot_vector  
(double res[3],double matrix[][3],double vector[3]);
```

Con questa piccola libreria di funzioni che sono state implementate, nella main possiamo eseguire l'esempio visto sopra:

- si assegnano tre vettori di input e i relativi tre vettori di output
- si costruiscono le tre matrici di connessione
- si ottiene la matrice somma delle tre matrici
- si prova che il prodotto della matrice somma per uno dei vettori di input da come risultato  $\eta$  per il vettore di output

### Listato 3.2 `modello_associativo.c`

```
#include <stdio.h>  
  
void matrix_dot_vector  
(double res[3],double matrix[][3],double vector[3]); void  
matrix_plus_matrix  
(double res[][3],double m1[][3],double m2[][3]);  
void matrix_to_vector  
(double w[][3],double v[]);  
void vector_dot_Tvector  
(double res[][3],double a[3],double Tb[3]);
```

```

void print_matrix
(double x[],int r, int c,int R,int C);
int main()
{
    double f1[3]={1,0,1};
    double g1[3]={1,1,0};
    double DA1[3][3];
    double f2[3]={1,0,-1};
    double g2[3]={1,0,1};
    double DA2[3][3];
    double f3[3]={0,1,0};
    double g3[3]={0,0,1};
    double DA3[3][3];
    double v[3*3];
    double A[3][3];
    double r[3];

    vector_dot_Tvector(DA1,g1,f1);
    matrix_to_vector(DA1,v);
    print_matrix(v,3,3,10,1);

    vector_dot_Tvector(DA2,g2,f2);
    matrix_to_vector(DA2,v);
    print_matrix(v,3,3,10,30);

    vector_dot_Tvector(DA3,g3,f3);
    matrix_to_vector(DA3,v);
    print_matrix(v,3,3,10,50);
    //Calcolo A=DA1+DA2+DA3
    matrix_plus_matrix(A,DA1,DA2);
    matrix_plus_matrix(A,A,DA3);
    matrix_to_vector(A,v);
    print_matrix(v,3,3,20, 1);
    //Calcolo A*f1
    matrix_dot_vector(r,A,f1);
    print_matrix(f1,3, 1,20, 22);
    print_matrix(r,3, 1,20, 28);
}

void vector_dot_Tvector
(double res[][3],double a[3],double Tb[3])
{
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            res[i][j]=a[i]*Tb[j];
        }
    }
}

void matrix_to_vector
(double w[][3],double v[])
{

```

```

int r,c;
int k;
r=c=3;
for(int i=0;i<r;i++)
{
    for(int j=0;j<c;j++)
    {
        k=i*3+j;
        v[k]=w[i][j];
    }
}
void print_matrix
(double x[],int r, int c,int R,int C)
{
for(int i=0;i<r;i++)
    for(int j=0;j<c;j++)
    {
        double gl;
        gl=x[i*c+j];
        if(gl>=0)
            printf("\x1b[%d;%dH%0.1lf \x1b[0m ",i*2+R,6*j+C,gl);
        else
            printf("\x1b[%d;%dH%0.1lf \x1b[0m ",i*2+R,6*j+C-1,gl);
    }
fflush(stdout);
}
void matrix_plus_matrix
(double res[][3],double m1[][3],double m2[][3])
{
for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
        res[i][j]=m1[i][j]+m2[i][j];
    }
}
}
void matrix_dot_vector
(double res[3],double matrix[][3],double vector[3])
{
for(int i=0;i<3;i++){
    res[i]=0;
    for(int j=0;j<3;j++){
        res[i]+=matrix[i][j]*vector[j];
    }
}
}

```

### **Compilazione** modello\_associativo.c

~/Documents/nn> gcc -o modello\_associativo  
**modello\_associativo.c**

```
~/Documents/nn> chmod u+x modello_associativo  
~/Documents/nn>./modello_associativo
```

Con questo abbiamo implementato in linguaggio C il modello matematico presentato nel paragrafo precedente.

Il codice ha solo valore didattico, è ridotto al minimo e non è ottimizzato, ma è una prima implementazione di rete neurale capace di riconoscere un vettore di input e produrre un vettore di output.

È opportuno notare che il risultato non è comunque privo di interesse applicativo, infatti, per esempio, le immagini di tipo *raster*, cioè rappresentate in memoria da una griglia di pixel, sono intrinsecamente dei vettori, quindi, scalando opportunamente le dimensioni degli array che sono stati usati, si potrebbe creare un sistema che associa ad una data immagine di input una specifica immagine di output. Per esempio? Supponiamo di avere un database di fotografie di coppie, presentando la fotografia di uno dei partner il sistema potrebbe produrre la fotografia dell'altro.

### 3.3 Rete neurale ispirata dalla biologia

Nel paragrafo precedente abbiamo visto come un modello matematico basato sull'algebra lineare può fornire un esempio di *stimolo/risposta* dove lo stimolo era rappresentato dal vettore **f** e la risposta dal vettore **g**.

In quel contesto non abbiamo però dato una motivazione biologica del modello. In questo paragrafo faremo due cose: per prima cosa forniremo tale motivazione, cioè vedremo che il modello matematico nasce da delle considerazioni sui meccanismi biologici con cui avviene l'associazione tra stimolo e risposta nel cervello animale (per lo meno per quel che è dato di capire), poi, per seconda, vedremo come il meccanismo di associazione tra lo stimolo e la risposta possa essere appreso mediante una

procedura detta appunto di *apprendimento* o *training*.  
Questo è un punto molto importante e molto attuale.

## Il modello *biologico* stimolo/risposta

In questo paragrafo proviamo un primo passo verso la comprensione di come può funzionare il cervello. Il condizionale è sembra d'obbligo, perché per quanto lo si possa studiare, il cervello, come ogni altro oggetto di studio scientifico, rimane sempre una realtà esterna alla nostra mente, quindi anche nel momento in cui esso potrebbe apparirci comprensibile, ricordiamo che ciò che stiamo davvero comprendendo non è la sua vera natura, ma è la natura del nostro pensiero.

Dopo questa doverosa premessa, vediamo che è possibile costruire un modello matematico di alcune capacità cerebrali che possono quindi essere trasformate in formule per essere studiate e sfruttate.

Immaginiamo il cervello costituito da due gruppi di neuroni: un gruppo deputato a ricevere degli stimoli ed un gruppo deputato a generare delle risposte.

Gli stimoli potrebbero essere sia di origine sensoriale, come la vista o il tatto, ma anche di origine *interna* come ad esempio un pensiero o un'idea.

Allo stesso modo, la risposta può essere l'attivazione di neuroni che controllano le aree del movimento, ma anche la produzione di un'idea o di un concetto. Vediamo di seguito qualche esempio:

1. Vedo un serpente e mi spavento: in questo caso posso pensare che i neuroni di input siano quelli legati alla vista, quindi neuroni sensitivi, mentre quelli di output sono i neuroni legati alla classificazione del pericolo, quindi concettuali.
2. Vedo un pettine e penso ai capelli: i neuroni di input sono

ancora quelli della vista, quelli di output sono neuroni concettuali.

3. Penso a casa e mi viene in mente la mamma: i neuroni di input sono concettuali come quelli di output.
4. Mi scotto una mano e la ritraggo: i neuroni di input sono sensitivi, poi c'è un passaggio per i neuroni intercalari (o interneuroni), mentre i neuroni di output sono motori.

Gli esempi sopra non hanno pretesa di precisione rispetto al linguaggio neurologico e biologico, sono esempi terra-terra che hanno il solo scopo di rendere l'idea.

Supponiamo quindi di avere un numero che indichiamo con la lettera  $n$  di neuroni che ricevono lo stimolo e li indichiamo con il simbolo  $s_i$  dove l'indice  $i$  assume valori da 1 a  $n$ , mentre con il simbolo  $r_j$  ( $j$  è un indice come  $i$ ) indichiamo i neuroni di risposta che per ora immaginiamo essere sempre in numero  $n$ . Ora proviamo a figurare un modello delle interconnessioni tra i neuroni di stimolo e quelli di risposta. Per farlo costruiamo una griglia in cui  $n$  linee orizzontali trasmettono gli stimoli  $s_i$ , mentre quelle verticali trasmettono i segnali di risposta (agli stimoli)  $r_j$  e ogni incrocio rappresenta una connessione sinaptica tra il neurone  $s_i$  e il neurone  $r_j$ . Indichiamo le connessioni sinaptiche con il simbolo  $m_{i,j}$ .

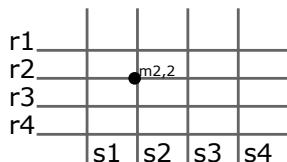


Fig. 3.1 - Le linee verticali rappresentano gli stimoli attivi ai neuroni di input, le orizzontali l'attività dei neuroni di output. Il cerchietto nero indica un collegamento sinaptico.

La domanda che ci poniamo ora è quale sarà la configurazione di risposta,  $r_i$  (o  $\mathbf{r}$  in notazione vettoriale), al presentarsi della

configurazione di stimolo  $s_i$  (o  $\mathbf{s}$  in notazione vettoriale). Prendiamo il primo esempio visto sopra e ci chiediamo: cosa succede se vedo un serpente? La risposta che ha dato la biologia è che la reazione agli stimoli è in buona parte appresa. (Sembra che per alcuni stimoli esista una sorta di memoria "read-only" già presente nel cervello, ma di questo non ci occupiamo). Quindi la reazione comandata dal cervello al presentarsi di un serpente dipende da come il mio cervello è stato abituato a reagire ai serpenti.

Per capire meglio, prendiamo il caso di un bambino che viva in una zona ricca di serpenti. Ogni volta che il bambino è in braccio alla madre, e i due vedono un serpente, la madre urla si agita e scappa.

Il cervello del bambino viene addestrato a provare quella emozione di fronte al serpente e, la prima volta che lo incontra, anche se è da solo, di fronte al serpente prende paura e scappa. Il cervello ha quindi la capacità di *modellarsi* e questa operazione di modellazione può essere chiamata addestramento o, nel linguaggio delle scienze umane, *educazione*.

Nel seguito, ci riferiremo sempre a questo processo con il termine addestramento o in inglese *training*.

Quindi un primo modo per addestrare un cervello, o meglio una rete neurale del cervello, è quello di sottoporlo ad un ciclo stimolo/risposta-forzata. Nel caso visto prima, la risposta-forzata è data dalla reazione della mamma che il bimbo ripete per imitazione.

Indichiamo con  $f_i$  la configurazione che assumono i neuroni del gruppo di risposta quando vengono forzati o, in altri termini, indichiamo con  $f_i$  lo stimolo di condizionamento, per intenderci la reazione della mamma di fronte ai serpenti. A questo punto, possiamo pensare che al presentarsi simultaneo della configurazione  $\mathbf{s}$  di input (il serpente) e della configurazione forzata  $\mathbf{f}$  (lo spavento della mamma), le connessioni sinaptiche si

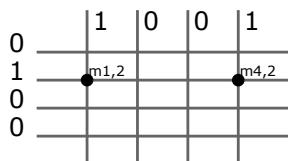
modifichino.

Una delle prime idee nate sul come le connessioni si modificassero è stata quella che ora è nota come *regola di Hebb*. L'idea che sta dietro tale regola è che la connessione tra l'i-esimo neurone di input (segnale) e il j-esimo di output (condizionamento) venga rinforzata se, e solo se, sono attivi simultaneamente sia il neurone  $i$  che il neurone  $j$ .

Partiamo dalla condizione in cui tutti i collegamenti sinaptici sono nulli: quindi i neuroni di input **non attivano** quelli di output. L'idea di Hebb è che presentando simultaneamente un input e un output, si consolidino, quindi si attivino e rimangano plastiche, le connessioni  $m_{i,j}$  relative ai neuroni  $i$  e  $j$  che sono attivi nelle configurazioni presentate in input e output.

Per esemplificare, supponiamo che la configurazione *serpente* equivalga al vettore  $s^{(1)} = (1, 0, 0, 1)$  e che la configurazione *pericolo* equivalga al vettore  $f^{(1)} = (0, 1, 0, 0)$ , dove i valori nulli (0) indicano che non c'è connessione sinaptica (i neuroni sono disconnessi) mentre un valore diverso da 0 indica che c'è connessione. Per semplificare, in questo esempio supponiamo che la connessione sia presente o assente (1 o 0) poi introdurremo l'idea che più è intenso il valore, maggiore è il grado di connessione.

In termini grafici abbiamo il seguente diagramma:



*Fig. 3.2 - Le linee verticali rappresentano gli stimoli attivi ai neuroni di input, le orizzontali l'attività di condizionamento sui neuroni di output. I cerchietti neri indicano i collegamenti sinaptici attivati dalla regola di Hebb.*

Dalla figura 3.2, vediamo che presentando queste configurazioni di input e output, nella rete neurale, si attivano (cioè maggiori di 0) solo le sinapsi indicate con  $m_{1,2}$  e  $m_{4,2}$ , mentre le altre sono non attive, cioè a 0: questa è l'essenza della regola di Hebb.

Il compito che ci diamo nel prossimo paragrafo è quello di creare un modello matematico che riproduca questo comportamento biologico, in questo modo poi potremo realizzarne un algoritmo da scrivere nel linguaggio C.

E' importante notare che quello che abbiamo detto fin'ora nasce da osservazioni empiriche dei sistemi neurali biologici e che, in questo contesto, alcune idee sono state un po' semplificate al fine di renderle più intuibili, senza comunque sacrificarne l'essenza.

## Modellazione matematica

Il modello matematico della regola di Hebb si ottiene in modo molto diretto: indichiamo con  $\mathbf{M}$  una matrice ad elementi  $m_{i,j}$ . La matrice  $\mathbf{M}$  è dipendente dal tempo: ci serve fare questo perché dobbiamo rappresentare la sua evoluzione durante l'addestramento (cioè quando gli mostreremo la reazione della mamma di fronte al serpente).

Diciamo allora che  $\mathbf{M}(t)$  è la matrice all'istante  $t$  e  $\mathbf{M}(t + \Delta t)$  è la stessa matrice all'istante successivo. Limitiamo la nostra analisi ad un contesto in cui consideriamo solo un'evoluzione del tempo a "scatti" di durata  $\Delta t$  o più propriamente ad intervalli discreti, non continui. Facciamo questo per evitare di usare derivate ed integrali che sono idee matematiche, non difficili, ma che non ho introdotto in questo testo.

Abbiamo detto che la regola di Hebb afferma che si ha variazione nell'intensità della connessione sinaptica tra il neurone di stimolo  $i$ -esimo e il neurone di condizionamento  $j$ -esimo solo se essi sono simultaneamente attivi. Matematicamente, possiamo dire che il neurone è attivo se il valore della sua attività è maggiore di zero,

quindi un buon modo per esprimere la regola di Hebb è di esprimere la variazione dell'intensità sinaptica come una quantità proporzionale al prodotto delle attività dei due neuroni.

Dal momento che stiamo parlando di una variazione, esprimiamo questa attività come una delta ( $\Delta$ ), che nel linguaggio matematico si usa appunto per indicare le variazioni. Detto questo avremo che la variazione dell'elemento i-j-esimo della matrice  $\mathbf{M}$  sarà dato da:

$$\Delta m_{i,j} = \lambda f_i s_j$$

La costante  $\lambda$  è un parametro usato per indicare la *plasticità* delle connessioni sinaptiche o la velocità di apprendimento, presto ne intuiremo il *peso* matematico. Usando la (3.1) possiamo calcolare come evolve la matrice  $\mathbf{M}$ , cioè l'*alter ego* matematico della rete neurale.

Scriviamo allora:

$$m_{i,j}(t + \Delta t) = m_{i,j}(t) + \Delta m_{i,j} = m_{i,j}(t) + \lambda f_i s_j \quad (3.5)$$

o in termini vettoriali:

$$\mathbf{M}(t + \Delta t) = \mathbf{M}(t) + \Delta \mathbf{M} = \mathbf{M}(t) + \lambda \mathbf{f} \mathbf{s}^T$$

L'equazione (3.5) esprime un algoritmo che descrive l'evoluzione temporale della matrice di valori di una rete neurale sottoposta ad apprendimento forzato o training. Detta procedura è ottenuta dalla regola di Hebb la quale ha basi puramente biologiche, quindi per il momento non abbiamo nessuna garanzia che il processo di addestramento o apprendimento, descritto in tale equazione, porti ad una matrice  $\mathbf{M}$  che giochi lo stesso ruolo della matrice  $\mathbf{A}$  in (3.3)

Vediamo cosa succede se ipotizziamo che le connessioni sinaptiche abbiano valore pari a 0 prima che l'addestramento abbia inizio. Possiamo scrivere:

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Calcoliamo poi il reciproco del parametro  $\lambda$ :

$$k = \lfloor \frac{1}{\lambda} \rfloor$$

Dove le parentesi  $\lfloor$  e  $\rfloor$  servono a indicare la funzione "parte intera" che restituisce solo la parte intera di un numero (i.e.  $\lfloor 3.14 \rfloor = 3$ ). Se ora pensiamo di sottoporre per un numero  $k$  di volte la nostra rete  $\mathbf{M}$  al condizionamento rappresentato dal vettore  $\mathbf{f}$  si avrà:

$$\mathbf{M}(t + k\Delta t) = 0 + \lambda \mathbf{f} \mathbf{s}^T + \dots + \lambda \mathbf{f} \mathbf{s}^T = k\lambda \mathbf{f} \mathbf{s}^T = \mathbf{f} \mathbf{s}^T \quad (3.6)$$

Come si vede dall'ultimo passaggio abbiamo ottenuto lo stesso risultato che avevamo "imposto" algebricamente in (3.3), quindi possiamo iniziare a stare tranquilli che se istruiamo una rete in modo "corretto" la rete si comporterà come le abbiamo insegnato. Bene, ci sono però due punti che dobbiamo chiarire: 1) Qual è il modo corretto di istruire la rete? 2) Cosa le abbiamo insegnato veramente? Prima di rispondere aggiungiamo un elemento a quanto fatto finora. Come nell'esempio del paragrafo precedente, anche a questa rete possiamo insegnare ad associare una reazione a più di uno stimolo, per essere precisi, potremmo educare la rete a tutti e quattro gli esempi visti sopra. In pratica aggiungiamo una etichetta, o indice, ai vettori di stimolo e condizionamento:  $\mathbf{s}^{(i)}$  e  $\mathbf{f}^{(i)}$  e riscriviamo l'algoritmo di condizionamento come segue:

$$\mathbf{M}(t + k\Delta t) = \sum_{i=1}^4 \mathbf{f}^{(i)} \mathbf{s}^{T(i)}$$

Ora che l'esempio è completo discutiamo le risposte.

Vediamo che esiste un legame tra il parametro  $\lambda$  e il numero  $k$  di iterazioni con cui istruiamo la rete. Nel caso ideale visto ora, cioè quando lo stimolo di condizionamento è sempre lo stesso, la relazione è semplice, infatti perché il training porti al risultato voluto è necessario che sulla destra dell'equazione (3.6) si abbia esattamente l'espressione  $\mathbf{f}\mathbf{s}^T$  e quindi il prodotto  $k\lambda$  deve essere uguale ad 1. Questo impone o un vincolo su  $\lambda$  o, fissato  $\lambda$ , un vincolo sul numero di iterazioni necessarie all'addestramento.

In effetti, nel caso in questione, sembra un dettaglio inutile, basterebbe porre  $\lambda = k = 1$ , quindi perché complicarsi la vita? La risposta arriva se proviamo prima a rispondere anche alla seconda domanda. Cosa abbiamo insegnato davvero alla rete? Le abbiamo insegnato che ad un dato stimolo (serpente) si risponde in un modo preciso (paura). Ma mettiamoci nel caso della mamma che deve educare il figlio al timore delle serpi. Compra un cobra giocattolo di gomma, lo mette in giardino e ogni mattina esce con il bambino in braccio, cammina fino al giocattolo e quando lo vede urla e corre in casa. Perfetto. Quando, a 5 anni ietà, il bambino uscendo di casa da solo incontrerà un vero cobra che differirà dal giocattolo per qualche dettaglio, lunghezza, colore, posizione ecc., che reazione avrà?

Riconoscerà che si tratta comunque di un serpente? Se il vettore  $\mathbf{s}$  di componenti  $(1, 0, 0, 1)$  rappresenta il serpente giocattolo, allora possiamo pensare che il serpente vero sia rappresentato da un vettore  $\mathbf{p}$  del tipo  $(1 + \delta\alpha, 0 + \delta\beta, 0 + \delta\gamma, 1 + \delta\epsilon)$  dove  $\delta\alpha, \delta\beta, \delta\gamma$ , e  $\delta\epsilon$  sono delle quantità piccole che rappresentano la variazione del serpente reale rispetto a quello giocattolo.

Se consideriamo che lo stato della rete neurale sia stato condizionato a riconoscere per ora solo il serpente, la matrice  $\mathbf{M}$  avrà la seguente configurazione:

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

La risposta della rete neurale allo stimolo dato dalla vista del serpente giocattolo ( $\mathbf{s}$ ) è calcolata come:

$$\mathbf{r}^{(1)} = \mathbf{Ms}^{(1)} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{s}^{(1)} = \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \end{bmatrix} = 2 \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 2\mathbf{f}^{(1)}$$

da dove si vede che  $\eta = 2$ .

Se indichiamo con  $\mathbf{p}$  il vettore che rappresenta un serpente reale, avremo che la risposta della rete neurale allo stimolo  $\mathbf{p}$  è data da  $\mathbf{r} = \mathbf{Mp}$  quindi si ha:

$$\begin{aligned} \mathbf{r} = \mathbf{Mp} &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{p} = \begin{bmatrix} 0 \\ 2 + \delta\alpha + \delta\epsilon \\ 0 \\ 0 \end{bmatrix} = \\ &= (2 + \delta\alpha + \delta\epsilon) \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \\ &= (2 + \delta\alpha + \delta\epsilon)\mathbf{f}^{(1)} \end{aligned}$$

dove si è tenuto conto che  $\eta = 2$ .

Come si vede la risposta della rete neurale allo stimolo "reale" o alterata rispetto allo stimolo usato per il suo condizionamento:

1. Serpente giocattolo  $\rightarrow 2(0, 1, 0, 0)$
2. Serpente reale  $\rightarrow (2 + \delta\alpha + \delta\epsilon)(0, 1, 0, 0)$

I due vettori risposta sono multipli l'uno dell'altro, quindi tutto sommato possiamo dire che sebbene una variazione dello stimolo

abbia influenzato l'intensità della risposta (da 2 passiamo a  $2 + \delta\alpha + \delta\epsilon$ ) non ha però modificato la direzione del vettore risposta (i due vettori sono paralleli), quindi possiamo pensare che la rete neurale abbia *elasticità*. Finora però l'abbiamo addestrata a senso unico, cioè la rete è stata addestrata a rispondere solo con il vettore *paura*. Cosa succede se proviamo a educare il bambino dell'esempio ad associare la vista del pettine al concetto di capelli?

Bene, supponiamo che il vettore stimolo del pettine sia  $\mathbf{s}^{(2)} = (0, 1, 0, 0)$  e quello dei capelli  $\mathbf{f}^{(2)} = (1, 0, 0, 0)$ , allora la rete neurale dopo l'addestramento sarà data dalla matrice:

$$\mathbf{M} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

e risponderà correttamente sia allo stimolo  $\mathbf{f}^1$  che allo stimolo  $\mathbf{f}^2$  (provare!). Cosa succede però se proviamo ora a stimolarlo con la vista di un serpente vero? Facciamo i conti:

$$\mathbf{r} = \mathbf{Mp} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{p} = \begin{bmatrix} \delta\beta \\ 2 + \delta\alpha + \delta\epsilon \\ 0 \\ 0 \end{bmatrix}$$

Il vettore di risposta  $\mathbf{r}$  non è più parallelo al vettore  $\mathbf{f}^{(1)}$ , cioè non può essere scritto come  $\mathbf{r} = k\mathbf{f}^{(1)}$  (dove  $k$  è una fattore moltiplicativo), quindi le cose non vanno più tanto bene: il nostro sistema di addestramento della rete ha prodotto una rete troppo rigida e il serpente non viene riconosciuto! Il sistema così come è implementato ora ha due grandi limitazioni:

1. Può separare solo le reazioni di stimoli rappresentabili attraverso un insieme di vettori linearmente indipendenti.
2. La reazione del sistema è corretta solo se lo stimolo

presentato in ingresso (input) è esattamente lo stesso di quello usato nella fase di apprendimento.

L'obiettivo dei prossimi paragrafi sarà quello di superare queste limitazioni.

### 3.4 Programma in C della regola di Hebb

In fondo a questo paragrafo si trova il codice completo che può essere usato per simulare una semplice rete neurale artificiale che implementa la regola di Hebb. La sua utilità in situazioni reali è piuttosto limitata, ma padroneggiare i concetti e il codice di questo primo passo è molto importante.

L'implementazione in C della regola di Hebb non comporta nulla di nuovo rispetto a quanto già visto in precedenza, se non che diversamente da prima, questa volta useremo anche due file per la gestione della nostra ANN. Useremo due file di testo, in questo modo sarà più semplice e diretto leggerli e modificarli.

I file dovranno essere chiamati con il nome esatto con cui vengono aperti nel programma in C e dovranno risiedere nella stessa cartella. Va da sé che l'utente più esperto potrà modificare il programma a piacere e posizionare i file dove vuole.

Il primo file deve essere nominato `data.txt`. Essendo un file di testo è composto da righe. Al suo interno scriveremo solo dei valori di tipo `double`, poi useremo delle virgolette (comma) e dei punti e virgola (semi column) per separare i valori. Nella prima riga scriveremo solo il valore del parametro  $\lambda$  visto prima. Nelle righe successive, scriveremo otto valori `double` divisi in due quartetti. Nel primo scriveremo i quattro valori delle componenti del vettore di input e nel secondo le componenti del vettore di condizionamento (o output). Le componenti di ogni quartetto devono essere separate da una virgola, mentre i due quartetti devono essere separati da un punto e virgola. Per esempio, per codificare nel file `data.txt` il valore  $\lambda = 0.91$  e i due vettori  $s^{(1)} = (0.9, 1, 0, 0.31)$  e  $f^{(1)} = (0, 1.1, 0, 0)$ , scriveremo il

seguente file:

0.91

0.9,1,0,0.31;0,1.1,0,0

## Gestione di un file

Un file è un insieme ordinato di dati memorizzati su un archivio permanente. I dati sono codificati in bit che sono memorizzati in byte, quindi un file è un insieme ordinato di byte (nota: se scrivo ordinato è perché è importante!). Pensiamo al file data.txt: lo abbiamo creato e salvato su un disco o su una chiavetta USB. Il sistema operativo non sa nulla di data.txt. Noi scriviamo un programma che deve usare i dati scritti dentro data.txt, come facciamo a recuperarli e leggerli? Come ho appena chiarito, il nostro sistema operativo non sa nulla del file, ma sul disco ha una sua tabella (accessibile solo al sistema operativo) in cui tiene una relazione tra i file e le loro proprietà. Ogni file è descritto da un numero detto *indice*. Il nome e la cartella, così come la dimensione e i permessi, sono proprietà (attributi) del file. L'informazione più importante, comunque, è l'indirizzo hardware in cui iniziano i dati del file.

Se vogliamo usare i dati memorizzati in data.txt dobbiamo prima chiedere al sistema operativo di rendere quei dati accessibili al programma che li vuole usare. Per fare questo in C si usa la funzione fopen() che fa parte della libreria <stdio.h>. La funzione fopen(), a sua volta, chiama la funzione open() che finalmente esegue una chiamata di sistema, cioè invoca l'esecuzione di una procedura che verrà eseguita direttamente dal sistema operativo, il quale ha i permessi per accedere e modificare il *File System*, cioè i dati che descrivono i file. Una nota importante, se l'utente sotto il quale gira il programma non ha i permessi di accesso al detto file, anche se l'accesso al File System

avviene dal sistema operativo, l'accesso al file sarà comunque negato.

La chiamata alla funzione fopen restituisce un puntatore alla struttura FILE, sempre descritta in <stdio.h>. Tale puntatore permette le operazione di lettura e scrittura sul file. Nel codice lo vediamo nei passaggi:

```
FILE * f=fopen("data.txt","rt");
```

```
...
```

```
double lambda;
```

```
fscanf(f,"%lf",&lambda);
```

Come si vede dalla prima riga qui sopra, per aprire un file dobbiamo specificare il nome e la modalità di accesso. In questo caso, indichiamo al sistema che vogliamo leggere un file di testo, quindi il secondo parametro che passiamo alla chiamata è la string "rt" che significa "read-text", cioè apri il file come un file di testo in modalità lettura. Nella quarta riga viene chiamata la funzione fscanf() passandole il puntatore al file che abbiamo aperto, la stringa che specifica il formato di dati per ogni riga di testo e infine un l'indirizzo in cui verrà memorizzata la lettura eseguita sul file. Notiamo che nelle righe successive del codice che viene presentato per intero al termine del paragrafo, la chiamata alla funzione fscanf() avviene passando lo stesso puntatore al file data.txt, ma cambiando opportunamente la stringa di formato e la lista degli indirizzi in cui memorizzare le letture.

Vediamo il codice:

```
fscanf(f,"%lf,%lf,%lf,%lf;%lf,%lf,%lf,%lf",
inp,inp+1,inp+2,inp+3,out,out+1,out+2,out+3);
```

Come si vede, la stringa di formato specifica che ogni riga è composta da valori di tipo double separati da una virgola e, in mezzo, da un punto e virgola. Essi vengono memorizzati nei due array **inp** e **out** usati per rappresentare i vettori **s** e **f** visti nel paragrafo precedente.

## Le macro

Le *macro define* in C sono delle stringhe a cui si può dare un nome. Il nome può essere poi usato nel codice al posto della stringa e il preprocessore provvederà a sostituire le occorrenze della macro con la stringa corretta. Nel codice ho usato le `#define` per indicare le dimensioni degli array. Nell'ANSI C, cioè nel C che rispetta lo standard ANSI, non è previsto che un array possa essere dimensionato a runtime mediante l'uso di una variabile. Per fare questo si deve usare la funzione `malloc`. Se non è previsto che le dimensioni dell'array possano cambiare a runtime, invece che allocare lo spazio di memoria a loro riservato mediante la `malloc` è possibile definire le loro dimensioni utilizzando le macro. Nel codice in questione ho usato le definizioni seguenti:

```
#define VROWS 4  
#define MROWS 4  
#define COLS 4
```

In questo modo se si deciderà di usare il codice per rappresentare vettori e matrici di ordine superiore a  $4 \times 4$ , per esempio  $8 \times 8$  sarà sufficiente sostituire il numero 4 con l'8 solo qui, anziché in tutto il codice, correndo anche il rischio di effettuare una sostituzione anche dove non andrebbe fatta.

Una cosa: chi volesse modificare il codice in tal modo tenga presente che dovrà modificare opportunamente anche la funzione `read_input_output` e la `read_test`, oltre ovviamente ai due file di dati. Come? Beh, se siete bravi da voler fare di testa vostra... fatelo fino in fondo e provate ad arrangiарvi!

## Algoritmo di Hebb

In questo paragrafo dobbiamo implementare l'algoritmo visto in equazione (3.6).

I passi sono i seguenti:

1. Apri il file `data.txt`
2. Leggi il parametro `lambda`.
3. Inizializza la matrice `M` (array a due indici) a 0.
4. Ripeti per ogni riga di dati:
  1. Leggi 4 double e caricali nell'array `in`.
  2. Leggi 4 double e caricali nell'array `out`.
  3. Esegui il prodotto vettore  $x$  trasposto e caricalo in `DM`
  4. Somma `lambda`  $\times$  `DM` a `M`
5. Chiudi il file `data.txt`
6. //Qui la rete è stata addestrata
7. Apri il file `test.txt`
8. Ripeti per ogni riga di dati:
  1. Leggi 4 double e caricali nell'array `in`.
  2. Esegui il prodotto di `M` per `in` e caricalo in `out`.
  3. Stampa il risultato a video.
9. Chiudi `test.txt`

## Il codice C

### Listato 3.3 `regola_hebb.c`

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define VROWS 4
#define MROWS 4
#define COLS 4

void matrix_dot_vector
(double res[MROWS],double matrix[][VROWS],double vector[VROWS]);
void matrix_init
(double res[][VROWS],double val);
void matrix_plus_matrix
```

```

(double res[][VROWS],double m1[][VROWS],double m2[][VROWS]);
void matrix_to_vector
(double w[][VROWS],double v[]);
void print_matrix
(double x[],int r, int c,int R,int C);
int read_input_output
(FILE *fp,double inp[],double out[]);
int read_test
(FILE *fp,double test[]);
void vector_dot_Tvector
(double res[][VROWS],double a[VROWS],double Tb[MROWS]);

int main()
{
    //Carica i dati di addestramento
    FILE * f=fopen("data.txt","rt");
    if(f==0) exit(1);

    double lambda;
    double in[VROWS];
    double out[VROWS];
    double M[VROWS][COLS];
    double DM[VROWS][COLS];
    double v[VROWS*VROWS];

    //Legge il valore di lambda
    fscanf(f,"%lf",&lambda);

    //Inizializza a 0 la la matrice M
    matrix_init(M,0);

    //Addestra la rete
    while(read_input_output(f,in,out)!=EOF)
    {
        vector_dot_Tvector(DM,out,in);
        matrix_plus_matrix(M,M,DM);
    }

    fclose(f);
    matrix_to_vector(M,v);
    //Mostra M
    print_matrix(v,4, 4,10,50);

    //Apre il file con i dati di test
    int tn=0;
    f=fopen("test.txt","rt");
    if(f==0) exit(1);
    //Testa la rete
    while(read_test(f,in)!=EOF)

```

```

{
    matrix_dot_vector(out,M,in);
    print_matrix(in,4, 1,10+tn*10,100);
    print_matrix(out,4, 1,10+tn*10,105);
    tn++;
}
return 0;
}

int read_input_output
(FILE *fp,double inp[],double out[]){
    int r;
    r=fscanf(fp,"%lf,%lf,%lf,%lf;%lf,%lf,%lf,%lf",
              inp,inp+1,inp+2,inp+3,out,out+1,out+2,out+3);
    return r;
}

int read_test
(FILE *fp,double test[]){
    int r;
    r=fscanf(fp,"%lf,%lf,%lf,%lf",
              test,test+1,test+2,test+3);
    return r;
}

void matrix_init(double res[][VROWS],double val)
{
    int r,c;
    r=c=VROWS;
    for(int i=0;i<r;i++)
        for(int j=0;j<c;j++)
    {
        res[i][j]=val;
    }
}

void matrix_to_vector
(double w[][VROWS],double v[])
{
    int r,c;
    int k;
    r=c=VROWS;
    for(int i=0;i<r;i++)
        for(int j=0;j<c;j++)
    {
        k=i*VROWS+j;
        v[k]=w[i][j];
    }
}

```

}

```

void print_matrix
(double x[],int r, int c,int R,int C)
{
    for(int i=0;i<r;i++)
        for(int j=0;j<c;j++)
        {
            double gl;
            gl=x[i*c+j];
            if(gl>=0)
                printf("\x1b[%d;%dH%0.1lf \x1b[0m ",i*2+R,6*j+C,gl);
            else
                printf("\x1b[%d;%dH%0.1lf \x1b[0m ",i*2+R,6*j+C-1,gl);
        }
    fflush(stdout);
}

void vector_dot_Tvector
(double res[][VROWS],double a[VROWS],double Tb[MROWS])
{
    for(int i=0;i<MROWS;i++){
        for(int j=0;j<VROWS;j++){
            res[i][j]=a[i]*Tb[j];
        }
    }
}

void matrix_dot_vector
(double res[MROWS],double matrix[][][VROWS],double vector[VROWS])
{
    for(int i=0;i<MROWS;i++){
        res[i]=0;
        for(int j=0;j<VROWS;j++){
            res[i]+=matrix[i][j]*vector[j];
        }
    }
}

void matrix_plus_matrix
(double res[][VROWS],double m1[][VROWS],double m2[][VROWS])
{
    for(int i=0;i<MROWS;i++){
        for(int j=0;j<VROWS;j++){
            res[i][j]=m1[i][j]+m2[i][j];
        }
    }
}

```

## **Compilazione** regola\_hebb.c

```
~/Documents/nn> gcc -o regola_hebb regola_hebb.c  
~/Documents/nn> chmod u+x regola_hebb  
~/Documents/nn>./regola_hebb
```

## **Addestrare la rete e provarla**

Vediamo in questo paragrafo come provare il codice mostrato nel paragrafo qui sotto. Seguiamo i punti seguenti:

1. Editiamo il file sorgente in un editor di testo e salviamolo nella cartella nn con il nome regola\_hebb.c

2. Compiliamo il sorgente.

3. Editiamo, sempre con l'editor di testo, un file che chiameremo data.txt con il contenuto seguente:

```
1  
1,0,0,1;0,1,0,0  
0,1,0,0;1,0,0,0  
0,0,1,0;0,0,1,0  
1,0,0,-1;0,0,0,1
```

4. Editiamo, sempre con l'editor di testo, un file che chiameremo test.txt con il contenuto seguente:

```
1,0,0,1  
0,1,0,0  
0,0,1,0  
1,0,0,-1
```

5. Dal prompt dei comandi lanciamo l'eseguibile regola\_hebb che abbiamo compilato prima:

```
~/Documents/nn> ./regola_hebb
```

l'output che ci aspettiamo è il seguente:

0.0	1.0	0.0	0.0		1.0	0.0
1.0	0.0	0.0	1.0		0.0	2.0
0.0	0.0	1.0	0.0		0.0	0.0
1.0	0.0	0.0	-1.0		1.0	0.0
				0.0	1.0	
				1.0	0.0	
				0.0	0.0	
				0.0	0.0	
				0.0	0.0	
				0.0	0.0	
				1.0	1.0	
				0.0	0.0	
				0.0	0.0	
				0.0	0.0	
				-1.0	2.0	

Fig 3.3 - Screenshot dell'output del programma hebb.c

**Attenzione:** come si capisce studiando il codice sorgente, se il programma non trova nella cartella corrente (la stessa cartella da cui lanciate il programma) i file specificati nelle due fopen, interrompe l'esecuzione senza nessun messaggio. In un programma completo questi controlli andrebbero sempre fatti, ma qui per motivi ovvi di didattica cerco sempre di tenere il codice il più ridotto possibile. Per cui, se non vedete l'output corretto, verificate di aver dato i giusti nomi ai file e anche che si trovino nella posizione corretta.

Dopo aver provato la rete con questo addestramento standard fate qualche prova alternativa per capire come funziona. Per esempio, dopo aver portato il parametro lambda ad un quarto di sé stesso (0.25), preparate per ogni stimolo  $s^{(i)}$  non uno, ma quattro vettori che differiscano di poco l'uno dall'altro, poi eseguite i test. Provate e riprovate, solo così vi farete un'idea del sistema che state trattando.

### 3.5 Struttura del percettrone e implementazione in C

Il modello del percettrone è stato pubblicato nel 1958 sulla rivista Psychological Review con il titolo *The perceptron: a probabilistic model for information storage and organization in the brain*, unico autore Rosenblatt. Questi erano gli anni d'oro della ricerca sui

meccanismi con i quali il cervello percepisce, memorizza ed elabora l'informazione. L'autore suggeriva che per capire a fondo tali meccanismi si dovesse cercare di dare una risposta a tre domande che qui riassumo:

1. Come vengono percepite dal sistema biologico (occhi/cervello /sensi ecc..) le informazioni che arrivano dal mondo fisico?
2. In che forma queste informazioni sono archiviate o memorizzate?
3. Attraverso quali meccanismi queste informazioni influenzano la capacità di riconoscere gli oggetti (o le idee) del mondo esterno e in che modo influenzano il comportamento?

Ora che ho elencato queste tre domande provate a chiudere il libro e riscrivetele su un foglio di carta, perché solo così se ne apprezzeranno la profondità e il valore dei modelli che tentano di fornire delle risposte.

Capire il problema prima di studiare la soluzione è importante e ci porta sulla giusta lunghezza d'onda. Mostrare qui modelli matematici e programmi in C può anche fare impressione, ma se non li si collega ad idee che siano la risposta a domande sensate, finiscono per lasciare il tempo che trovano. Possiamo anche usare un linguaggio di moda su una piattaforma di moda, ma se non sappiamo perché lo stiamo facendo finiremo per spercare del tempo.

Tornando alla prima domanda, Rosenblatt scrive che ad essa risponde già la fisiologia e che gli strumenti della ricerca sono già ben affinati, per questo si concentra sulle successive due.

Per quanto riguarda la memorizzazione delle informazioni l'idea di Rosenblatt è che esse siano archiviate nell'intensità dei pesi collegamenti fra i neuroni. Per dimostrare questa idea, sviluppò il concetto di percettrone, un sistema capace di classificare una immagine di input senza che sia stata pre-caricata in memoria

una immagine di (esempio) della classe a cui essa appartiene.

Pensiamo ad un sistema che abbia in memoria l'immagine di una mela e a cui vengano presentate come input le immagini di diversi frutti. Il sistema per classificare il frutto come mela/non-mela, confronta l'input con l'immagine della mela che ha in memoria, e stabilisce se essa appartiene o no alla classe mela. Questo sistema non è il modo con cui lavora il percettrone che, al contrario, non memorizza l'immagine di esempio di una mela, ma memorizza la configurazione dei pesi dei collegamenti dei singoli neuroni di input quando si ha che in input viene presentata la mela e in output essa è classificata correttamente.

Vediamo nel dettaglio come funziona.

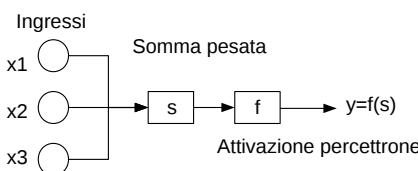


Fig. 3.4 - Schema a blocchi del percettrone di Rosenblatt.

Indichiamo con  $\mathbf{x}$  il vettore le cui componenti rappresentano gli  $n$  neuroni di input. In output abbiamo un solo neurone che quindi indichiamo con il valore scalare  $y$ . Indichiamo poi con  $f$  la funzione di soglia che attiva o meno il neurone di output in base all'eccitazione ricevuta dall'input. L'idea è che la funzione  $f$  dipenda dalla somma pesata  $s$  dei neuroni di input in modo che se  $s$  supera un certo valore di soglia si abbia  $f(s) = 1$  altrimenti  $f(s) = 0$ . L'idea di Rosenblatt è che ogni neurone contribuisca in misura proporzionale all'intensità con cui esso è connesso all'output.

In figura 3.4 è riportata una descrizione schematica di un percettrone a tre ingressi, in cui si evidenzia il blocco funzionale dove si sommano gli stimoli e il blocco di attivazione in cui la

funzione  $f$  stabilisce se attivare o meno l'uscita del percettrone.

Indicando con  $\mathbf{w}$  il vettore le cui componenti rappresentano i pesi di connessione di ogni neurone di input a quello di output si può scrivere:

$$s = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^n w_i x_i + b \quad (3.7)$$

dove  $b$  è detto bias. Il segnale di input è classificato dalla funzione  $f(s)$  che si definisce in modo che possa assumere solo due valori (uno per classe). Per esempio si può definire  $f$  come segue

$$f(s) = \begin{cases} 1 & \text{se } s > 0 \\ 0 & \text{se } s \leq 0 \end{cases} \quad (3.8)$$

Il percettrone può essere addestrato a riconoscere la classe di appartenenza di un segnale di input con una procedura algoritmica indipendente dalla logica con cui i segnali di input sono classificati nella realtà oggettiva.

L'algoritmo di addestramento utilizza un insieme di dati di apprendimento, cioè dati campione di cui sia nota la classe di appartenenza.

Si inizializza il percettrone assegnando un valore arbitrario alle componenti del vettore  $\mathbf{w}$  e al bias  $b$ .

Usando tale valore si computa la  $f$ , e, usando il valore noto della classe di appartenenza del dato, quindi il valore che la  $f$  dovrebbe avere, si modifica il vettore  $\mathbf{w}$  in modo che sia soddisfatta la (3.8).

Il procedimento continua fino alla completa convergenza, cioè una situazione in cui qualsiasi nuova iterazione di apprendimento non comporta più modifiche al vettore  $\mathbf{w}$ . L'algoritmo di apprendimento si definisce su un insieme  $X$  di coppie dato/classe (detto appunto training set) date da  $(\mathbf{x}, d)$  dove  $\mathbf{x}$  è un vettore che rappresenta un segnale di input e  $d$  è la sua classe che deve essere nota.

Per semplificare molto la notazione e anche la successiva computazione in linguaggio C, si può integrare il bias  $b$  nel vettore  $\mathbf{w}$  inserendolo come componente di indice 0:  $w_0 = b$  e allo stesso tempo aggiungendo la componente di indice 0 anche al vettore  $\mathbf{x}$  con valore costante 1, cioè  $x_0 = 1$ . In questo modo l'equazione (3.7) può essere scritta nella forma compatta data dall'equazione seguente:

$$s = \mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^n w_i x_i = b \times 1 + \sum_{i=1}^n w_i x_i \quad (3.9)$$

L'algoritmo può essere descritto come segue:

1. Inizializzazione del vettore  $\mathbf{w}$ .
2. Per ogni elemento dell'insieme  $X$  si eseguono i passi seguenti:
  1. calcolo della somma  $s$  e dell'output  $y = f(s)$
  2. aggiornamento del vettore  $\mathbf{w}$  secondo la seguente:  

$$\mathbf{w}(t+1) = \mathbf{w}(t) + (d - y)\mathbf{x}$$

Ci si potrebbe chiedere quante iterazioni sono necessarie per addestrare correttamente il percettrone, ma non è possibile rispondere a priori a questa domanda, quello che invece è dato sapere è che se i dati sono linearmente separabili allora è possibile addestrare il percettrone a riconoscerne la classe senza errori. Questo risultato è noto come *teorema di convergenza del percettrone* di cui non vedremo i dettagli matematici che sono peraltro ben disponibili in rete.

Prima di presentare il codice C per la realizzazione di un semplice programma che usa il percettrone, vediamo un esempio di cui possiamo seguire la computazione eseguendo i calcoli manualmente. Consideriamo a questo scopo un problema semplice e trattabile. Si supponga che le fisionomie fisiche delle persone si possano classificare in longilinee e normotipe. Supponiamo inoltre che l'intelligenza umana sia in grado di

classificare la tipologia di appartenenza semplicemente osservando un fisico, ma essendo all'oscuro del possibile algoritmo usato dal proprio cervello.

Come esseri umani, possiamo (nel senso che abbiamo le capacità) osservare un nostro simile e classificarlo come longilineo o normotipo, ma se ci chiedessero come abbiamo fatto non sapremmo rispondere.

Proviamo a vedere se riusciamo ad addestrare un percettrone a farlo al posto nostro, cioè a classificare le fisionomie semplicemente istruendolo secondo un insieme di dati campione che gli forniamo. A tale scopo prepariamo un insieme di dati in cui segnamo l'indice di altezza e di peso e la classe di appartenenza di un insieme di persone che abbiamo appunto classificato. Detto  $IA$  l'indice di altezza supponiamo (ma è un gioco) che l'altezza sia data da:  $h = 150 + IA \times 3$  mentre il peso sia dato da:  $p = 50 + IP \times 1.5$  dove con  $IP$  abbiamo inteso l'indice del peso.

Fatto ciò usiamo parte dei dati per istruire il percettrone e parte per verificarne la correttezza. In tabella XI inserisco i dati di un campione immaginario, senza alcuna pertinenza con la realtà.

*Tabella XI. Dati di addestramento e test per un perceptron che distingue la fisionomia longilinea dal normotipo. **IA** è l'indice di altezza, **IP** quello di peso. La classe 0 indica il tipo normolinoe mentre la 1 il longilineo.*

#	IA	IP	Classe
Apprendimento			
1	1	7	0
2	3	1	1
3	3	9	0
4	4	3	1
5	2	3	0
6	7	3	1
7	2	5	0
8	7	5	1
Test			
9	2	6	0
10	3	7	0
11	8	4	1
12	9	6	1

Anzi tutto inizializziamo  $\mathbf{w}(t)$  con dei valori scelti casualmente:

$$\mathbf{w}(t) = (0.2, 0.5, -2.5)$$

Prendiamo ora il primo elemento della tabella e calcoliamo  $s$  secondo la (3.9):

$$s = 0.2 + 0.5 + 7 \times (-2.5) = -16.8$$

da cui segue che

$$y = f(-16.8) = 0$$

Calcoliamo quindi il nuovo valore di  $\mathbf{w}(t+1)$  come:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + (d - y)\mathbf{x}$$

quindi

$$\mathbf{w}(t+1) = (0.2, 0.5, -2.5) + (0 - 0) \times (1, 1, 7)$$

dove si ricordi che abbiamo fissato  $x_0 = 1$ .

Si ha quindi che la prima iterazione non comporta variazione dei pesi del percettrone. Vediamo insieme la seconda e lasciamo le successive per esercizio.

Si ha:

$$s = 0.2 + 1.5 + 1 \times (-2.5) = -0.8$$

quindi

$$y = f(-0.8) = 0$$

. Il nuovo vettore dei pesi è dato da:

$$\mathbf{w}(t+2) = (0.2, 0.5, -2.5) + (1 - 0) \times (1, 3, 1) = (1.2, 3.5, -1.5)$$

che essendo diverso da quello iniziale, ci mostra l'evoluzione (o apprendimento) del sistema.

## Il codice C

### Listato 3.4 percettrone.c

```
#include <stdlib.h>
#include <stdio.h>

/*Numero di neuroni di input*/
#define VROWS 2

/*Ampiezza degli assi x e y (da -X_Y_MAX/2 a + X_Y_MAX/2)
Si modifichi la dimensione della finestra del terminale
in conseguenza del valore assegnato a questa macro.
*/
#define X_Y_MAX 40

/* Calcola il valore di attivazione del percettrone in base all'input e ai pesi*/
double calculate_activation
(double weight[],double input[]);

/* determina l'uscita alta o bassa (0 o 1) del percettrone*/
int classify_activation(double activation_value);

/*Stampa a video il grafico dei punti della retta separante le classi*/
void print_graph
(double x,double y, int cls, double w[],char label[]);

void vector_init(double res[VROWS],double val);
int read_input_output(FILE *fp,double inp[],double * out);
int read_test(FILE *fp,double test[]);

int main()
{
    /*Carica i dati di addestramento*/
    FILE * f=fopen("perc_data.txt","rt");
    if(f==0) exit(1);

    double x[VROWS+1];/*input cognitrone*/
    double out; /*output cognitrone*/
    double w[VROWS+1];/*pesi cognitrone*/
    double r; /*velocità apprendimento*/
    double a;
    int cls;
```

```

/*Inizializza il vettore pesi del percettrone*/
vector_init(w,1);
w[0]=0.2;
w[1]=-0.5;
w[2]=-2.5;
vector_init(x,1);
r=0.05;

/*Addestra il percettrone*/
//printf("Addestramento Percettrone\n*****\n");
while(read_input_output(f,x,&out)!=EOF)
{
    a=calculate_activation(w,x);
    cls=classify_activation(a);
    printf("x=(%lf,%lf), out=%lf, cls=%d\n",x[1],x[2],out,cls);
    /*Calcolo il nuovo vettore pesi*/
    for(int i=0; i<VROWS+1; i++)
    {
        w[i]=w[i]+r*(out-cls)*x[i];
        printf("w_%d=%lf\t",i,w[i]);
    }
    print_graph( x[1],x[2], out, w,"Apprendimento");
    printf("\n\n");
}
//printf("Percettrone addestrato\n*****\n");

fclose(f);

f=fopen("perc_test.txt","rt");
if(f==0) exit(1);
/*Testa il percettrone*/
while(read_test(f,x)!=EOF)
{
    a=calculate_activation(w,x);
    cls=classify_activation(a);
    print_graph( x[1],x[2], cls, w,"Test");
    //printf("x=(%lf,%lf), cls=%d\n",x[1],x[2],cls);
}
fclose(f);
}

double calculate_activation(double w[],double x[])
{
    int c;
    double a=0;
    c=VROWS;
    for(int j=0;j<c+1;j++)
    {

```

```

        a=a+w[j]*x[j];
    }
    return a;
}

int classify_activation(double a)
{
    if(a>0) return 1; else return 0;
}

void vector_init(double res[VROWS],double val){
    int c;
    c=VROWS;

    for(int j=0;j<c+1;j++)
    {
        res[j]=val;
    }
}

int read_input_output(FILE *fp,double inp[],double* out){
    int r;
    r=fscanf(fp,"%lf,%lf,%lf",inp+1,inp+2,out);
    return r;
}

int read_test(FILE *fp,double inp[]){
    int r;
    r=fscanf(fp,"%lf,%lf",inp+1,inp+2);
    return r;
}

void print_graph(double x,double y, int cls, double w[],char label[])
{
    /*Mappa logica del grafico*/
    static char plot_r_c[X_Y_MAX+1][X_Y_MAX+1];
    int r,c;
    /*Cancella il riquadro*/
    printf("\x1b[2J");
    /*Stampa la retta e gli assi*/
    double m,q;
    if(w[2]==0) return;
    m=-w[1]/w[2];
    q=-w[0]/w[2];

    for(int i=-X_Y_MAX/2+1; i<X_Y_MAX/2;i+=1)
    {
        r=X_Y_MAX/2;

```

```

c=X_Y_MAX/2+i;
printf("\x1b[%d;%dH%c", r, c, '-');
c=X_Y_MAX/2;
r=X_Y_MAX/2+i;
printf("\x1b[%d;%dH%c", r, c, '|');

c=X_Y_MAX/2+i;
r=X_Y_MAX/2-(m*i)+q);
if(c<=X_Y_MAX&&r<=X_Y_MAX&&r>0){
    if(plot_r_c[r][c]!='o'&&plot_r_c[r][c]!='+')
        plot_r_c[r][c]='/';

    printf("\x1b[%d;%dH%c", r, c, '/');
}
}

/*Aggiunge le frecce degli assi*/
r=X_Y_MAX/2;
c=X_Y_MAX;
printf("\x1b[%d;%dH%s", r, c, "->x");
c=X_Y_MAX/2;
r=1;
printf("\x1b[%d;%dH%s", r, c-1, "y^");

/*Ripristina i punti*/
for(int i=1;i<X_Y_MAX+1;i++)
    for(int j=1;j<X_Y_MAX+1;j++)
        if(plot_r_c[i][j]=='+'||plot_r_c[i][j]=='o')
        {
            printf("\x1b[%d;%dH%c", i, j, plot_r_c[i][j]);
        }

r=-y+X_Y_MAX/2;
c=X_Y_MAX/2+x;
if(cls==0)
//Stampa cls=0
{
    printf("\x1b[%d;%dH%c", r, c, '+');
    plot_r_c[r][c]='+';
}
else
//Stampa cls=0
{
    printf("\x1b[%d;%dH%c", r, c, 'o');
    plot_r_c[r][c]='o';
}

char retta[30];
sprintf(retta,"m=%0.2lf, q=%0.2lf",m,q);
printf("\x1b[3;1H%s",retta);
printf("\x1b[2;1H%s",label);
getchar();

```

}

### **Compilazione** percettrone.c

~/Documents/nn> gcc -o percettrone percettrone.c

~/Documents/nn> chmod u+x percettrone

~/Documents/nn>./percettrone

### **Addestrare e provare il percettrone**

Per addestrare e testare il percettrone seguiamo gli stessi passi visti nel paragrafo precedente.

1. Editiamo il file sorgente in un editor di testo e salviamolo nella cartella nn con il nome percettrone.c

2. Compiliamo il sorgente.

3. Editiamo, sempre con l'editor di testo, un file che chiameremo perc\_data.txt con il contenuto seguente:

1,7,0

3,1,1

3,9,0

4,3,1

2,3,0

7,3,1

2,5,0

7,5,1

4. Editiamo, sempre con l'editor di testo, un file che chiameremo perc\_test.txt con il contenuto seguente:

2,6

3,7

8,4

9,6

5,5

5. Dal prompt dei comandi lanciamo l'eseguibile percettrone e il programma stamperà su un sistema di assi cartesiani il

primo punto le cui coordinate sono lette dal file `perc_data.txt` e una retta. Il punto verrà visualizzato con il carattere '+' o con il carattere 'o'. Premendo poi il tasto Invio o Ret il programma leggerà le coordinate del secondo punto e lo stamperà nello schermo. Noteremo la retta cambiare inclinazione cercando di passare in mezzo ai due punti. La variazione dell'inclinazione della retta è frutto dell'apprendimento del percepitrone che ad ogni ciclo modifica i pesi sinaptici in base al confronto tra la classificazione calcolata (0 o 1) e quella impostata nel file di addestramento (il terzo parametro dopo le coordinate). Sulla sinistra è stampata una etichetta che mostra l'inclinazione  $m$  della retta (detta coefficiente angolare) e lo scostamento  $q$  dall'asse delle  $y$ . L'etichetta specifica anche se si tratta di dati di prova o di test. Durante la fase di prova è possibile (anzi doveroso) che il percepitrone sbagli la classificazione. Il dataset da me preparato prevede una parte di addestramento e una di test.



*Fig 3.5 - Quattro screenshot dell'output del programma `percettore.c` mostrano l'apprendimento del sistema.*

L'addestramento di una rete neurale è un'attività delicata. Giunti alla fase di test dovremmo aspettarci che il percepitrone distingua le due classi senza più errori e, una volta superata quest'ultima fase i risultati del percepitrone saranno considerati corretti. Per capire la delicatezza del processo, provate ad immaginare un percepitrone che viene usato per diagnosticare una patologia in base a due parametri sanguigni. Per fissare le idee consideriamo

ad esempio che si voglia usare un percepitrone per valutare il rapporto tra numero di globuli rossi e i bianchi nell'ambito di una data patologia. Per addestrare il percepitrone prenderemo cento persone malate e cento sane. Su di un file analogo a perc-data.txt, per ognuno dei pazienti inseriremo una riga dove scriveremo il numero dei globuli rossi e bianche misurati dalle analisi del sangue, e la classe del soggetto, cioè malato (1) o sano (0). Ovviamente avremo stabilito la classe del soggetto attraverso un altro criterio diagnostico, per esempio una radiografia. Dopo aver addestrato il percepitrone prendiamo altri cento soggetti tra sani e malati (ma sempre precedentemente diagnosticati) e vediamo se vengono classificati correttamente, ma senza più variare i pesi sinaptici. Se il percepitrone supera il test allora diciamo che è pronto. Da quel momento in poi, il percepitrone verrà usato nella clinica al posto della radiografia. Ci sono due cose da notare, la prima è quella che innalza questi sistemi alla classe di "intelligenti". Vediamo che nessun analista ha cercato di comprendere la relazione tra i parametri sanguigni e la patologia, semplicemente si è istruito un sistema informatico a riconoscere una relazione "nascosta" nei dati, una relazione che spesso rimane concretamente criptata. Questo perché nei casi reali il percepitrone deve distinguere non punti del piano ma punti di spazi che possono avere anche 20 o 30 dimensioni, separandoli non con una retta ma con un iper-piano (non ne abbiamo parlato qui). La seconda è che dal momento in cui il percepitrone sostituirà la radiografia (per evitare le radiazioni ionizzanti) non ci sarà più un dato di confronto per sapere se la diagnosi è corretta, quindi se sbaglia, sbaglia sulla vita di un paziente. Per questo è importante avere le idee molto chiare sulla matematica alla base di questi sistemi e anche sui limiti di affidabilità, di cui tra l'altro parleremo nel prossimo paragrafo. Per ora provate a cambiare i dati a piacere per vedere in quali condizioni riuscite ancora ad addestrare il percepitrone. Provate con qualche esempio reale, per

esempio con altezza e peso dei vostri amici o età e girovita, quello che volete, però attenzione al codice perché è impostato per stampare solo i dati in un certo range, se dovete usare un range diverso modificalo a piacere. In bocca al lupo!

## Come funziona l'addestramento?

Nel paragrafo precedente è stato introdotto un algoritmo per l'apprendimento del percettrone. L'algoritmo presentato si basa sulla regola di Widrow-Hoff, presentata per la prima volta nell'articolo dei due autori (Bernard Widrow e Marcian Hoff) intitolato *Adaptive Switching Circuits*, presenta la regola per l'addestraento di Adaline, uno dei primi sistemi dedicati di intelligenza artificiale.

La regola di Widrow-Hoff, anche nota come *delta rule* è una caso particolare della regola generale di addestramento, in cui cade anche la regola di Hebb vista prima, che può essere espressa come segue:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + r(t)$$

Nel caso della regola di Hebb, abbiamo che  $r(t)$  è data dal prodotto dei vettori stimolo e risposta, come visto in equazione (3.5), nel caso della delta rule, invece, si ha  $(d - y)\mathbf{x}$ .

I due meccanismi di apprendimento, fanno quindi parte di uno stesso quadro generale, ma hanno significati diversi. La regole di Hebb, come già visto, interpreta il principio fisiologico secondo cui sono fortificati i collegamenti sinaptici tra i neuroni afferenti ed efferenti che sono attivi simultaneamente. La delta rule, invece, si basa su un'idea diversa e più matematica che biologica. Anzi tutto si considera una funzione detta di *costo* che l'algoritmo di apprendimento dovrà minimizzare (quindi: minimizzare i costi). Tale funzione nel caso della regola di Widrow-Hoff, è scelta come:

$$E(t) = \frac{1}{2}(d - y)^2$$

Nel caso del percepitrone (che stiamo analizzando) il significato della funzione di costo è chiaro, essa è infatti proporzionale al quadrato dell'errore compiuto dal percepitrone nella classificazione dell'input, per cui il significato di minimizzare tale funzione è in sé chiaro.

Definita la funzione di costo, si calcola la variazione dei pesi  $\mathbf{w}$  che la minimizza. Questa variazione è proprio il gradiente della funzione di costo calcolato rispetto le varie componenti del vettore dei pesi, i calcoli esplicativi conducono esattamente alla formula che abbiamo usato per l'apprendimento.

Sebbene in questo testo introduttivo non seguiamo nel dettaglio i conti esplicativi di questa regola, è importante averne chiaro il principio generale che è stato qui esposto, specie in previsione del prossimo capitolo, dove essa sarà usata per l'addestramento di reti neurali con più strati.

### 3.6 Una rete di perceptroni, gli strati *deep*

#### **Il percepitrone ha un limite importante!**

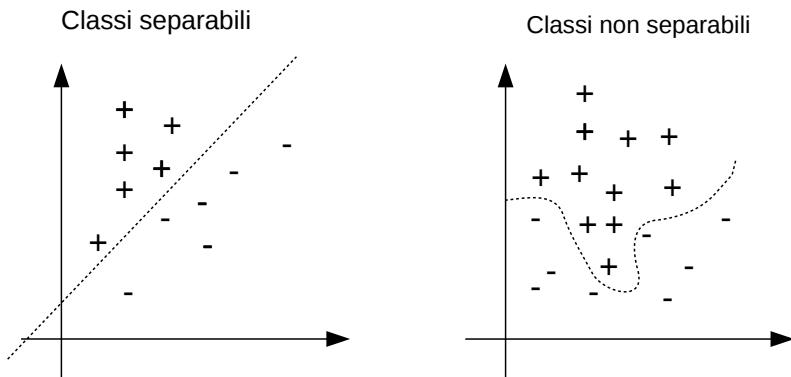
I risultati ottenuti nel paragrafo precedente sono stati storicamente molto interessanti e nel passato hanno dato uno stimolo importante alla ricerca nel campo della modellazione dei sistemi neurali. Sebbene il modello presentato abbia una singola uscita ed essa sia limitata ad assumere un valore binario (1 o 0), allineando più perceptroni è possibile realizzare una rete con un numero arbitrario di input e di output, potendo così rappresentare in forma digitale (sequenza binaria) qualsiasi output. La rete di perceptroni può quindi sembrarci, e sembrò in passato, la soluzione efficiente e completa ai modelli basati sull'idea di memoria associativa. Purtroppo essa non tardò a mostrare un

limite importante (se avete avuto pazienza di giocare un po' con il codice l'avrete già trovato): una rete ad un solo strato di percetroni è capace di distinguere solo tra dati linearmente separabili. Tornando all'esempio della fisionomia normolinea o longilinea, se invece di testare il percettrone con i dati di tabella XI, inserissimo quelli proposti nella tabella XII il sistema non sarebbe in grado di funzionare correttamente.

*Tabella XII*

#	IA	IP	Classe
Apprendimento			
1	1	7	1
2	3	1	0
3	3	9	1
4	4	3	0
5	2	3	0
6	7	3	1
7	2	5	0
8	7	5	1
Test			
9	2	6	1
10	3	7	0
11	8	4	0
12	9	6	1

Il fatto è che il percettrone, così come è definito, è capace solo di classificare dei dati che siano separabili da un iper-piano. Per esempio, se ci sono solo due classi, è necessario che, disposti i punti su un piano cartesiano, tra le due classi di punti passi una retta. Vedi un esempio di dati separabili ed uno di dati non separabili nella figura seguente (Fig. 3.5 bis).



*Fig. 3.5 bis - Confronto tra classi di dati separabili linearmente e non separabili linearmente.*

Questo limite può essere superato se si costruisce una rete di percettroni che siano tra loro connessi in modo che si crei uno strato (layer) di percettroni che faccia da "cuscino" tra il livello di input e quello di output. L'idea è presa sempre dalla biologia dove, come visto in precedenza, esistono dei neuroni, detti neuroni intercalari o interneuroni, che non sono né cellule deputate a ricevere gli stimoli dall'esterno né deputate a innescare un'azione, ma forniscono solo una "rappresentazione interna" degli stimoli esterni. Prima di affrontare questo discorso in dettaglio, vediamo come scrivere il codice C di un singolo percettrone.

### Sviluppo del codice per il percettrone singolo

Quando abbiamo trattato il problema del singolo percettrone abbiamo inserito la sua logica direttamente nella funzione main perché non c'era necessità di astrarre l'algoritmo in termini funzionali. Ora abbiamo l'obiettivo di realizzare una rete intera e a più strati di percettroni, per cui è conveniente *incapsulare* il codice che implementa la logica del percettrone in una funzione il cui uso ridurrà le linee di codice e aumenterà la leggibilità del codice. Propongo allora il prototipo che segue:

double perc\_calc\_output  
(double v\_w[],double v\_x[],int n\_dend);  
che incapsula completamente i calcoli che devono essere eseguiti  
e che riporto di seguito:

```
/* v_w: vettore pesi di dimensione n_dend+1
 * v_x: vettore input n_dend+1 (c'è il bias)
 * n_dend numero di dendriti */
double perc_calc_output
(double v_w[],double v_x[],int n_dend)
{
    double a=0;
    /*somma pesata degli stimoli di ingresso*/
    for(int i=0;i<n_dend+1;i++)
        a=a+v_w[i]*v_x[i];
    /*Attivazione del perceptron*/
    printf("\nInput %lf\n",a);
    for(int i=0;i<n_dend+1;i++)
        printf("%d) %lf*%lf=",i,v_w[i],v_x[i]);
    return a;
}
double activ_function
(double summed_input)
{
    double r=tanh(summed_input);
    return r;
}
double Dactiv_function
(double summed_input)
{
    double r=tanh(summed_input);
    return 1-r*r;
}
```

dove abbiamo introdotto le due funzioni `activ_function` e `Dactiv_function` che rappresentano la funzione di attivazione e la sua derivata rispettivamente. Queste ci serviranno tra poco e visto che sono legate al percepitrone le ho inserite già qui. A questo proposito vale la pena anticipare un risultato importante: affinché la nostra rete superi il limite del singolo percepitrone è necessario che la funzione di attivazione sia non-lineare, altrimenti anche un sistema a più strati si dimostra essere algebricamente riconducibile ad una rete composta da un singolo strato che può classificare solo dati appartenenti a classi separabili linearmente. Bene, detto ciò continuiamo con la progettazione di una rete a più strati.

### Una rete di percepitori con 3 ingressi e 2 uscite

In figura 3.6 ho rappresentato una schematizzazione del neurone/percepitrone che si astrae dai dettagli funzionali presenti in figura 3.4 mostrando solo i connettori necessari per collegare i percepitori in una rete. Sono evidenziati tre canali di input (dendriti) e una sola uscita di output. L'idea è quella di combinare insieme più percepitori portando l'uscita di alcuni verso l'entrata di altri. Lo faremo dandoci delle regole che renderanno il problema trattabile da un punto di vista matematico, ma non è sempre detto che abbiano una controparte biologica. (È importante sottolineare questi aspetti perché bisogna sapere quali limiti ci si auto impone per poi saperli superare al momento giusto.)

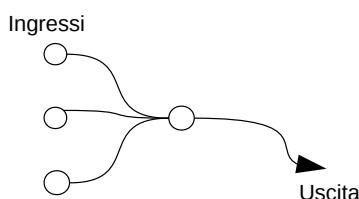
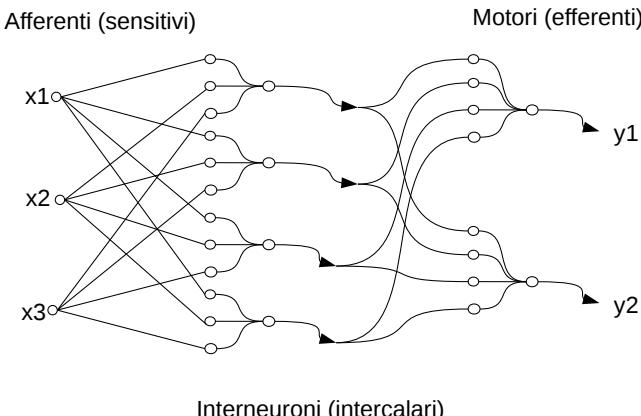


Fig. 3.6 - Schema del singolo percepitrone.

La prima regola è che esiste uno strato di neuroni che non fa nulla, riceve un input e lo trasmette ad altri neuroni, chiameremo questi neuroni sensitivi o afferenti e ci riferiremo ad essi come all'input della rete. Diciamo che questi siano in numero  $n$  e per l'esempio che porto qui (sempre meglio fare un esempio) diciamo che  $n = 3$ . Ognuno dei neuroni afferenti deve essere collegato ad un percettrone che deve avere esattamente lo stesso numero  $n$  di dendriti (canali di input verso il proprio centro, o soma). Stabilito che esiste uno strato di 3 neuroni di input abbiamo anche stabilito che esiste uno strato di percetroni (neuroni/percetroni) con 3 dendriti. Questo strato può essere costituito da un numero arbitrario di elementi. Chiamiamo questo lo strato profondo o *deep layer* e i percetroni di questo strato, neuroni intercalari o interneuronati. Questo è lo strato in cui la realtà esterna viene rappresentata nel sistema neurale. Ognuno dei canali di output dei neuroni intercalari, deve essere connesso ad un dendrite di un successivo strato di percetroni che chiameremo motori o efferenti, quindi i percetroni del secondo strato, o strato esterno, devono essere in numero uguale al numero di percetroni dello strato interno, nel nostro caso 3. Stabiliamo per questo esempio che il numero di percetroni esterni sia 2, quindi due percetroni con 4 dendriti ciascuno. In figura 3.7 è riportato uno schema della rete appena descritta. Una rete come questa, in cui nessun dendrite rimane privo di un input e tutti gli output sono portati ad un ingresso dendritico è detta essere completamente connessa.



*Fig. 3.7 - Rete multi strato di percetroni completamente connessi.*

Una volta inquadrato il sistema neurale come una rete di percetroni, può essere comodo pensarlo anche come una successione di strati o appunto layers. In questo caso ogni strato è completamente descritto dal numero di ingressi e dal numero di uscite che presenta. Con riferimento alla figura 3.8, si capisce che la numerosità dendritica dei neuroni di uno strato (numeri di canali di input per percettrone) è uguale al numero di ingressi della rete, mentre il numero di uscite determina il numero di neuroni (percetroni) presenti nello strato. Il valore dei neuroni dello strato di input rappresenta l'input per i neuroni dello strato profondo o interno. L'insieme degli output di questo strato è l'input per il secondo strato, per questo motivo si parla di *feed forward*, cioè un layer alimenta quello avanti a sé.

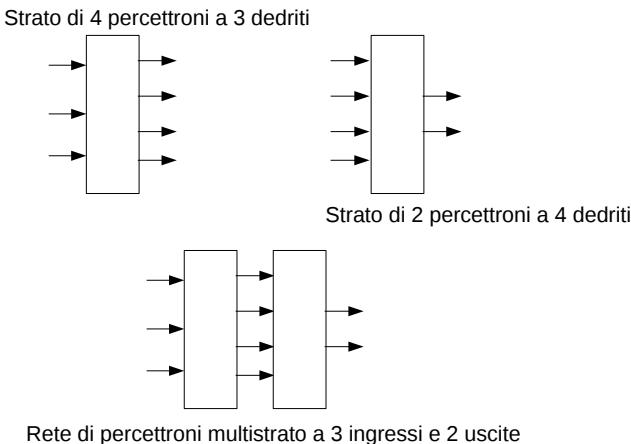
Ai fini di modellare una rete a strati in linguaggio C è conveniente sviluppare un modello, cioè una o più funzioni che astraggano il layer come una entità, quindi che permettano di modificarne l'input e acquisirne l'output senza dover agire su ogni singolo percettrone. Qui propongo una funzione che accetta in ingresso i pesi dei canali e l'input alla rete, e restituisce un array con l'output dello strato e con le somme interne, che come vedremo prestissimo, sono necessarie per calcolare la propagazione inversa.

```

/* v_s: vettore delle somme dei canali dendritici
per gli n_perc percetroni (out)
* v_y: vettore degli output per gli n_perc
percetroni (out)
* v_w: vettore dei pesi dendritici per gli n_perc
percetroni (in)
* v_x: vettore degli input al percettrone (uguale
per tutti gli n_perc percetroni) (in)
* n_perc: numero di percetroni nello strato (in)
* n_dend: numero di dendriti per percettrone (in)*/
void layer_feed_forward
(double v_s[],double v_y[],double v_w[],double
v_x[],int n_perc, int n_dend)
{
    for(int i=0;i {
        /*calcola l'output per ogni percettrone*/
        v_s[i]=perc_calc_output(v_w+i*
(n_dend+1),v_x,n_dend);
        v_y[i]=activ_function(v_s[i]);
    }
}

```

I parametri `v_s` e `v_w` sono array di scrittura, cioè sono buffer di memoria che passiamo alla funzione per permettergli di scrivere l'output a noi necessario.



*Fig. 3.8 - Schematizzazione della rete multistrato di percetroni in due strati.*

## Apprendimento supervisionato con propagazione inversa

L'idea dell'apprendimento supervisionato è già stata introdotta nei paragrafi precedenti quando abbiamo usato dei dati di apprendimento per istruire la rete. Di fatto abbiamo modificato i pesi delle connessioni sinaptiche in base alla differenza calcolata tra l'output prodotto e quello che si desiderava venisse prodotto. In pratica questo assomiglia al modo di educare o istruire una persona lasciandogli fare una esperienza a modo proprio, per poi mostrargli quale sarebbe dovuto essere il suo risultato in modo che si auto corregga. Facciamo l'esempio di un allenatore che senza spiegare la tecnica di calcio del rigore, lasci calciare un giocatore mostrandogli ogni tiro i metri di cui ha mancato la porta e lasciando che sia lui, di per sé a trovare la giusta direzione del piede. Il concetto di propagazione inversa nasce invece con l'introduzione nella rete di un secondo strato, il quale deve usare il *feedback* del primo per auto correggersi. In questo senso, il segnale di correzione viaggia in senso inverso rispetto al segnale

di input, da qui si ha il termine **propagazione inversa**. Vediamo di sviluppare un modello matematico per descrivere il processo di propagazione inversa.

Indichiamo con  $\mathbf{u}^{(i)}$  il vettore dei pesi sinaptici dell'i-esimo percettrone nello strato esterno (output) della rete, mentre con  $\mathbf{t}^{(i)}$  l'analogo vettore per lo strato interno o deep (vedi figura 3.7 bis).

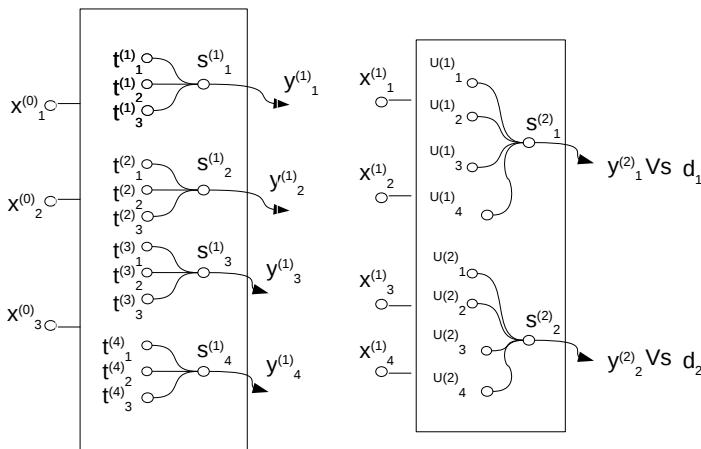


Fig. 3.7 bis - Rete a due strati di percettroni completamente connessi. Il primo strato (deep) presenta 4 percettroni a 3 ingressi, il secondo 2 percettroni a 4 ingressi. La figura evidenzia sia il modello a percettroni che la sua astrazione in strati. Il nome dei parametri del modello è messo in evidenza.

l'idea di base dell'apprendimento supervisionato è che per istruire i neuroni (percettroni) si deve:

**modificarne opportunamente il peso delle connessioni sinaptiche in modo che il segnale di uscita sia quanto più possibile vicino a quello desiderato.**

Per concretizzare questa idea si può utilizzare una tecnica matematica nota con il nome di *gradiente discendente* equivalente a quanto fatto nel paragrafo precedente, con la complicazione che ora l'intensità della variazione delle connessioni

di un percettrone dipende anche dal valore di uscita dei percettroni dello strato interno. Con tale tecnica la variazione dei pesi dendritici del percettrone  $i$ -esimo dello strato esterno è data da:

$$\Delta \mathbf{u}^{(i)} = \begin{bmatrix} 1 \times (d_i - y_i^{(2)}) f'(s_i^{(2)}) \\ y_1^{(1)}(d_i - y_i^{(2)}) f'(s_i^{(2)}) \\ y_2^{(1)}(d_i - y_i^{(2)}) f'(s_i^{(2)}) \\ y_3^{(1)}(d_i - y_i^{(2)}) f'(s_i^{(2)}) \\ y_4^{(1)}(d_i - y_i^{(2)}) f'(s_i^{(2)}) \end{bmatrix} \quad (3.10)$$

Nella (3.10) si noti (molto importante) che se l'uscita di un percettrone è uguale al valore desiderato per essa, allora i pesi delle sue connessioni con lo strato interno non vengono modificati, infatti, se l'uscita voluta e calcolata coincidono, si ha che il fattore  $(d_i - y_i^{(2)})$  è identicamente nullo. Le (3.10) sono frutto di una mia personale elaborazione matematica che ho voluto sviluppare per evidenziare il ruolo del singolo percettrone nella rete, anziché mostrare le equazioni che governano l'intera rete come spesso si trova in altri testi. Detto ciò ci tengo a sottolineare che queste equazioni non descrivono nessun modello biologico, esse hanno un puro uso didattico e la loro correttezza deve essere valutata solo rispetto all'efficacia dell'algoritmo in cui vengono usate.

Analogamente a quanto fatto per i percettroni dello strato esterno sviluppiamo ora l'equazione per calcolare la variazione delle connessioni del neurone  $i$ -esimo dello strato interno (deep layer). La logica matematica che ci porta a scrivere questa è un po' più complessa della tecnica a gradiente discendente. Infatti mentre per lo strato esterno abbiamo il "desired output" che ci permette di ri-calibrare la rete, qui non lo abbiamo. Come ho anticipato, lo strato interno è una sorta di rappresentazione della realtà, e se vogliamo provare a rimanere aderenti ai modelli neurali biologici,

(per lo meno in una certa misura) dobbiamo "inventarci" una tecnica di addestramento che preveda la conoscenza non solo dell'output desiderato ma anche di quello che dovrebbe essere la sua rappresentazione interna, il che forse, è ancora prematuro (a livello scientifico, intendo, non per questo testo). Pertanto anche per modificare i pesi dei percettroni interni dobbiamo usare lo scostamento dell'output dei percettroni esterni rispetto a quello desiderato, quindi di nuovo troveremo il fattore  $(d - y^{(2)})$ . Questo comunque non deve meravigliare. Supponiamo che una rete produca già il giusto output, vorremmo aspettarci che successivi cicli di addestramento non ne modifichino più le connessioni, quindi il fattore  $(d - y^{(2)})$  ci assicura che tali variazioni saranno sempre nulle una volta raggiunto l'obiettivo fissato dall'addestramento. Si parla di convergenza della rete!

Per calcolare le variazioni delle connessioni sinaptiche interne, calcoliamo anzi tutto la variazione desiderata dell'output del primo strato  $\Delta y^{(1)}$  come:

$$\Delta y^{(1)} = \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \mathbf{u}_4 \end{bmatrix} \begin{bmatrix} (d_1 - y_1^{(2)})f'(s_1^{(2)}) \\ (d_2 - y_2^{(2)})f'(s_2^{(2)}) \end{bmatrix}$$

dove si ricorda che:

$$\mathbf{u}_i = \begin{bmatrix} u_i^{(1)} & u_i^{(2)} \end{bmatrix}$$

per  $i = 1 : 4$ , o più intuitivamente:

$$\mathbf{u}^{(i)} = \begin{bmatrix} u_1^{(i)} \\ u_2^{(i)} \\ u_3^{(i)} \\ u_4^{(i)} \end{bmatrix}$$

per  $i = 1 : 2$ . Con questo si ha quindi:

$$\Delta y^{(1)} = \begin{bmatrix} u_1^{(1)}(d_1 - y_1^{(2)})f'(s_1^{(2)}) + u_1^{(2)}(d_2 - y_2^{(2)})f'(s_2^{(2)}) \\ u_2^{(1)}(d_1 - y_1^{(2)})f'(s_1^{(2)}) + u_2^{(2)}(d_2 - y_2^{(2)})f'(s_2^{(2)}) \\ u_3^{(1)}(d_1 - y_1^{(2)})f'(s_1^{(2)}) + u_3^{(2)}(d_2 - y_2^{(2)})f'(s_2^{(2)}) \\ u_4^{(1)}(d_1 - y_1^{(2)})f'(s_1^{(2)}) + u_4^{(2)}(d_2 - y_2^{(2)})f'(s_2^{(2)}) \end{bmatrix}$$

Definendo  $\Delta s^{(1)} = \Delta y^{(1)} \circ f'(s^1)$  si può definire la variazione del i-esimo vettore dei pesi sinaptici ( $\mathbf{t}^i$ ) per lo strato interno come:

$$\Delta \mathbf{t}^i = \begin{bmatrix} 1 \times (\Delta y_i^{(1)} f'(s_i^{(1)})) \\ x_1^{(0)}(\Delta y_i^{(1)} f'(s_i^{(1)})) \\ x_2^{(0)}(\Delta y_i^{(1)} f'(s_i^{(1)})) \\ x_3^{(0)}(\Delta y_i^{(1)} f'(s_i^{(1)})) \end{bmatrix} = \begin{bmatrix} 1 \times \Delta s_i^{(1)} \\ x_1^{(0)} \Delta s_i^{(1)} \\ x_2^{(0)} \Delta s_i^{(1)} \\ x_3^{(0)} \Delta s_i^{(1)} \end{bmatrix} \quad (3.11)$$

Con le (3.10) e (3.11) è possibile ora definire completamente il procedimento di apprendimento della rete. Prima di scrivere i passi dell'algoritmo introduciamo ancora il parametro  $\eta$ , detto appunto learning rate che, moltiplicato per la variazione dei vettori pesi, stabilirà la velocità di convergenza della rete. Ovviamente viene da chiedersi perché non porlo uguale ad uno, però in diverse situazioni, la tecnica del gradiente discendente porterebbe ad oscillazioni troppo "robuste" che potrebbero far "mancare" la soluzione. Un po' come se cercando un oggetto in una stanza buia tastando su un tavolo, muovessimo la mano con movimenti troppo grandi rispetto alla dimensione dell'oggetto cercato... rischieremmo di mancarlo!

Vediamo ora l'algoritmo di apprendimento della rete:

1. Carica i dati di input e l'output desiderato.
2. Calcola la risposta (output 1) dello strato 1 all'input caricato (input 0).
3. Calcola la risposta (output 2) dello strato 2 all'input (output 1).
4. Calcola lo scostamento tra l'output 2 e l'output desiderato.

5. Se lo scostamento è maggiore del limite di tolleranza:
  - o Modifica i pesi delle connessioni dello strato 1 e 2 come:  
$$\mathbf{t} = \mathbf{t} + \eta \Delta \mathbf{t}$$
 e  $\mathbf{u} = \mathbf{u} + \eta \Delta \mathbf{u}$ .
  - o Torna al punto 1.
6. Termina l'addestramento.

Quando l'esecuzione dell'algoritmo raggiunge il punto 6 la rete è stata addestrata e può essere usata per lo scopo per cui è stata progettata.

### 3.7 Un esempio in C di MLP: riconoscere le cifre digitali a sette segmenti

In questo paragrafo presento un esempio completo in C di una rete multistrato di percetroni, quella che viene comunemente chiamata una deep neural network. Si tratta di una rete a due strati che può essere configurata a piacere modificando il numero di dendriti e di percetroni. Il codice che vedremo è aperto ad ogni uso, ma qui ne presento un' applicazione per il riconoscimento di immagini. In realtà si tratta di un esempio molto semplice però la metodologia usata è scalabile e una volta appresa può essere usata per affrontare problemi più complessi.

L'idea è quella di addestrare la rete a riconoscere i numeri digitali composti da sette segmenti. Non ricordate più cosa sono o addirittura siete giovanissimi nerds che non li hanno mai visti? Nessun problema, guardate la prossima figura. I display a sette segmenti permettono di rappresentare graficamente le dieci cifre decimali e diverse lettere dell'alfabeto. In pratica un simbolo (lettera o cifra) può essere mostrato sul display attivando la opportuna combinazione di segmenti, come mostrato in figura 3.9 dove, a titolo di esempio, la cifra 3 corrisponde all'attivazione dei segmenti 1, 2, 3, 4, 5.

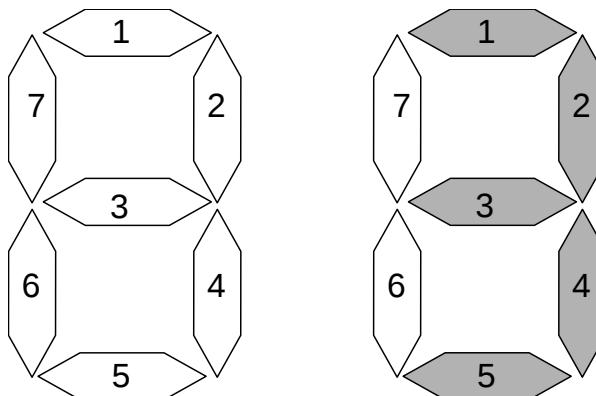


Fig. 3.9 - Display a sette segmenti.

## Analisi Top Down

Il programma usa due file di configurazione, in uno sono registrati i pesi iniziali delle connessioni sinaptiche dei percetroni usati, nell'altro i dati che si vogliono usare per addestrare la rete. I due file sono in formato testo e possono essere modificati a piacere, la loro struttura è descritta poco più avanti. I parametri che caratterizzano la rete, cioè il numero di percetroni del primo strato, il numero di dendriti dei percetroni, il numero di percetroni del secondo strato e il numero di dendriti dei percetroni di detto strato, sono definiti come macro `#define` prima della funzione `main`. Anche il numero di dati per il training, il numero di cicli di addestramento e il learning rate sono macro iniziali.

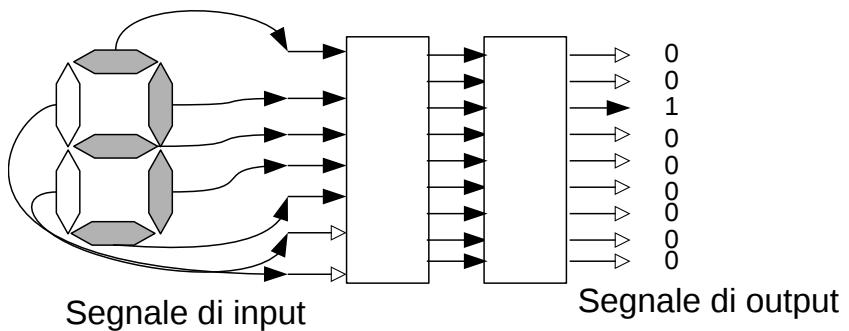
Allo start la `main` carica dal detto file i pesi iniziali delle connessioni poi inizia il ciclo di iterazioni per l'addestramento. Ad ogni ciclo il programma compie un altro ciclo sui dati usati per il training caricando ad ogni iterazione una coppia `input/output_desiderato`. Nella stessa iterazione il programma calcola l'`output` del primo strato, lo invia all'`input` del secondo strato, poi usando l'`output` del secondo strato e l'`output_desiderato` calcola la correzione dei pesi del primo e del secondo strato. Ad ogni ciclo di addestramento il programma stampa a video il numero di iterazioni raggiunto, poi per ogni iterazione sul ciclo dei dati, il programma stampa il valore di `output` confrontandolo con l'`output` desiderato.

Ovviamente la `main` non esegue tutti i compiti da sola, ma chiama le funzioni che incapsulano la logica della rete. Queste sono disposte su tre livelli logici, il primo è il livello della rete, cioè della rete vista come un'unità organica, il secondo è il livello percettrone, in cui sono esposte le funzionalità dirette dell'elemento neurale, il terzo è rappresentato dalla funzione di attivazione e dalla sua derivata prima, queste sono:

- layer\_feed\_forward
- layer\_map\_out\_in
- layer\_update
  - perc\_outlayer\_update
    - activ\_function
    - Dactiv\_function
  - perc\_deeplayer\_update
    - activ\_function
    - Dactiv\_function

La funzione main accede solo alle funzioni del primo livello e in pratica vede la rete neurale come due strati di neuroni senza entrare nel dettaglio di quanti essi siano. Le funzioni del primo strato invece accedono al secondo dove ho volutamente isolato il comportamento del singolo percepitrone per mostrare il suo ruolo nella rete. Lo scopo di questo "design" applicativo è quello di evidenziare la presenza del percepitrone e di mostrare la rete neurale come l'insieme connesso di più perceptroni.

## Configurazione e funzionamento



*Fig. 3.10 - Configurazione della rete per il riconoscimento delle cifre digitali. Il terzo neurone di output dall'alto è attivo in corrispondenza del numero tre presentato ai neuroni di input.*

Come mostrato in figura 3.10, la rete usa 9 percettroni da 7 dendriti nel primo strato e 9 da 9 dendriti nel secondo strato. Per inizializzare i pesi delle connessioni sinaptiche si deve creare un file (io l'ho chiamato init.txt) con i valori di queste. La struttura è la seguente:

```
#Layer1
<bias>
<peso 1>
<peso 2>
<peso 3>
<peso 4>
<peso 5>
<peso 6>
<peso 7>
...
#Layer2
<bias>
<peso 1>
<peso 2>
<peso 3>
<peso 4>
<peso 5>
<peso 6>
<peso 7>
<peso 8>
<peso 9>
...
```

dove la prima riga è un qualsiasi commento non più lungo di 200 caratteri e senza spazi (io ho scritto Layer1), le 7 righe successive devono riportare un numero che sarà usato per inizializzare il valore del bias e i pesi sinaptici (da 1 a 7) del primo percettore. Poi lo stesso, ma senza la riga di commento va ripetuto per tutti e 7 i percettroni del primo strato al posto dei

punti di sospensione. Finiti i valori dei pesi del primo strato si inserisce un commento con le stesse regole viste sopra e di seguito il bias e i pesi del secondo strato seguendo la stessa struttura vista per il primo, ma facendo attenzione che ora i pesi sono 9 (come i dendriti) e 9 sono anche i percettroni di questo strato. A titolo di esempio mostro la configurazione dei pesi del primo percettrone che ho usato io:

```
#Layer1
```

```
0.1
```

```
0.01
```

```
0.20
```

```
0.03
```

```
0.04
```

```
0.050
```

```
0.06
```

```
0.07
```

Per qualsiasi dubbio riguardo alla struttura dei file di configurazione, mi raccomando di consultare sempre il codice sorgente, perché anche se ci vorrà un po' di tempo quello è l'unico vero interessato a comprendere i dati che inserirete nei file.

Dopo aver editato il file init.txt è il momento del data.txt dove inseriamo i dati di input e l'output desiderato per il training.

La struttura di questo file è la seguente:

```
#DatoN1
```

```
<1 (fisso)>
```

```
<input 1>
```

```
<input 2>
```

```
<input 3>
```

```
<input 4>
```

```
<input 6>
```

```
<input 6>
```

```
<input 7>
```

```
#OutputDesideratoPerN1  
<output 1>  
<output 2>  
<output 3>  
<output 4>  
<output 5>  
<output 6>  
<output 7>  
<output 8>  
<output 9>
```

e deve essere ripetuta per tutti i campioni che si intende inserire. Io ho inserito nove campioni, uno per ogni configurazione dei "led" da 1 a 9. Riporto di seguito, su tre colonne i dati di input. Il file è scritto su tre colonne per non occupare inutilmente le pagine del testo e renderlo più leggibile, ma deve essere editato su una colonna sola.

*Tabella XIII. La tabella rappresenta il contenuto del file data.txt. Il contenuto è diviso su tre colonne solo per questioni di leggibilità.*

#L1_ND=7;L1_NP=9;L2_ND=9;L2_NP=4;RATE=0.01;N_T=50000; Input_7dendriti_N1	#Input_7dendriti_N4	#Input_7dendriti_N7
1	1	1
0	0	1
1	1	1
0	1	0
1	1	1
0	0	0
0	0	0
0	1	0
#desired_output	#desired_output	#desired_output
1	0	0
0	0	0
0	0	0
0	1	0
0	0	0
0	0	0
0	0	0
0	0	0
#Input_7dendriti_N2	#Input_7dendriti_N5	#Input_7dendriti_N8
1	1	1
1	1	1
1	0	1
1	1	1
0	1	1
1	1	1
1	0	1
0	1	1
#desired_output	#desired_output	#desired_output
0	0	0
1	0	0
0	0	0
0	0	0
0	1	0
0	0	0
0	0	0
0	0	0
#Input_7dendriti_N3	#Input_7dendriti_N6	#Input_7dendriti_N9
1	1	1
1	1	1
1	0	1
1	1	1
1	1	1
1	1	1
1	1	0
0	1	1
0	#desired_output	#desired_output
#desired_output	0	0
0	0	0
0	0	0
1	0	0
0	0	0
0	1	0
0	0	0
0	0	0
0	0	1

Nella prima riga di commento ho inserito anche i valori da impostare nelle macro. Analizziamo insieme un dato di input, per esempio il terzo che per comodità riporto qui sotto.

**#Input\_7dendriti\_N3**

```
1
1
1
1
```

```
1  
1  
0  
0  
#desired_output  
0  
0  
1  
0  
0  
0  
0  
0  
0  
0  
0
```

Se leggiamo la sequenza di numeri, vediamo che il primo 1 è fisso e, come spiegato nel paragrafo precedente, serve a rendere più compatta la scrittura dell'equazione (3.9). La successiva serie di sette numeri rappresentano i led da attivare (1) o disattivare (0) per ottenere il simbolo del numero 3 (vedi Fig. 3.9), quindi tutti 1 tranne gli ultimi 2. La sequenza di output è una classica classificazione. In pratica, volendo classificare il simbolo attivato in input, vogliamo che la rete apprenda a classificarlo tra 9 simboli (perché poi non ci ho messo lo zero? Mah, provate voi!). L'output desiderato corrisponde quindi con la "classe" che vogliamo venga attribuita al dato in input. Per esempio per il numero tre, vogliamo che siano a 0 tutte le classi tranne la terza, a cui appunto attribuiamo il significato di 3. Lo stesso ragionamento si applica agli altri numeri.

La rete fornisce in input nove classificazioni di tipi si/no. La prima ci dice se la configurazione di led inserita corrisponde ad un 1, la seconda ad un 2 e via dicendo. Il comportamento corretto della rete sarebbe quello che si attivasse la giusta classe in

corrispondenza della corretta configurazione dei led (dati di input) mentre tutte le altre dovrebbero disattivarsi.

Come potrete riscontrare, al termine dell'addestramento la rete è in grado di riconoscere correttamente le configurazioni di input per cui è stata addestrata. Cosa accade se inserite una configurazione nuova? Credo che a questo punto abbiate abbastanza materiale per ragionarci da soli! Ciò su cui invece vorrei portare l'attenzione è un altro aspetto: quanto serve lo strato "deep" in questa rete? Proviamo a rispondere a questa domanda nel prossimo paragrafo.

## Lo strato deep

Dopo aver articolato una rete così complessa, è naturale chiedersi se non si poteva affrontare il problema in modo più semplice usando i modelli di memoria associativa ad un solo strato visti all'inizio della sezione. Supponiamo allora che il vettore  $\mathbf{l}$  a sette componenti (uno per led) rappresenti la configurazione di input di un sistema ad un solo strato e che i pesi sinaptici verso lo strato di output siano memorizzati nella matrice  $\mathbf{W}$  di ordine  $9 \times 7$ . Il vettore a nove componenti  $\mathbf{c}$  rappresenta la classificazione dell'input. Di seguito riporto un esempio di associazione tra l'input  $\mathbf{l}$  e la corrispondente classificazione.

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Secondo il modello lineare visto in precedenza, il vettore di classificazione è dato dal prodotto

$$\mathbf{c} = \mathbf{W}\mathbf{l}$$

La domanda di inizio paragrafo può quindi essere ridotta ad una precisa domanda algebrica: i 9 vettori  $\mathbf{l}$  a 7 componenti possono dare come prodotto i 9 vettori a 9 componenti  $\mathbf{c}$ ? La risposta è no, e nasce direttamente dal carattere lineare delle equazioni. Vediamo che i 9 vettori  $\mathbf{l}$  sono un insieme **linearmente dipendente**, cioè alcuni di loro possono essere scritti come combinazioni degli altri. In particolare è sufficiente scegliere 7 di questi per rappresentare tutti e 9, questo mi pare anche intuitivo

se si considera la figura 3.9. Allora numeriamo i vettori  $\mathbf{l}$  da 1 a 9 come segue:  $\mathbf{l}^{(1)}, \mathbf{l}^{(2)}, \dots, \mathbf{l}^{(9)}$ . Con questa numerazione vediamo facilmente che si ha che la configurazione dell'otto può essere ottenuta da altre tre come:  $\mathbf{l}^{(8)} = \mathbf{l}^{(6)} + \mathbf{l}^{(9)} - \mathbf{l}^{(5)}$ . Supponiamo di aver configurato gli elementi della matrice  $\mathbf{W}$  in modo da avere:

$$\mathbf{W}\mathbf{l}^{(6)} = \mathbf{c}^{(6)},$$

$$\mathbf{W}\mathbf{l}^{(5)} = \mathbf{c}^{(5)},$$

$$\mathbf{W}\mathbf{l}^{(9)} = \mathbf{c}^{(9)},$$

avremo che

$$\mathbf{W}\mathbf{l}^{(8)} = \mathbf{W} \left( \mathbf{c}^{(6)} + \mathbf{c}^{(9)} - \mathbf{c}^{(5)} \right) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \neq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

che ovviamente non corrisponde alla classe dell'8.

La risposta è no, non è possibile risolvere questo problema con un solo sistema lineare quindi lo strato deep è necessario. Bisogna però porre attenzione su una cosa, se la funzione di attivazione che si implementa nello strato deep è una funzione lineare, allora una rete con strato interno (deep) è riconducibile ad un semplice sistema lineare, quindi non solo è necessario lo strato interno, ma anche l'uso di funzioni di attivazioni non lineari, come la tangente iperbolica usata in questo esempio.

## Convergenza della rete

L'ultimo aspetto che ci rimane da analizzare è la relazione esistente tra il learning rate ( $\eta$ ), il numero dei cicli di

addestramento e la configurazione iniziale dei pesi delle connessioni sinaptiche. Una trattazione organica non mi è possibile in questo contesto, ma vediamo che la logica intuitiva che tiene insieme queste grandezze è più semplice di quello che si potrebbe pensare. Abbiamo visto che le variazioni  $\Delta\mathbf{t}$  e  $\Delta\mathbf{u}$  sono sommariamente date da  $(d - y)f'(s)$ . Considerando che la  $f'(s)$  varia tra un minimo di 0 e un massimo di 1, la variazione dei pesi sinaptici per ogni ciclo di learning è dell'ordine  $\eta(d - y)$  quindi, se si itera per un numero  $n$  di cicli di apprendimento, si avrà una variazione massima dei pesi data da:  $n\eta(d - y)$ . Il prodotto  $n\eta(d - y)$  è quindi il parametro su cui porre l'attenzione nella prima fase di addestramento della rete, cioè la fase in cui si cerca di calibrare questi parametri. Il risultato di  $n$  iterazioni condotte con learning rate pari ad  $\eta$  modificherà i pesi di una grandezza dell'ordine  $(d - y)$ . Una volta impostati dei valori iniziali per i pesi, si valuta lo scostamento  $(d - y)$  tra l'output ottenuto e quello desiderato e si imposta di conseguenza il learning rate e il numero di iterazioni in modo che il prodotto  $\eta n$  sia dell'ordine dello scostamento misurato.

### Listato 3.5 display\_7segmenti.c

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

/* Strato (layer) 1*/
#define L1_ND 7
#define L1_NP 9

/* Strato (layer) 2*/
#define L2_ND 9
#define L2_NP 9

/* Numero campioni di addestramento */
#define N_C 9

/* Numeri di ripetizioni dell'addestramento */
#define N_T 1000

/* Velocità apprendimento*/
```

```

#define RATE 0.01

/*** Livello 1 ***/

/* Calcola l'output di uno strato percepironi*/
void layer_feed_forward
(double v_s[],double v_y[],double v_w[],double v_x[],int n_perc, int n_dend);
/* Mappa l'output v_y in v_x aggiungendo prima l'elemento v_x[0]=1*/
void layer_map_out_in
(double v_x[],double v_y[], int n_dend);
/* Calcola la variazione delle connessioni sinaptiche */
void layer_update
(double v_u[], double v_y2[], double v_x1[],double v_s2[],
double v_d[],double v_t[],double v_x0[],double v_s1[],
double rate,int l2nd, int l2np,int l1nd );

/*** Livello 2 ***/

/* Calcola la risposta del percettrone*/
double perc_calc_output
(double v_w[],double v_x[],int n_dend);
/* Calcola la variazione delle connessioni dendritiche del out layer e le
aggiorna */
void perc_outlayer_update
(double v_w[],double y,double v_x[],double z,double d,double rate,int n_dend);
/* Calcola la variazione delle connessioni dendritiche del deep layer e le
aggiorna */
void perc_deeplayer_update
(double v_w_t[],double v_w_u[],double v_y_2[],double v_x[],
double z,double v_s_2[],double d[],double rate,
int n_dend,int n_perc_2);

/*** Livello 3 ***/

/* Calcola il valore della risposta del percettrone*/
double activ_function
(double summed_input);
/* Calcola il valore della derivata della risposta del percettrone*/
double Dactiv_function
(double summed_input);

int main()
{
    /*Velocita' apprendimento*/
    double rate=RATE;

/*Strato 1*/
    double v_x0[L1_ND+1];/* input dei percetroni del layer 1*/
    double v_t[(L1_ND+1)*L1_NP];/* NP vettori di peso dendritico*/
    double v_Dt[(L1_ND+1)*L1_NP];/* Variazione v_t */
    double v_s1[L1_NP]; /*NP valori input*/

```

```

double v_y1[L1_NP];/* NP output uno per percettrone*/

/*Strato 1*/
double v_x1[L2_ND+1];/* input dei percetroni del layer 2*/
double v_u[(L2_ND+1)*L2_NP];/* NP vettori di peso denditrico*/
double v_Du[(L2_ND+1)*L2_NP];/* Variazione v_u*/
double v_s2[L2_NP]; /*NP valori input*/
double v_y2[L2_NP];/* NP output uno per percettrone*/

/*Output desiderato*/
double v_d[L2_NP];/* NP output desiderato uno per percettrone*/

/*Inizializza la rete dal file di input*/
FILE * f=fopen("init.txt","rt");
if(f==0) exit(1);
char comment[300];

/*Carica dal file le configurazioni iniziali della rete*/

/* bias+pesi strato 1*/
fscanf(f,"%s",comment);
for(int i=0;i<L1_NP;i++)
    for(int j=0;j<L1_ND+1;j++) fscanf(f,"%lf",v_t+i*(L1_ND+1)+j);

/* bias+pesi strato 2*/
fscanf(f,"%s",comment);
for(int i=0;i<L2_NP;i++)
    for(int j=0;j<L2_ND+1;j++) fscanf(f,"%lf",v_u+i*(L2_ND+1)+j);

fclose(f);

for(int k=0;k<N_T;k++){
    printf("\n\n\n*** Ciclo di training %d ***",k);
    f=fopen("data.txt","rt");
    if(f==0) exit(1);

/*Carica i dati di training ed esegue il training*/
    for(int nr=0;nr<N_C;nr++)
    {
/* 1+input*/
        fscanf(f,"%s",comment);
        for(int i=0;i<L1_ND+1;i++) fscanf(f,"%lf",v_x0+i);

/*  output desiderato*/
        fscanf(f,"%s",comment);
        for(int i=0;i<L2_NP;i++)
        {
            fscanf(f,"%lf",v_d+i);
        }
    }
}

```

```

    printf("\n-Dato n. %d ",nr+1);

/* Feed Forward: Input->L1->output to L2*/
    layer_feed_forward(v_s1,v_y1,v_t,v_x0,L1_NP,L1_ND);

/* Mappa y1 in x1 aggiungendo l'elemento x1_0=1*/
    layer_map_out_in(v_x1, v_y1,L2_ND);

/* Feed Forward: L1->L2->output*/
    layer_feed_forward(v_s2,v_y2,v_u,v_x1,L2_NP,L2_ND);

/* Propagazione inversa dell'errore*/
    layer_update(v_u,v_y2,v_x1,v_s2,v_d,v_t,v_x0,
    v_s1,rate,L2_ND, L2_NP,L1_ND);

/*Stampa risultato*/
    for(int i=0;i<L2_NP;i++)
    {
        int b;
        if(v_y2[i]>=0.4)b=1;else b=0;
        printf("%0.1lf (%d) vs %0.1lf\t",v_y2[i],b,v_d[i]);
    }
}
fclose(f);
}

}

/* v_s: vettore delle somme dei canali dendritici per gli n_perc percetroni
 * v_y: vettore degli output per gli n_perc percetroni
 * v_w: vettore dei pesi dendritici per gli n_perc percetroni
 * v_x: vettore degli input al percettrone (uguale per tutti gli n_perc
percetroni)
 * n_perc: numero di percetroni nello strato
 * n_dend: numero di dendriti per percettrone */
void layer_feed_forward
(double v_s[],double v_y[],double v_w[],double v_x[],int n_perc, int n_dend)
{
    for(int i=0;i<n_perc;i++)
    {
/*calcola l'output per ogni percettrone*/
        v_s[i]=perc_calc_output(v_w+i*(n_dend+1),v_x,n_dend);
        v_y[i]=activ_function(v_s[i]);
    }
}

void layer_update
(double v_u[], double v_y2[], double v_x1[],double v_s2[], double v_d[],
double v_t[],double v_x0[],double v_s1[],double rate,int l2nd, int l2np,int

```

```

l1nd)
{
/* Strato 1*/
/* ciclo sui percetroni */
for(int i=0;i<L2_NP;i++)
{
    perc_outlayer_update
    ( v_u+i*(L2_ND+1),v_y2[i],v_x1,v_s2[i],v_d[i], rate,L2_ND);
}

/* Strato 2*/
/* ciclo sui percetroni */
for(int i=0;i<L1_NP;i++)
{
    perc_deeplayer_update
    (v_t+i*(L2_ND+1),v_u+i*(L1_ND+1),v_y2,v_x0,v_s1[i],v_s2,
     v_d, rate,L1_ND,L2_NP);
}
}

void perc_outlayer_update
(double v_w[],double y,double v_x[],double z,double d,double rate,int n_dend)
{
/* ciclo sui dendriti */
    v_w[0]=v_w[0]+rate*(1*(d-y)*Dactiv_function(z));
    for(int i=1;i<n_dend+1;i++)
    {
        v_w[i]=v_w[i]+rate*v_x[i]*(d-y)*Dactiv_function(z);
    }
}

/* v_w_t vettore pesi layer 1, v_w_u vettore pesi layer 2
n_perc_2 percetroni di output
*/
void perc_deeplayer_update
(double v_w_t[],double v_w_u[],double v_y_2[],double v_x[],
 double z,double v_s_2[],double v_d[],
 double rate,int n_dend,int n_perc_2)
{
/* ciclo sui dendriti */
    for(int i=0;i<n_dend+1;i++)
    {
        double dd=0;
        for(int j=0;j<n_perc_2;j++)
        {
            dd=dd+v_w_u[j]*(v_d[j]-v_y_2[j])*Dactiv_function(v_s_2[j]);
        }
        v_w_t[i]=v_w_t[i]+rate*v_x[i]*dd*Dactiv_function(z);
    }
}

```

```

}

void layer_map_out_in
(double v_x[],double v_y[], int n_dend)
{
    v_x[0]=1;
    for(int i=1;i<n_dend+1;i++)v_x[i]=v_y[i-1];
}

/* v_w: vettore di dimensione n_dend+1 di pesi dendritici
 * v_x: vettore delgi n_dend+1 (c'è il bias) input al percettrone
 * n_dend numero di dendriti */
double perc_calc_output
(double v_w[],double v_x[],int n_dend)
{
    double a=0;
    /*somma pesata degli stimoli di ingresso*/
    for(int i=0;i<n_dend+1;i++) a=a+v_w[i]*v_x[i];

    /*Attivazione del percettrone*/
    return a;
}
double activ_function
(double summed_input)
{
    double r=tanh(summed_input);
    return r;
}
double Dactiv_function
(double summed_input)
{
    double r=tanh(summed_input);
    return 1-r*r;
}

```

### **Compilazione** display\_7segmenti.c

```

~/Documents/nn> gcc -o dspl7s display_7segmenti.c
-lm
~/Documents/nn> chmod u+x dspl7s
~/Documents/nn>./dspl7s

```

Nota bene, per compilare questo sorgente devi aggiungere l'opzione `-lm` alla linea di comando, perché Linux (se compili su Linux) non linka automaticamente la libreria matematica.

## Addestrare e provare la rete

Per addestrare e testare la rete seguiamo gli stessi passi visti nel paragrafo precedente.

1. Editiamo il file sorgente in un editor di testo e salviamolo nella cartella nn con il nome `display_7segmenti.c`
2. Compiliamo il sorgente.
3. Editiamo, sempre con l'editor di testo, un file che chiameremo `mlp_init.txt` con il contenuto già discusso.
4. Editiamo, sempre con l'editor di testo, un file che chiameremo `mlp_data.txt` con il contenuto già visto.
5. Dal prompt dei comandi lanciamo l'eseguibile `dspl7s`

L'esecuzione del programma mostra lo stato della rete per ogni ciclo di addestramento, confrontando la classificazione operata dalla rete rispetto alla classificazione corretta. Per non appesantire il codice già molto lungo non ho inserito la fase di test dopo la fase di addestramento, ma guardando i cicli di addestramento si vede che circa dal numero 975 la rete è già istruita, si possono considerare i successivi come test.

### 3.8 Riconoscimento di scrittura a mano

L'esempio sviluppato del capitolo precedente è senz'altro affascinante, ma in effetti è privo di una reale utilità in quanto le cifre rappresentate con sette segmenti sono riconoscibili facilmente da un task specific algorithm. In questo capitolo svilupperemo di nuovo esempio di riconoscimento delle cifre numeriche, ma questa volta svilupperemo un MLP capace di riconoscere delle cifre scritte a mano (*handwritten*). L'obiettivo è quindi quello di addestrare una rete neurale a più strati a classificare delle immagini scegliendo tra dieci possibili classi (da 0 a 9). Le immagini di input però non seguono uno schema preciso come nel caso del display a sette segmenti, ma solo il

risultato della scrittura umana. Ovviamente, per essere processate dal MLP, le immagini devono essere digitalizzate.

## Preparazione dell'esperimento

Per addestrare la rete al riconoscimento della scrittura manuale (limitata alle cifre decimali) è necessario disporre di un campione di dati, cioè un insieme di immagini digitali rappresentati delle cifre scritte a mano.

Possiamo preparare il campione da noi, oppure, come vedremo nel prossimo paragrafo, possiamo usarne uno prodotto dal NIST (l'ente per gli standard negli USA) che si trova disponibile in rete.

Per produrre le immagini digitali delle cifre si possono seguire due strade. La prima consiste nell'scrivere a mano una cifra su un foglio di carta per poi acquisire l'immagine con una fotocamera e quindi dall'immagine estrarre la matrice dei valori di intensità luminosa di ogni pixel, come mostrato in figura 3.11.

Sebbene questo sia il metodo più diretto, per poter estrarre i valori dei pixel è necessario avere una conoscenza più approfondita del formato delle immagini digitali (jpeg, png, GIF, ecc.) oppure (come per il caso mostrato in figura) ottenere tali valori direttamente dal dispositivo hardware usato per acquisire l'immagine. Anche questa seconda opportunità è piuttosto complessa (per chi sta iniziando), comunque, per chi è curioso di vedere come si può fare, al tempo in cui scrivo queste righe, il codice per estrarre l'immagine direttamente dalla webcam è presente sulla mia pagina di GitHub.

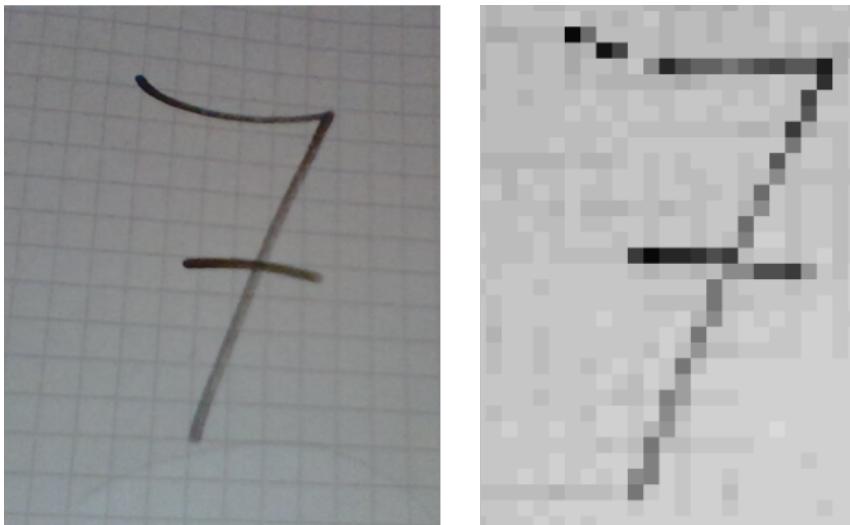
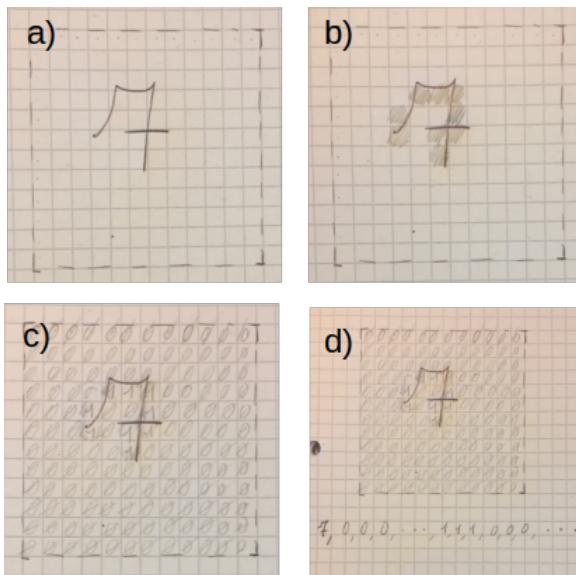


Fig. 3.11 - Esempio di digitalizzazione della cifra 7 scritta a mano.

La seconda consiste nello scrivere i numeri uno ad uno su un foglio di carta all'interno di un quadrato di una certo lato, per esempio di 12 quadretti, poi di selezionare i quadretti in cui è presente la traccia della cifra. Fatto questo partendo dal primo quadretto in alto a destra si passa tutto il quadrato quadretto per quadretto, assegnando il valore 0 ai quadretti vuoti e il valore 1 ai quadretti segnati. Completata questa procedura si apre un file di testo in cui si scrivono di seguito tutti i valori segnati (sempre da sinistra a destra e dall'alto in basso) separandoli con una virgola. Prima di iniziare la scrittura della sequenza di 0 e 1 si scrive il valore della cifra, cioè il *ground truth* visto in precedenza. La sequenza delle operazioni necessarie per eseguire questa procedura è riportata in figura 3.12.

Provare a produrre il campione di dati di addestramento seguendo questa metodologia ha sicuramente un elevato valore didattico, ma va da sé che non è un metodo efficiente. Per addestrare significativamente la rete sono necessarie alcune migliaia di immagini, e per quanta pazienza si possa avere, questo è un compito spropositato da completare in questo modo. Pertanto,

dopo aver preso confidenza con le idee e i concetti che sono stati esposti, consiglio di proseguire con il paragrafo successivo, dove spiegherò come fare uso di un database di dati già preconfezionato e pronto all'uso, il database del NIST.



*Fig. 3.12 - Procedura per la produzione dei dati di addestramento per una MLP atta a riconoscere le cifre. a) scrittura manuale di una cifra, b) selezione dei quadretti di interesse, c) valorizzazione dei quadretti a 0 e 1, d) scrittura della sequenza di 0 e 1 rappresentati la cifra, preceduta dal valore decimale della cifra stessa.*

## Il database MNIST

L'istituto statunitense NIST preparò un primo database con circa 60000 immagini digitalizzate di cifre scritte a mano ed un secondo database con circa 10000.

Il primo database viene normalmente usato per addestrare i sistemi AI (come il MLP) a riconoscere le cifre, mentre il secondo viene usato come test.

I database originali possono essere trovati all'indirizzo: <http://yann.lecun.com/exdb/mnist/> qui si possono scaricare

quattro file

- `train-images-idx3-ubyte.gz`: training set images (9912422 bytes)
- `train-labels-idx1-ubyte.gz`: training set labels (28881 bytes)
- `t10k-images-idx3-ubyte.gz`: test set images (1648877 bytes)
- `t10k-labels-idx1-ubyte.gz`: test set labels (4542 bytes)

i primi due sono rispettivamente il file con le immagini di addestramento e le label (ground truth) per ogni immagine. I secondi due sono sempre le immagini ma di test con le relative label.

Le immagini sono in formato *raw*, cioè sono la pura sequenza dei valori di ogni pixel da sinistra a destra e dall'alto in basso. Ogni pixel è rappresentato da un valore di tipo `unsigned byte` cioè in C `unsigned char`. Dopo l'ultimo byte di ogni immagine comincia l'immagine successiva. Prima delle immagini vere e proprie, nel file è presente un *header*, cioè una testata, di 16 byte che riporta i dati di intestazione, cioè:

- il così detto *magic number* che può essere usato da un'applicazione per riconoscere lo scopo del file
- il numero di immagini presenti nel file
- il numero di righe per immagine
- il numero di colonne per immagine

Le label hanno un formato simile: un header di 8 byte che riporta il magic number e il numero totale di label precede la sequenza di byte il cui valore è l'etichetta della corrispondente immagine sul file delle immagini. Quindi, riassumendo, nel file `train-images-idx3-ubyte` si trovano in sequenza 60000 immagini rappresentati le cifre decimali (da 0 a 9) scritte a mano, mentre nel file `train-labels-idx1-ubyte` ci sono le corrispondenti *etichette* che servono per fornire alla rete il feedback sulla propria performance.

Gli stessi dati presenti nel database MNIST sono stati riprodotti in formato CSV e sono disponibili sul sito <https://pjreddie.com/projects/mnist-in-csv/>

Troviamo solo due file indicati indicati dai link *train set* e *test set*. Il vantaggio di questi è che sono di più immediata lettura e possono essere rapidamente ispezionati usando un foglio di calcolo come Calc o Excel.

Il codice che verrà presentato tra breve, assume che i dati siano in formato CSV, e che i file di addestramento e test si chiamino `mnist_train.csv` e `mnist_test.csv` rispettivamente.

Comunque, se si preferisce usare il formato originale del MNIST, è possibile farlo modificando il codice di conseguenza.

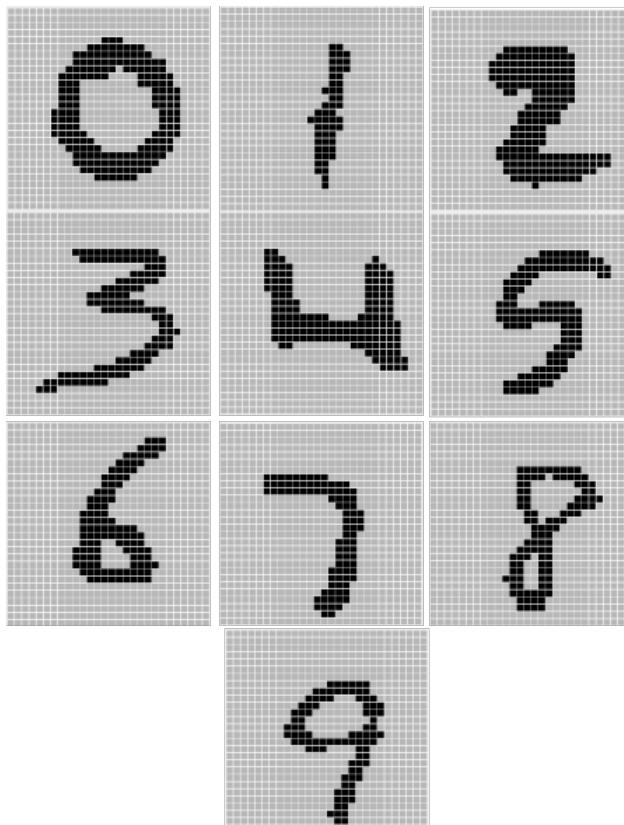


Fig. 3.11 - Esempi di dieci cifre decimali scritte a mano e digitalizzate in griglie di 28x28 pixel.

## Il problema della persistenza

Prima di presentare il codice del MLP destinato a classificare la calligrafia umana nelle dieci cifre decimali, affrontiamo l'aspetto della persistenza dello stato della rete.

Le reti neurali biologiche hanno la caratteristica di essere plastiche, cioè di poter variare l'intensità delle connessioni tra le sinapsi in base all'esperienza. Questa caratteristica è alla base sia della capacità di apprendimento che della capacità di recupero del cervello.

In questo testo, abbiamo visto come la caratteristica di plasticità venga simulata nelle reti neurali artificiali memorizzando i pesi sinaptici in strutture dati modificabili.

Abbiamo visto che durante l'esecuzione di un programma che implementa una rete neurale, il codice agisce principalmente per assegnare i giusti valori alle connessioni sinaptiche, fino al raggiungimento odi un grado di addestramento giudicato sufficiente.

Sebbene da un punto di vista didattico questo possa essere considerato esaustivo, se si vuole usare una rete neurale nella pratica, bisogna considerare che dopo la fase di addestramento esisterà una fase di lavoro, in cui verranno sfruttati i valori raggiunti dai pesi delle sinapsi.

Al termine dell'esecuzione del programma che ha addestrato la rete, lo stato della rete è perso, e si si vuole usare ancora la rete è necessario addestrarla nuovamente. Visto che il processo di addestramento della rete potrebbe variare dalla durata di alcuni minuti fino a giorni interi, è naturale pensare ad un modo per evitare di dover ripetere il processo di addestramento ogni volta che la si vuole riusare.

Il metodo più naturale è quello di scrivere in un file i valori corrispondenti ai pesi sinaptici della rete dopo il suo addestramento. In questo modo, al successivo utilizzo, non sarà

più necessario ripetere la procedura di addestramento, ma sarà sufficiente leggere in memoria dal file detti valori e usarli direttamente senza averli ricalcolati.

Siccome la rete che useremo ha tre strati di neuroni: input, deep e output, avremo due livelli di interconnessione, cioè le connessioni tra input e deep e quelle tra deep e output, quindi useremo due file per rappresentare tali connessioni. Ricordo, che la scelta di limitare le connessioni tra gli strati ai soli strati adiacenti, è solo una scelta topologica, e non è in sé obbligatoria. Ci sono reti che prevedono il collegamento dello strato di input direttamente allo strato di output anche in presenza di uno strato deep.

## Il codice C di addestramento

Rispetto a quanto visto fin'ora, il codice per questa rete presenta una piccola novità, che oltre ad essere funzionale alla scrittura del programma, completerà anche parte della conoscenza del linguaggio C.

Fino adesso tutti i codici presentati erano contenuti in un unico file. Il C permette però anche la programmazione *modulare*, cioè il codice sorgente può essere diviso in più file a cui spesso ci si riferisce come moduli.

La divisione in moduli può essere utile per diverse ragioni, per esempio separando un file molto lungo in diversi moduli è più facile mantenerlo ordinato, o all'occorrenza, permettere a più persone di modificare indipendentemente parti diverse dello stesso programma, scrivendo su moduli separati. Possiamo fare una analogia con un'auto vettura, in cui meccanici, gommisti e carrozzieri, possono intervenire simultaneamente, senza generare conflitti sui componenti (moduli) in lavorazione.

Un altro vantaggio, ancora più importante, è che separando il codice sorgente in file diversi, è possibile ri-usare le funzioni definite in un modulo per un programma differente.

Vediamo di chiarire con un esempio. Supponiamo di dover scrivere il programma A nel file A.c e il programma B nel file B.c. Supponiamo che i due programmi condividano una parte della logica applicativa, allora possiamo scrivere un programma C, nel file C.c in modo che il programma A sia ora scritto nei due file A.c e C.c e il programma B nei due file B.c e C.c.

In realtà questo è quello che abbiamo fatto fin'ora includendo le librerie come la stdlib.h con la differenza che chi ha scritto il compilatore GCC ha già fatto questo lavoro per noi, mentre questa volta saremo noi a modularizzare il nostro codice per estrarne una libreria di funzione da riusare.

Quello che faremo è di estrarre, dal codice sviluppato prima, le funzioni base che servono a gestire una MLP network, per poi usarle in due diversi programmi. Il primo che chiameremo addestra\_lettore.c ha lo scopo di addestrare la rete a riconoscere le cifre manoscritte e di salvare su file lo stato della rete addestrata. Invece il secondo, che chiameremo becnhmark\_lettore.c ci serve per testare la rete e stabilire la percentuale di successo con cui essa classifica le immagini. Dal secondo programma, sarà immediato estrarre un eventuale codice per usare la rete in diverse situazioni applicative.

Al lettore più attento non sarà sfuggito un problema. Come abbiamo visto nei capitoli precedenti, il compilatore C ha bisogno di conoscere la definizione delle funzioni prima di poterle richiamare. In pratica, tornando al nostro esempio precedente, perché compilatore possa compilare il codice scritto in A.c che chiama le funzioni definite in C.c, ha bisogno di conoscere la testata di dette funzioni. Per risolvere questo problema, si usa scrivere i prototipi delle funzioni *libreria* in file header, appunto quelli che terminano con l'estensione .h. Quindi accanto al file C.c avremo anche il file C.h che riporterà le intestazioni delle funzioni presenti in C.c. I file header generati dall'utente (programmatore) devono essere inclusi nel sorgente come si fa

per i file della libreria standard, cioè usando la direttiva di precompilazione `#include`, con la differenza che il nome del file non va tra parentesi angolari ma tra apicetti, come sarà mostrato tra breve.

Il primo listato che vediamo è quello del file header della libreria. A questa diamo il nome di `ss_snn_lib`, dove ss sta per Scuola Sisini, snn sta per shallow neural network e lib ovviamente per libreria.

### Listato 3.6 ss\_snn\_lib.h

```
/*
 | ss_snn_lib
 | Scuola Sisini Shallow Neural Network Library
 | Francesco Sisini (c) 2019
 */

/*** Livello 1 ***/
/* Calcola l'output di uno strato percettroni */
void layer_feed_forward
(double v_s[],double v_y[],double v_w[],double v_x[],int n_perc, int n_dend);

/* Mappa l'output v_y in v_x aggiungendo prima l'elemento v_x[0]=1*/
void layer_map_out_in
(double v_x[],double v_y[], int n_dend);

/*** Livello 2 ***/
/* Calcola la risposta del percettrone */
double perc_calc_output
(double v_w[],double v_x[],int n_dend);

/* Corregge i pesi del percettrone */
void perc_correzione
(double v_w[],double v_x[],double z,double d,double rate,int n_dend);

/*** Livello 3 ***/
/* Calcola il valore della risposta del percettrone*/
double activ_function
(double summed_input);

/* Calcola il valore della derivata della risposta del percettrone*/
double Dactiv_function
(double summed_input);
```

```

/* Legge un immagine 28x28 in o e la sua label in out_label */
int get_image
(int * o,int * out_label,FILE * in_stream);

/* Stampa a video una matrice r x c in R,C */
void print_object
(double x[],int r, int c,int R,int C);

/*
_____
| stream: il file su cui scrivere
| v_w: l'array sequenziale con tutti i pesi del layer
| n_dend: numero di dendriti per percettrone
| n_perc: numero di percetroni nel layer
*/
void layer_writedown
(FILE * stream,double *v_w, int n_dend, int n_perc);

/*
_____
| stream: il file da cui legge i pesi delle connessioni
| v_w: l'array sequenziale con tutti i pesi del layer
| n_dend: numero di dendriti per percettrone
| n_perc: numero di percetroni nel layer
*/
void layer_read
(FILE * stream,double *v_w, int n_dend, int n_perc);

```

Come si vede dai prototipi, alcune delle funzioni che verranno definite in questa libreria sono le stesse già definite nell'esempio del display a sette segmenti, altre sono modificate, e altre ancora invece sono completamente nuove.

Vediamo anzitutto le modifiche alle funzioni `perc_deeplayer_update` e `perc_outlayer_update` sono sostituite dalla funzione `perc_correzione`, evidenziando che ad ogni layer è possibile applicare lo stesso algoritmo di *apprendimento* per le connessioni tra i neuroni.

Per quanto riguarda invece le funzioni nuove, abbiamo la funzione `get_image` che legge i dati delle immagini e delle label dal file di addestramento o di test. Le due funzioni `layer_writedown` e `layer_read` che servono invece l'una a scrivere e l'altra a rileggere lo stato della rete dopo l'addestramento.

La definizione di tutte le funzioni è riportata nel sorgente che segue.

### Listato 3.7 ss\_snn\_lib.c

```
/*
| ss_snn_lib
| Scuola Sisini Shallow Neural Network Library
| Francesco Sisini (c) 2019
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "ss_snn_lib.h"

#define D_SIZE 784

void layer_writedown
(FILE * stream,double *v_w, int n_dend, int n_perc)
{
    fwrite(v_w, sizeof(double), n_dend*n_perc, stream);
}

void layer_read
(FILE * stream,double *v_w, int n_dend, int n_perc)
{
    fread(v_w, sizeof(double), n_dend*n_perc, stream);
}

void layer_feed_forward
(double v_s[],double v_y[],double v_w[],double v_x[],int n_perc, int n_dend)
{
    for(int i=0;i<n_perc;i++)
    {
        /*calcola l'output per ogni percettore*/
        v_s[i]=perc_calc_output(v_w+i*(n_dend+1),v_x,n_dend);
        v_y[i]=activ_function(v_s[i]);
    }
}

void perc_correzione
(double v_w[],double v_x[],double z,double d,double rate,int n_dend)
{
    /* ciclo sui dendriti */
    for(int i=0;i<n_dend+1;i++)
    {
```

```

    v_w[i]=v_w[i]+rate*v_x[i]*(d)*Dactiv_function(z);
}
}

void layer_map_out_in
(double v_x[],double v_y[], int n_dend)
{
    v_x[0]=1;
    for(int i=1;i<n_dend+1;i++)v_x[i]=v_y[i-1];
}

double perc_calc_output
(double v_w[],double v_x[],int n_dend)
{
    double a=0;
    /*somma pesata degli stimoli di ingresso*/
    for(int i=0;i<n_dend+1;i++) a=a+v_w[i]*v_x[i];

    /*Attivazione del percepitrone*/
    return a;
}

double activ_function
(double summed_input)
{
    double r=1/(1+exp(-summed_input));
    return r;
}

double Dactiv_function(double summed_input)
{
    double r=activ_function(summed_input);
    return r*(1-r); //corretta, fare i conti!
}

int get_image
(int * o,int * out_label,FILE * in_stream)
{
    int r = fscanf(in_stream,"%d",out_label);

    for(int i=0; i<D_SIZE; i++)
        r += fscanf(in_stream,"%d",o+i);

    return r;
}

void print_object
(double x[],int r, int c,int R,int C)
{
    double min,max;

```

```

min=10;
max=-10;

for(int i=0;i<r;i++)
    for(int j=0;j<c;j++)
    {
        if(x[i*c+j]>max) max=x[i*c+j];
        if(x[i*c+j]<min) min=x[i*c+j];
    }

double i_range=max-min;

double max_c=255;
double min_c=0;
double c_range=max_c-min_c;
double conv=c_range/i_range;

for(int i=0;i<r;i++)
    for(int j=0;j<c;j++)
    {
        double gl;
        gl=x[i*c+j];

        int col=max_c-(min_c+(double)(gl-min)*conv)+min_c;
        if(col<0)printf("%f",gl);
        //if(gl>0.9) col=0;
        printf("\x1b[%d;%dH\x1b[48;5;%dm \x1b[0m",i+R,2*j+C,col);
    }
fflush(stdout);
}

```

Il codice appena definito non produce un eseguibile perché in esso manca la funzione `main`, infatti, la logica applicativa, e quindi il programma vero e proprio, è definito nel codice `addestra_lettore.c` che segue a breve. Il compito di `addestra_lettore.c` è di caricare una ad una le immagini contenute nel file di addestramento, leggendo sia la label che il valore dei 784 (28x28) pixel che compongono l'immagine, eseguire il *feed forward* dell'input e la *back propagation* dell'errore.

Una volta eseguita questa operazione, per tutte le immagini contenute nel file di training, il compito del programma è quello di salvare (a volte si dice persistere) lo stato della rete nei due file

`layer1.w` e `layer2.w`, dove l'estensione `w` sta per *weight*.

Per chi volesse approfondire l'argomento, il codice appena scritto può essere compilato usando l'opzione `-c` la quale informa il compilatore che il codice in input non ha un'entrata di tipo `main` e quindi il compilatore deve limitarsi a produrre un file oggetto che poi potrà essere linkato ad un altro file oggetto che contiene il punto di accesso. Comunque, per semplicità, in questo esempio seguiremo un'altra strada, cioè compileremo tutti i sorgenti insieme al fine di ottenere un eseguibile.

Vediamo ora il programma principale.

Il programma accetta dei parametri di input in modo che si possano cambiare alcune condizioni di esecuzione senza doverlo sempre ricompilare. Questi sono:

1. il nome del file CSV contenente i dati delle immagini e la label
2. il numero di volete (epoch) che si desidera ripetere l'addestramento usando lo stesso dataset
3. il numero di immagini nel campione
4. il learning rate della rete
5. il valore massimo che possono assumere le assegnazioni iniziali dei valori delle sinapsi
6. il seed usato per la generazione dei numeri

I parametri di input sono dei valori che possono essere passati al programma quando questo viene lanciato, per esempio, attraverso la riga di comando.

Il linguaggio C fornisce un modo semplice per memorizzare i parametri di input all'interno di variabili del programma. Infatti la funzione `main`, oltre al prototipo visto fin dall'inizio:

```
int main();
```

che non prevede nessun argomento, prevede anche il prototipo

```
int main(int argc, char * argv[]);
```

Il parametro `argc` è valorizzato dal sistema operativo al valore del numero di parametri con cui è stato lanciato il programma. Se

non si passa nulla alla riga di comando, questo parametro vale uno, perché conta come parametro anche il nome dell'eseguibile che stiamo lanciando. Il valore dei parametri passati è invece memorizzato nell'array di stringhe argv. Il nome del programma è memorizzato nel primo elemento dell'array: argv[0], il primo parametro (aggiuntivo) è memorizzato nel secondo elemento dell'array argv[1] e via dicendo.

Per fare una prova, provate a scrivere un semplice programma come quello che segue, compilarlo e lanciarlo passando alcuni parametri alla linea di comando:

```
#include <stdio.h>

//testami.c
int main(int argc, char * argv[])
{
    for(int i=0;i<argc;i++)
        printf("Parametro %d = %s\n",i,argv[i]);
}
```

compilatelo assegnandogli il nome di testami o quale altro nome volete, ed eseguitelo passandogli alcuni parametri, per esempio:

~/Documents/nn> gcc -o testami testami.c

~/Documents/nn> chmod u+x testami

~/Documents/nn>./testami ciao miao 0 1 xx

Ora che abbiamo definito il file *libreria ss\_snn\_lib.c* e abbiamo chiarito come si acquisiscono i parametri dalla riga di comando, vediamo finalmente il codice completo per l'addestramento della rete.

### Listato 3.7 addestra\_lettore.c

```
#include <stdlib.h>
#include <stdio.h>
#include "ss_snn_lib.h"

/*
 *| addestra_lettore.c
 *| Francesco Sisini (c) 2019
 */
```

```

/* Strato (layer) 1*/
#define L1_ND 784
#define L1_NP 100

/* Strato (layer) 2*/
#define L2_ND 100
#define L2_NP 10

/* Velocità apprendimento*/
#define RATE 0.2

int main(int argc,char *argv[])
{
    /* Epoche e campioni*/
    int epoche=1;
    int campioni=1;

    if(argc<4)
    {
        printf
            ("uso: ss_addestra <file.csv>" 
             "<numero_epoche> <numero_immagini>" 
             "<learning_rate> <max_sinapsi> <rnd_seed>");
        exit(1);
    }
    if(sscanf(argv[2],"%i",&epoche)!=1)
    {
        printf("%s non e' un intero\n",argv[2]);
        exit(1);
    }
    if(sscanf(argv[3],"%i",&campioni)!=1)
    {
        printf("%s non e' un intero\n",argv[3]);
        exit(1);
    }

    /*Velocita' apprendimento*/
    double rate=RATE;

    if(argc>=5)
    {
        printf("ok|");
        if(sscanf(argv[4],"%lf",&rate)!=1)
        {
            printf("%s non e' un float\n",argv[4]);
            exit(1);
        }
    }
}

```

```

/*Estremo superiore del valore iniziale delle sinapsi (0 to ...)*/
double sinapsi=0.001;

if(argc>=6)
{
    if(sscanf(argv[5],"%lf",&sinapsi)!=1)
    {
        printf("%s non e' un float\n",argv[5]);
        exit(1);
    }
}

/* Seme (seed) di inizializzazione dei numeri pseudocasuali */
int seed=1;

if(argc>=7)
{
    if(sscanf(argv[6],"%i",&seed)!=1)
    {
        printf("%s non e' un int\n",argv[6]);
        exit(1);
    }
}

/* Immagine da file CSV */
int img[L1_ND];
/* Label da file CSV (desiderd output) */
int label;

/*Strato 1*/
double v_x0[L1_ND+1];/* input dei percetroni del layer 1*/
double v_t[(L1_ND+1)*L1_NP];/* NP vettori di peso dendritico*/
double v_Dt[(L1_ND+1)*L1_NP];/* Variazione v_t */
double v_s1[L1_NP]; /*NP valori input*/
double v_y1[L1_NP];/* NP output uno per percettrone*/

/*Strato 2*/
double v_x1[L2_ND+1];/* input dei percetroni del layer 2*/
double v_u[(L2_ND+1)*L2_NP];/* NP vettori di peso dendritico*/
double v_Du[(L2_ND+1)*L2_NP];/* Variazione v_u*/
double v_s2[L2_NP]; /*NP valori input*/
double v_y2[L2_NP];/* NP output uno per percettrone*/

/*Output desiderato*/
double v_d[L2_NP];/* NP output desiderato uno per percettrone*/

/*Carica dal file le configurazioni iniziali della rete*/
srand(seed);

```

```

/*2) bias+pesi strato 1*/
for(int i=0;i<(L1_ND+1)*L1_NP;i++)
    v_t[i]=sinapsi*(double)rand()/(double)RAND_MAX;

printf
(" \n Campioni: %d, epoche: %d, l-rate: %f\n", campioni, epoche, rate);

/*3) bias+pesi strato 2*/
for(int i=0;i<(L2_ND+1)*L2_NP;i++)
    v_u[i]=sinapsi*(double)rand()/(double)RAND_MAX;

for(int ii=0;ii<epoche;ii++)
{
    printf("\x1b[5;1HEpoca %d di %d\n",ii,epoche);

    FILE* stream = fopen(argv[1], "r");
    if(stream==0) exit(1);
    /*Carica i dati di training ed esegue il training*/
    for(int jj=0;jj<campioni;jj++)
    {
        if(jj%100==0)
            printf("\x1b[6;1HCampione %d di %d\n",jj,campioni);

        get_image(img,&label,stream);

        /* conversione immagine da int a double */
        v_x0[0]=1;
        for(int i=0;i<L1_ND;i++)
        {
            if(img[i]>90)v_x0[i+1]=1;///((double)img[i]/255.);
            else v_x0[i+1]=0;
            //v_x0[i+1]=(double)img[i];

        }
        /*4 output desiderato*/
        for(int i=0;i<L2_NP;i++)
        {
            if(i == label)
                v_d[i]=1;
            else
                v_d[i]=0;
        }

    }

    /** PROPAGAZIONE AVANTI **/
    /** Feed Forward: Input->L1->output to L2*/
    layer_feed_forward(v_s1,v_y1,v_t,v_x0,L1_NP,L1_ND);

    /** Mappa y1 in x1 aggiungendo l'elemento x1_0=1*/
    layer_map_out_in(v_x1, v_y1,L2_ND);
}

```

```

/** Feed Forward: L1->L2->output*/
layer_feed_forward(v_s2,v_y2,v_u,v_x1,L2_NP,L2_ND);

/* CORREZIONE PER PROPAGAZIONE INVERSA */

/** Propagazione inversa dell'errore in L2 (v_u <- v_y2) */
for(int i=0;i<L2_NP;i++)
{
    /* correzione dei pesi (v_u) del percettore i-esimo */
    perc_correzione
    ( v_u+i*(L2_ND+1),v_x1,v_s2[i],v_d[i]-v_y2[i], rate,L2_ND);
}

/** Propagazione inversa dell'errore in L1 (v_t <- v_y2)*/
for(int i=0;i<L1_NP;i++)
{
    double dd=0;
    for(int j=0;j<L2_NP;j++)
    {
        /* w: peso del i-esimo dendrite del j-esimo
           percettore dello strato più esterno */
        double w=v_u[j*(L2_ND+1)+i];
        /* correzione */
        dd=dd+w*(v_d[j]-v_y2[j])*Dactiv_function(v_s2[j]);
    }

    /* correzione del percettore i-esimo*/
    perc_correzione
    ( v_t+i*(L1_ND+1),v_x0,v_s1[i],dd, rate,L1_ND);
}
fclose(stream);
}

/*
 *| Salva la rete
 */
FILE* sw = fopen("layer1.w", "w");
layer_writedown(sw,v_t, L1_ND, L1_NP);
fclose(sw);

sw = fopen("layer2.w", "w");
layer_writedown(sw,v_u, L2_ND, L2_NP);
fclose(sw);
}

```

```
Compilazione addestra_lettore.c  
~/Documents/nn> gcc -o ss_addestra  
addestra_lettore.c ss_snn_lib.c -lm  
~/Documents/nn> chmod u+x ss_addestra  
~/Documents/nn>./ss_addestra mnist_train.csv 7 60000  
0.2 0.01 9 100
```

Per quel che riguarda la compilazione, bisogna notare che questa volta, passiamo al gcc due file sorgenti e non uno solo. In questo esempio abbiamo dato per scontato che i due sorgenti e i file header siano nella stessa directory. Se così non fosse, il gcc prevede delle opzioni per configurare dette situazioni, e invito il lettore che vuole inespirtirsi a studiarle da sé.

## Il codice C di test

L'esecuzione di `ss_addestra` produce la scrittura dei due file `layer1.w` e `layer2.w`, e questo è l'unico risultato tangibile dell'esecuzione, perché poi il programma si arresta e perde il proprio stato di *addestrato* senza aver eseguito alcun riconoscimento di immagini.

Si noti la differenza con i codici visti prima, in cui il programma dopo l'addestramento, continuava il suo *lavoro* nel nuovo stato di *addestrato*, mentre in questo caso, l'azione di addestramento termina insieme al processo del programma.

Per sfruttare il lavoro di addestramento, scriviamo un altro programma, che sfrutta sia la libreria `ss_snn_lib.c` che i due file appena prodotti. Lo scopo di questo programma è di testare la rete e di calcolare il numero di immagini riconosciute rispetto al totale delle immagini usate per il test.

### Listato 3.8 `benchmark_lettore.c`

```
#include <stdlib.h>  
#include <stdio.h>
```

```

#include "ss_snn_lib.h"

/*
*| benchmark_lettore.c
*| Francesco Sisini (c) 2019
*/

/* Strato (layer) 1*/
#define L1_ND 784
#define L1_NP 100

/* Strato (layer) 2*/
#define L2_ND 100
#define L2_NP 10

#define N_TEST 10000

int main(int argc,char* argv[])
{
    if(argc<5)
    {
        printf("uso: ss_lettore <digit.csv> <layer1.w> <layer2.w>");
        exit(1);
    }

    /* Immagine da file CSV */
    int img[L1_ND];
    int label;
    int score=0;

    /*Strato 1*/
    double v_x0[L1_ND+1];/* input dei percettroni del layer 1*/
    double v_t[(L1_ND+1)*L1_NP];/* NP vettori di peso denditrico*/
    double v_s1[L1_NP]; /*NP valori input*/
    double v_y1[L1_NP];/* NP output uno per percettore*/

    /*Strato 2*/
    double v_x1[L2_ND+1];/* input dei percettroni del layer 2*/
    double v_u[(L2_ND+1)*L2_NP];/* NP vettori di peso denditrico*/
    double v_s2[L2_NP]; /*NP valori input*/
    double v_y2[L2_NP];/* NP output uno per percettore*/

    /* carica i pesi v_t e v_u dai file */
    FILE* w=fopen(argv[2],"rb");
    if(w==0)
    {
        printf("File %s non trovato\n",argv[2]);
        exit (1);
    }
}

```

```

}

layer_read(w,v_t, L1_ND, L1_NP);
fclose(w);

w=fopen(argv[3],"rb");
if(w==0)
{
    printf("File %s non trovato\n",argv[3]);
    exit (1);
}

layer_read(w,v_u, L2_ND, L2_NP);
fclose(w);

/* carica inout dai file */
w=fopen(argv[1],"r");
if(w==0)
{
    printf("File %s non trovato\n",argv[1]);
    exit (1);
}
for(int ii=0;ii<N_TEST;ii++)
{
    get_image(img,&label,w);
    /* conversione immagine da int a double */
    v_x0[0]=1;
    for(int i=0;i<L1_ND;i++)
    {
        if(img[i]>90)v_x0[i+1]=1;//((double)img[i]/255.);
        else v_x0[i+1]=0;
    }

    /*** PROPAGAZIONE AVANTI ***/
    /* Feed Forward: Input->L1->output to L2*/
    layer_feed_forward(v_s1,v_y1,v_t,v_x0,L1_NP,L1_ND);

    /* Mappa y1 in x1 aggiungendo l'elemento x1_0=1*/
    layer_map_out_in(v_x1, v_y1,L2_ND);

    /* Feed Forward: L1->L2->output*/
    layer_feed_forward(v_s2,v_y2,v_u,v_x1,L2_NP,L2_ND);

    /* Valutazione output */
    int imax;
    double fmax=0;

    for(int i=0; i<L2_NP;i++)
    {

```

```

        if(v_y2[i]>fmax)
        {
            fmax=v_y2[i];
            imax=i;
        }
    }
    if(label==imax) score++;
    print_object(v_x0+1,28, 28,10,30);

    fflush(stdout);
}
printf("\nScore %d/%d\n",score,N_TEST);
fclose(w);
}

```

### **Compilazione** benchmark\_lettore.c

```

~/Documents/nn> gcc -o ss_bm benchmark_lettore.c
ss_snn_lib.c -lm
~/Documents/nn> chmod u+x ss_benchmark
~/Documents/nn>./ss_addestra mnist_test.csv layer1.w
layer2.w

```

L'esecuzione del programma mostra, in rapida successione, le immagini *raw* lette dal file di addestramento. Al termine stampa sullo schermo il rapporto tra le immagini correttamente riconosciute e quelle non riconosciute.

Una domanda: in che modo sono legati i parametri che abbiamo passato al programma di addestramento, al risultato appena ottenuto? Provate a cambiarli, magari provate a cambiare anche il numero di neuroni hidden, e vedrete quanto sia difficile configurare una rete in modo da avere un risultato corretto.

## Considerazioni

L'esempio appena riportato è senz'altro più intrigante rispetto all'esempio del display a sette segmenti, ma a volte, quello che sembra non corrisponde completamente alla realtà. Configurando in modo opportuno la rete appena sviluppata è possibile

raggiungere una percentuale di errore attorno al 3%. Bene, questo è un ottimo risultato, ma è ancora più incredibile pensare che un risultato simile, cioè un errore dell'8% si può raggiungere con un percepitrone ad un solo strato.

Se si pone l'attenzione sul problema da un punto di vista matematico, ci si accorge subito della grande differenza tra i due problemi presentati, cioè il display a sette segmenti e il riconoscimento di cifre manoscritte.

Nel primo caso si hanno sette canali di input e dieci di output. Nel secondo caso, si hanno 784 canali di input e sempre 10 di output, è ovvio, che possano esistere, per la maggior parte delle cifre manoscritte, degli iperpiani che possono separare linearmente.

Questa piccola considerazione è solo per ricordarsi che il primo strumento che si deve affinare usando l'AI è la propria intelligenza, e non solo la potenza di calcolo o gli algoritmi alla moda.

### 3.9 Rete neurale a tre strati

#### Modellazione

Negli esempi visti fin'ora abbiamo sviluppato prima una rete ad un solo strato computazionale, cioè il percettrone a singolo strato, e poi una rete a due strati di percettroni.

Come abbiamo visto un singolo strato è sempre sufficiente quando i dati sono separabili linearmente, mentre è necessario implementare due strati quando i dati non hanno questa proprietà, per esempio se si vogliono separare i punti che stanno al di sotto del grafico di una parabola da quelli che stanno al di sopra, come illustrato nella figra 3.12.

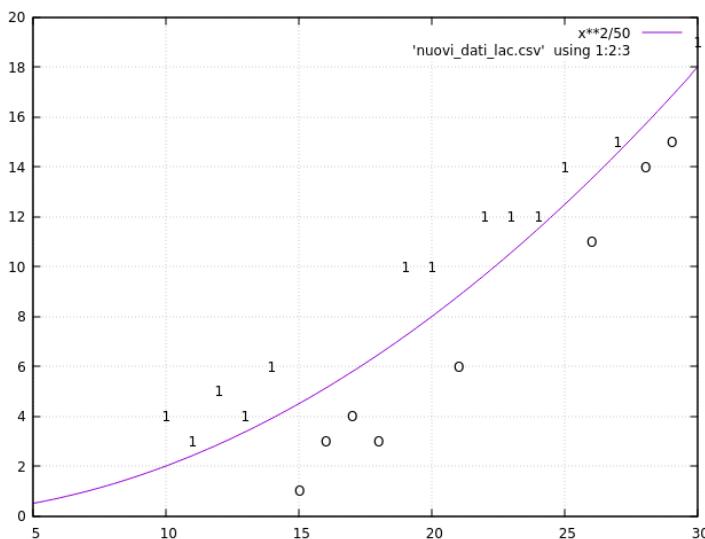
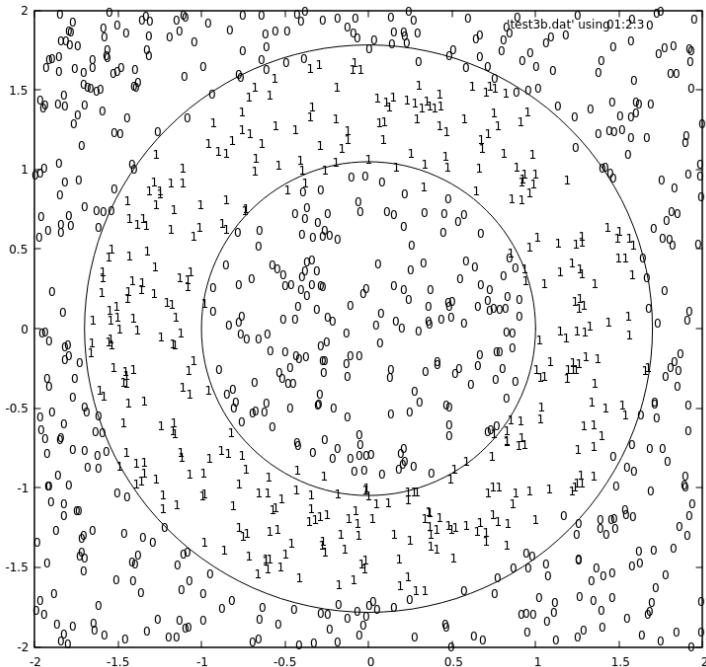


Fig. 3.12 - I punti sopra la parabola rappresentano i risultati delle prove scritte degli studenti ammessi ad un corso, mentre quelli sotto rappresentano gli studenti non iscritti. L'esempio completo è discusso su [pumar.it](http://pumar.it) nella sezione delle reti neurali.

A questo punto è naturale domandarsi se due strati di percettroni siano sufficienti per separare qualsiasi geometria in cui possano

essere divisi i dati. La risposta è no. Infatti due strati sono sufficienti solo per classificare dei dati la cui geometria sia descrivibile da regioni convesse, che in questo contesto, per semplificare, penseremo come tutte le forme geometriche che siano prive di buchi o di concavità.

Per esempio i dati di classe 1, indicati in figura 3.13, giacciono in una corona circolare e non possono essere classificati in modo efficiente usando una rete a due soli strati.



*Fig. 3.13 - Classificazione ,con una rete neurale a tre strati, in 1 e 0 dei punti che appartengono o meno alla corona circolare definita dai raggi 1 e 1.7.*

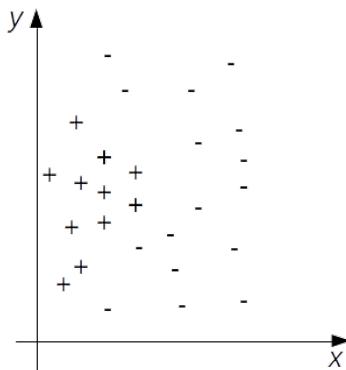
Il motivo può essere compreso anche attraverso dei semplici ragionamenti intuitivi.

Nel paragrafo 3.5 abbiamo visto che un singolo percettrone è sufficiente a classificare in due classi distinte dei dati che siano linearmente separabili.

Per semplificare i ragionamenti continuiamo a pensare di dover classificare dei dati che sono identificabili con dei punti del piano,

in questo caso linearmente separabili significa separabili da una retta (vedi fig. 3.5 bis).

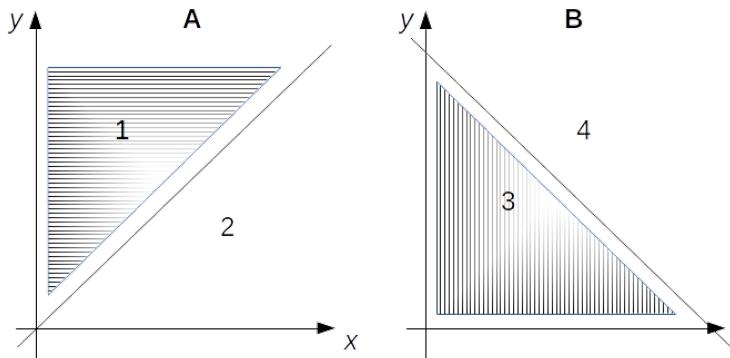
Supponiamo di dover addestrare una rete a distinguere le due classi di dati indicate in figura 3.14 una con i trattini e l'altra con le crocette.



*Fig. 3.14 - Dati appartenenti a due regioni non linearmente separabili.*

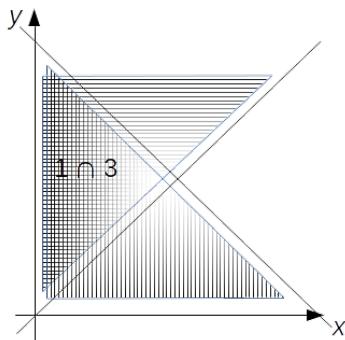
Abbiamo visto che un singolo percepitrone non può separare correttamente questi dati in due classi perché le regioni in cui giacciono trattini e crocette non sono linearmente separabili, cioè non esiste nessuna (singola) retta che divida il piano in modo che ogni semipiano conteggi solo una delle due classi di dati.

Per capire come una rete a due strati possa invece classificare i dati rappresentati in fig. 3.14, progrediamo per gradi e costruiamo le quattro regioni (1,2,3 e 4) rappresentate in fig. 3.15 **A** e **B**. Dalla figura si vede abbastanza facilmente che la regione contenente le crocette (in fig. 3.14) è data dall'intersezione delle due regioni 1 e 3. L'intersezione è rappresentata in fig. 3.16



*Fig. 3.15 - La retta di equazione  $y = x$  divide il piano nelle regioni 1 e 2 (A). La retta di equazione  $y = 1-x$  divide il piano nelle regioni 3 e 4 (B)*

Entrambe le regioni 1 e 3 sono linearmente separabili, quindi sia i punti appartenenti alla regione 1 che quelli appartenenti alla 3 possono essere classificati per mezzo di un percettrone ad un singolo strato, il primo di equazione  $y = x$  ed il secondo di equazione  $y = 1-x$ .



*Fig. 3.16 - Intersezione delle regioni 1 e 3.*

Possiamo pensare di definire un percettrone per la regione 1 e uno per la regione 3, costruendo quindi una rete con due neuroni di ingresso ( $x_1, x_2$ ) e due neuroni di uscita ( $y_1, y_2$ ), come rappresentato in figura 3.17

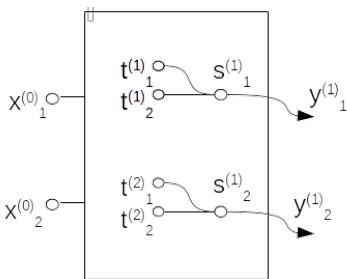


Fig. 3.17 - Rete composta da un singolo strato di due percettroni.

Per classificare correttamente le crocette dobbiamo scegliere solo i punti che sono stati riconosciuti da entrambi i percettroni, in pratica l'insieme dei punti che soddisfa il sistema di disequazioni  $y > x$  e  $y < 1-x$ . Potremmo quindi classificare un dato come appartenente alla regione 1n3 se entrambe le uscite  $y^{(1)}_1$  e  $y^{(1)}_2$  sono attive simultaneamente. Continuando ad usare le funzioni definite precedentemente per il calcolo dell'attivazione dei percettroni, potremmo scrivere il codice per classificare i dati come segue:

```

...
y1 = activ_function(s1);
y2 = activ_function(s2);
if( y1>0.5 && y2>0.5)
{
// CLASSE +
}
else
{
// CLASSE -
}

```

In sostanza abbiamo aggiunto uno strato di logica *if-else* alla classificazione eseguita dalla rete neurale.

Questa soluzione può sembrare ragionevole, ma richiede che si conosca a priori la geometria delle regioni da classificare, come

nell'esempio in questione.

Normalmente le reti neurali vengono impiegate in problemi di cui non è nota a priori la suddivisione geometrica delle regioni corrispondenti alle classi, un esempio è quello già visto della digit recognition. Inoltre, se la geometria delle regioni è nota a priori, l'uso di una rete neurale non è necessario, perché è possibile costruire un algoritmo task specific.

L'aggiunta di un secondo strato di percetroni alla rete neurale sostituisce lo strato if-else visto sopra, con la differenza che la clausola logica per classificare i dati non deve essere nota al momento della compilazione, ma si *auto definisce* a runtime, attraverso il processo di addestramento visto nei paragrafi precedenti.

Il modo di classificare i dati anche quando le regioni non sono linearmente separabili si basa sul ragionamento analitico appena condotto. Infatti, se un dato appartiene alla regione 1 $\cap$ 3, allora dovranno essere attivi entrambi gli output del percettrone. A questo punto, per classificare il dato, è sufficiente indirizzare gli output  $y^{(1)}_1, y^{(1)}_2$  ai neuroni di input di un terzo percettrone, come mostrato in figura 3.18.

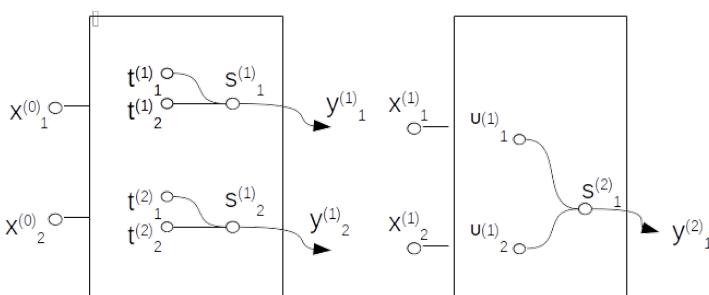


Fig. 3.18 - Rete composta da due strati di percetroni.

La rete verrà addestrata in modo che il neurone di output  $y^{(2)}_1$  si attivi solo quando sono simultaneamente attivi i due neuroni di ingresso  $x^{(1)}_1, x^{(1)}_2$ , a loro volta connessi ai neuroni di uscita  $y^{(1)}_1, y^{(1)}_2$ .

## **Neuroni o percetroni?**

A volte potrebbe nascere un po' di confusione riguardo alla differenza tra neuroni e percetroni, proviamo a capire in che modo questi termini vengono utilizzati e perché si può cadere in confusione.

Vediamo anzitutto che i neuroni sono componenti biologici e non sono presenti negli attuali microprocessori (anche se esistono esperimenti e tecnologie che iniziano a farne uso), quindi quando parliamo di neurone lo facciamo sempre in modo figurato. Il percettrone invece è una unità di calcolo che ha un certo numero di variabili di input e una sola variabile di output. Dal confronto con la biologia, possiamo pensare che gli ingressi (input) del percettrone siano dei neuroni sensitivi, mentre le uscite siano dei neuroni attuatori, o motori o efferenti.

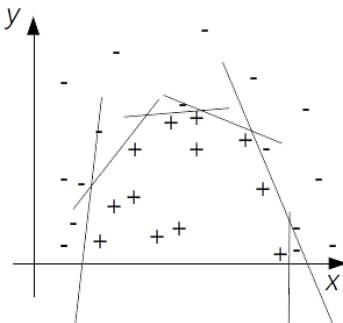
Quando un primo strato di percetroni è collegato ad un secondo strato, i neuroni sensitivi del secondo strato possono anche essere pensati come dei dendriti, come spiegato nel paragrafo 3.5.

Quindi questi termini devono essere intesi ed usati con elasticità in base al contesto, ricordando che si tratta solo di rappresentazioni matematiche di entità biologiche.

L'analisi che abbiamo condotto dovrebbe chiarire anche uno dei principali interrogativi che attanaglia chi si trova a dover configurare ed addestrare una rete neurale: *quanti neuroni servono?*

Il numero di neuroni di ingresso è stabilito dal problema stesso. Se si devono classificare dati che stanno sul piano cartesiano allora servono due neuroni di ingresso, se si devono classificare dati acquisiti da una matrice di 786 pixel, come il caso della digit recognition, allora servono settecento ottantasei neuroni. Anche il numero dei neuroni di output è definito in base al numero delle classi in cui si vogliono classificare i dati di ingresso. Quindi: sia i neuroni di input che quelli di output sono stabiliti dalla natura stessa del problema. Il numero dei neuroni intercalari, cioè dei

percetroni che si trovano in uno strato interno, deve invece essere scelto in base alla complessità della geometria che descrive le regioni di appartenenza delle classi. In altre parole dipende dalla forma delle regioni, come si può vedere nella figura seguente:



*Fig. 3.19 - La regione dei dati indicati dalle crocette è definita attraverso le intersezioni di sei rette nel piano.*

Come si vede nella figura 3.19, i dati possono essere racchiusi in due regioni separabili attraverso sei rette, quindi in questo caso sono necessari sei percetroni dello strato intercalare, cioè il secondo strato di percetroni.

Come abbiamo anticipato all'inizio del capitolo, due soli strati di percetroni non sono **sempre** sufficienti. Vediamo il caso delle due regioni di dati evidenziate nella figura 3.20. Come si vede le due classi sono separabili per mezzo di quattro rette, ma la regione identificata dalle loro intersezioni non è una regione convessa in quanto presenta una concavità.

Le coordinate dei dati della classe *crocetta* devono soddisfare uno o l'altro dei seguenti due sistemi:

$$\begin{cases} y > x + 5 \\ y < 6 - x \end{cases}$$

$$\begin{cases} y > x + 10 \\ y < 11 - x \end{cases}$$

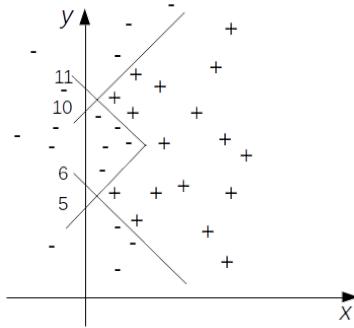


Fig. 3.20 - La regione dei dati indicati dalle crocette è definita attraverso le intersezioni di quattro rette nel piano. La regione presenta una concavità.

Coma abbiamo visto, ognuno dei due sistemi di equazioni può essere implementato mediante una rete neurale a due strati, ma per eseguire l'OR logico tra le uscite del secondo strato è necessario inserire un terzo strato, come riportato nella figura seguente:

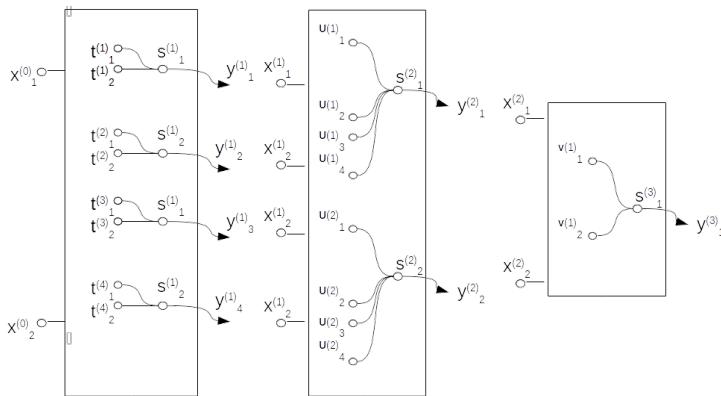


Fig. 3.21 - rete neurale a tre strati di percetroni.

Tre strati di percetroni sono sufficienti per rappresentare qualsiasi regione spaziale in più dimensioni. Infatti, il percettrone modella correttamente sia la funzione logica AND che l'OR. Il numero di neuroni della rete deve essere ovviamente adeguato alla complessità della geometria che definisce le regioni delle classi dei dati.

## **Perché esistono reti con più di tre strati?**

Ora che si è chiarito che tre strati di percetroni sono sufficienti a classificare i dati appartenenti a regioni con geometrie arbitrarie, ci si può chiedere perché nel *machine learning* si usino reti *deep* con molti più strati di tre.

Le reti *deep* nascono dall'idea di non connettere completamente gli strati dei percetroni, definendo quella che Fukushima chiamò l'area di connessione, e che è alla base del concetto di rete neurale convoluzionale, poi evoluta in *deep neural network*. In pratica, le connessioni neurali che vengono perse tra gli strati adiacenti, si recuperano in profondità, aumentando il numero dei layer. Questo porta alla semplificazione di diversi problemi e cul de sac legati alla computazione concreta delle reti neurali *full connected*, cioè quelle in cui l'uscita di un percettrone da uno strato, è portata all'entrata di tutti i percetroni dello strato successivo.

Per comprendere le basi del *deep learning* si consiglia di studiare prima il modello matematico del *cognitrone* che è il modello matematico sviluppato da Fukushima per spiegare l'apprendimento senza supervisione da parte dei vertebrati superiori. L'introduzione a questo argomento è fornita al termine di questo libro e per il quale ho scritto una monografia dedicata.

## **Addestramento**

La logica e la matematica che descrivono il terzo strato sono analoghe a quelle già viste nel capitolo 3.6. Fin'ora abbiamo usato i simboli  $t^{(i)}$  e  $u^{(i)}$  per indicare i vettori dei pesi sinaptici dei percetroni presenti nel primo e nel secondo strato computazionale, per continuità usiamo il simbolo  $v^{(i)}$  per indicare i pesi sinaptici dell' $i$ -esimo percettrone del terzo strato (vedi figura 3.21).

L'addestramento di una rete neurale a tre strati può essere

condotto similmente a quanto già visto per la rete a due strati usando la *backpropagation*.

L'output prodotto dal terzo strato, per esempio il valore  $y^{(3)}_1$  viene confrontato con il valore atteso (cioè la label, anche detta target value) e si usa la loro differenza per calcolare la variazione dei pesi  $\mathbf{v}^{(i)}$  come detritto dalle seguenti equazioni:

$$\Delta \mathbf{v}^{(1)} = \begin{bmatrix} 1 \times (d - y_i^{(3)}) f'(s_1^{(3)}) \\ y_1^{(2)} (d - y_1^{(3)}) f'(s_1^{(3)}) \\ y_2^{(2)} (d - y_1^{(3)}) f'(s_1^{(3)}) \end{bmatrix}$$

dove  $d$  rappresenta il risultato atteso.

Per gli starti **t** e **u** valgono invece le equazioni (3.11) già descritte in precedenza.

### 3.10 La libreria ReLe

Nel capitolo precedente abbiamo visto come modularizzare un progetto separando il codice in file distinti: uno per il modulo principale, quello che contiene la funzione `main` e che possiamo chiamare il modulo utente, altri per i moduli di libreria, cioè quelli contenenti le implementazioni delle funzioni in cui il codice è stato strutturato.

La modularizzazione permette a chi sta programmando di avere una visione più chiara ed ordinata del proprio progetto, inoltre permette anche il riuso semplice del codice.

Indipendentemente dal motivo per cui si intende usare una rete neurale, per esempio per risolvere un problema di classificazione di immagini o per interpretare dei valori clinici, ci sono certe operazioni che risultano comuni, come il *feed forward*, per questo inserirle in un modulo specifico che possa essere compilato e linkato quando necessario è un'opzione molto comoda.

La modularizzazione realizzata fin'ora però non è ancora sufficiente per poter considerare il file `ss_snn_lib.c` una libreria a tutti gli effetti. A ben guardarsi, il codice presente nella funzione `main` del listato `addestra_lettore.c` è ancora molto elaborato, e questo comporta che si perda facilmente il quadro generale.

Una vera libreria, deve liberare il programmaore dai dettagli implementativi del problema che astrae (in questo caso la computazione neurale), permettendogli di concentrarsi sui suoi obiettivi di sviluppo.

In questo capitolo vediamo proprio una libreria per l'utilizzo di reti neurali che ho sviluppato specificatamente per questo testo, inserendo solo le caratteristiche fondamentali che deve avere una rete per essere usata concretamente in un progetto sofware e che sono state qui trattate.

Ho cercato di mantenere l'interfaccia della rete, cioè l'insieme

delle funzioni che vengono chiamate direttamente dal programmatore, abbastanza simile all'interfaccia di altre librerie professionali, come ad esempio la libreria Fast Artificial Neural Network (FANN) che è stata una delle principali librerie per l'addestramento delle reti neurali in C degli ultimi dieci anni. Una volta presa confidenza con la logica e l'implementazione della libreria ReLe, qui presentata, sarà molto più semplice passare a librerie professionali.

## L'interfaccia

La libreria ReLe introduce il tipo di dato `rele_rete` che è una struttura i cui campi servono a memorizzare tutti i parametri della rete neurale.

Le operazioni base per lavorare su una rete neurale sono

- dichiarare una variabile di tipo `rele_rete`
- creare la rete usando la funzione `rele_Crea_rete`
- addestrare la rete con la funzione `rele_Addestra`
- salvare la rete su file con la funzione `rele_Salva`
- aprire una rete da file con la funzione `rele_Apri`
- classificare un dato usando la funzione `rele_Classifica`

Usando questo insieme di funzioni è possibile scrivere qualsiasi programma che si basi sull'utilizzo di una rete neurale.

## Compilare e linkare la libreria

Il codice completo della libreria è mostrato nei listati 3.13 e 3.14 in fondo al capitolo. Si tratta però di un codice assai lungo, per cui, se si preferisce, la libreria può anche essere scaricata dalla risorsa indicata alla fine del libro. Al fine di mantere il libro coerente e autoconsistente, anche di fronte alla normale eventualità che negli anni alcune risorse web vengano spostate o eliminate, il codice è comunque riportato qui per intero. Per

continuare la lettura con gli esempi proposti è necessario editare e compilare la libreria dai listati 3.13 e 3.14, oppure scaricarla.

Se si sceglie di scaricare la libreria da GitHub (al momento il progetto è ospitato lì), si seguano le istruzioni riportate nel documento Readme. Se invece si decide di editare il codice dal presente testo, si seguano le operazioni descritte qui di seguito:

1. Editare `librele.h`
2. Editare `librele.c` (salvarli nella stessa directroy)
3. Compilare la libreria `gcc -c librele.c -lm`

Se la compilazione avviene senza errori viene prodotto il file `librele.o` con il codice oggetto della libreria. Tale file verrà poi linkato a tutti gli esempi che vederemo ora nel seguito.

L'istruzione di compilazione è sempre relativa al sistema operativo Linux, ma la libreria è indipendente dal sistema operativo e può essere compilata su qualsiasi piattaforma per cui si abbia un compilatore C.

## **Creare una rete, salvarla e riaprirla**

Con questo esempio vediamo le operazioni di base per iniziare ad usare la libreria. Nel listato che segue dichiariamo un puntatore ad una variabile di tipo `rele_rete` e lo inizializziamo con la funzione `rele_Crea_rete`. Alla funzione passiamo come argomento il valore 2 che rappresenta il numero dei neuroni di input, e 1 quello di neuroni di output. I successivi due parametri rappresentano il numero di neuroni del primo e del secondo strato intercalare, e in questo esempio vengono lasciati a 0, quindi si tratta di una rete ad un singolo strato di percetteroni.

Dopo aver creato la rete la salviamo su un file usando la funzione `rele_Salva`. Dopo questa chiamata il lettore è invitato ad aprire il file con un editor di testo o usando un foglio di calcolo tipo Calc o

Excel.

### **Listato 3.9 crea\_salva\_apri.c**

```
#include "librele.h"
#include <stdio.h>
#include <stdlib.h>

int main()
{
    rete_rete * r =  rete_Crea_rete(2,1,0,0);
    rete_parametri par;

    FILE * f = fopen("rete.csv","wt");
    rete_Salva(r,f);
    fclose(f);
    rete_Libera_rete(r);

    f= fopen("rete.csv","rt");
    r = rete_Apri(f);

    rete_Libera_rete(r);
}
```

```
Compilazione crea_salva_apri
~/Documents/nn> gcc -o crea_salva_apri
crea_salva_apri.c librele.o -lm
~/Documents/nn> chmod u+x crea_salva_apri
~/Documents/nn>./crea_salva_apri
```

Dopo aver compilato il programma, lanciatelo come descritto qui sopra. Se non ci sono errori, nella cartella troverete il file rete.csv che aprendolo con un editor di testo appare circa come segue, tenendo conto che il valore iniziale dei pesi dello strato **t** è stabilito in modo random:

```

#Distribuzione dei neuroni sulla rete
      2      1      0      0      1
#Neuroni e dendriti
      1      2      0      0      0      0
#Qualità rete
      0.000000      0      0.000000
#Strato t
      0.268038      0.178877      0.256620
#Strato u
#Strato v

```

Nella prima riga sono riportati rispettivamente:

1. il numero di neuroni di ingresso (input)
2. il numero di neuroni di uscita (output)
3. il numero di neuroni del primo strato interno
4. il numero di neuroni del secondo strato interno
5. il numero totale di strati di percetroni, cioè di strati computazionali

I numeri presenti nella seconda riga riportano il numero di percetroni e il numero di dendriti per percettrone rispettivamente per ogni strato computazionale.

1. il numero di percetroni nel primo strato
2. il numero di dendriti per ogni percettrone
3. il numero di percetroni nel secondo strato
4. il numero di dendriti per ogni percettrone
5. il numero di percetroni nel terzo strato
6. il numero di dendriti per ogni percettrone

Nella terza riga sono riportati i parametri che servono per valutare

la qualità della risposta della rete:

1. l'errore quadratico dell'ultima operazione di addestramento
2. il numero di operazioni di addestramento totale eseguite
3. l'errore quadratico medio calcolato in base al numero di operazioni di addestramento

Le ultime righe riportano i pesi delle connessioni sinaptiche (o dendritiche) degli strati **t**, **u** e **v** rispettivamente.

1. il primo valore rappresenta il bias del percettrone a cui seguono  $n$  valori quanti sono i dendriti del percettrone. Sono presenti tante righe quanti sono i percetroni dello strato.

## Addestramento di una rete ad un solo strato

Vediamo ora come usare la libreria ReLe per addestrare una rete neurale a riconoscere dei dati che siano linearmente separati. Consideriamo l'esempio semplice in cui tutti i punti che giacciono sopra la retta bisettrice del piano siano di classe 0 e quelli sotto di classe 1.

Genereremo i dati per l'addestramento usando una funzione generatrice anziché dei dati sperimentali, lo stesso faremo per testare l'addestramento della rete.

Trattandosi di punti nel piano, la rete sarà costituita da due neuroni di ingresso cioè le due coordinate del punto nel piano che mapperemo negli ingressi dendritici  $x_1$  e  $x_2$  e un neurone di uscita, ovvero la sua classe il cui valore è presentato nella variabile  $y$  (vedi fig. 3.22).

Come criterio di arresto per la fase di addestramento usiamo l'errore quadratico medio del valore di uscita  $y$  rispetto alla classe definita dalla funzione generatrice.

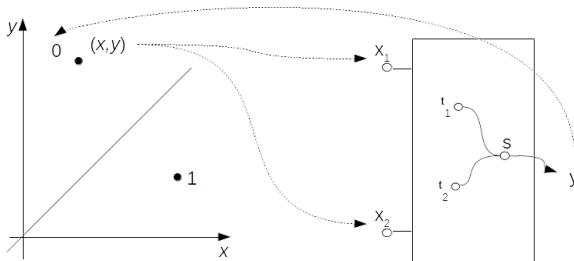


Fig. 3.22 - Rete neurale ad un singolo strato. Le coordinate di un punto nel piano vengono inserite come input nella rete neurale.

### Listato 3.10 addestra\_uno\_strato.c

```
#include "librele.h"
#include <stdio.h>
#include <stdlib.h>

#define EQM_ACCETTABILE 0.01
#define ITERAZIONI 100.

int main()
{
    rele_rete * r = rele_Crea_rete(2,1,0,0);
    rele_parametri par;
    par.fattore_apprendimento = 0.05;

    double d[2];
    double c[1];

    int iterazioni = 0;
    double eqm;
    do
    {
        eqm = 0;
        for(int i=0;i<ITERAZIONI;i++)
        {
            iterazioni++;
            /* Genera le coordinate di un punto nel piano */
            d[0]=(double)rand()/(double)RAND_MAX;
            d[1]=(double)rand()/(double)RAND_MAX;
            c[0]=1;
            /*
            Funzione generatrice:
            il punto è classificato come 0 se giace sopra la bisettrice */
            if(d[0]<d[1]) c[0]=0;
            r = rele_Addestra(r,&par,d,c);
        }
    } while(eqm > EQM_ACCETTABILE);
}
```

```

        eqm+=1./ITERAZIONI*r->EQ;
    }
}

/* Termina quando l'errore quadratico, mediato su 100 iterazioni è soddisfacente */
while(eqm>EQM_ACCETTABILE);

/* Testa la rete con dati creati dalla funzione generatrice */
printf("Terminato in %d iterazioni\n",iterazioni);
for(int i=0;i<1000;i++)
{
    d[0]=(double)rand()/(double)RAND_MAX;
    d[1]=(double)rand()/(double)RAND_MAX;
    rete_Classifica(r, d);
    printf("%lf %lf %d\n",d[0],d[1],r->strato_uscita[0]>0.5);
}
rete_Libera_rete(r);
}

```

### **Compilazione** addestra\_<uno>\_strato.c

```

~/Documents/nn> gcc -o addestra_<uno>_strato
addestra_<uno>_strato.c librele.o -lm
~/Documents/nn> chmod u+x addestra_<uno>_strato
~/Documents/nn>./addestra_<uno>_strato

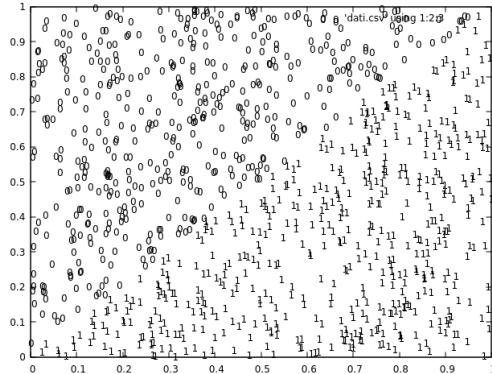
```

L'output deve produrre sul monitor un elenco delle coordinate di mille punti comprese tra zero ed uno, affiancate dalla classe riconosciuta dalla rete.

Per verificare la classificazione si può salvare l'output su un file come segue:

```
~/Documents/nn>./addestra_<uno>_strato > classi.csv
```

e poi aprirlo con Calc o Excel per plottarlo, come riportato nella figura seguente.



*Fig. 3.23 - Classificazione di dati generati dalla funzione generatrice del programma addestra\_uno\_strato.c.*

## Addestramento di una rete a due e a tre strati

Per addestrare una rete a due strati è necessario definire il numero dei percettori presenti nel secondo strato di computazione.

Di seguito vengono riportati gli esempi di codice per creare ed addestrare una rete a due e a tre strati.

### Listato 3.11 addestra\_due\_strati.c

```
#include "librele.h"
#include <stdio.h>
#include <stdlib.h>

#define EQM_ACCETTABILE 0.01
#define ITERAZIONI 100.
int main()
{
    rele_rete * r = rele_Crea_rete( 2,1,30,0);
    rele_parametri par;
    par.fattore_apprendimento = 0.05;

    double d[2];
    double c[1];

    int iterazioni = 0;
    double eqm;
    do
    {
```

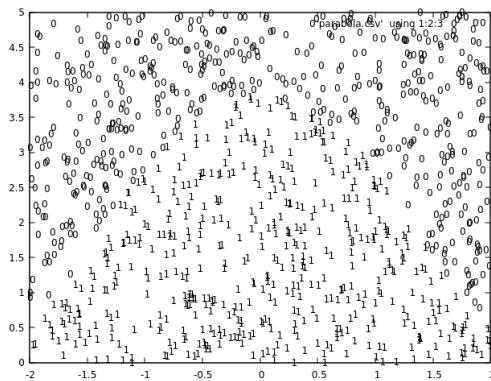
```

eqm = 0;
for(int i=0;i<ITERAZIONI;i++)
{
    iterazioni++;
    /* Genera le coordinate di un punto nel piano */
    d[0]=4.*(double)rand()/(double)RAND_MAX-2;
    d[1]=5*(double)rand()/(double)RAND_MAX;
    c[0]=1;
    /* il punto è classificato come 0 se giace sotto la parabola */
    if(-d[0]*d[0]+4 < d[1]) c[0]=0;
    /* addestra la rete passando le coordinate del punto e il target
       output, cioè la label della classe di appartenenza */
    r = rete_Addestra(r,&par,d,c);
    eqm+=1./ITERAZIONI*r->EQ;
}
}

/* Termina quando l'errore quadratico, mediato su 100 iterazioni è soddisfacente */
while(eqm>EQM_ACCETTABILE);

printf("Terminato in %d iterazioni\n",iterazioni);
for(int i=0;i<1000;i++)
{
    d[0]=4.*(double)rand()/(double)RAND_MAX-2;
    d[1]=5*(double)rand()/(double)RAND_MAX;
    rete_Classifica(r, d);
    printf("%lf %lf %d\n",d[0],d[1],r->strato_uscita[0]>0.5);
}
rete_Libera_rete(r);
}

```



*Fig. 3.24 - Classificazione di dati generati dalla funzione generatrice del programma addestra\_due\_strati.c.*

L'esempio che segue, oltre ad implementare una rete a tre strati

computazionali, prevede due percetroni di output anziché uno solo.

### Listato 3.12 addestra\_tre\_strati.c

```
#include "librele.h"
#include <stdio.h>
#include <stdlib.h>

#define EQM_ACCETTABILE 0.01
#define ITERAZIONI 100.

int main()
{
    rete_rete * r =  rele_Crea_rete(2,2,20,20);
    rete_parametri par;
    par.fattore_apprendimento = 0.3;

    double d[2];
    double c[2];

    int iterazioni = 0;
    double eqm;
    do
    {
        eqm = 0;
        for(int i=0;i<ITERAZIONI;i++)
        {
            iterazioni++;
            /* Genera le coordinate di un punto nel piano */
            d[0]=8.*((double)rand()/(double)RAND_MAX-4;
            d[1]=8.*((double)rand()/(double)RAND_MAX-4;
            c[0]=0;
            c[1]=0;
            /* il punto è classificato come 1 se appartiene alla corona circolare */
            if((d[0]*d[0] + d[1]*d[1] >1) && (d[0]*d[0] + d[1]*d[1] < 3.)) c[0] = 1 ;
            if((d[0]*d[0] + d[1]*d[1] >4) && (d[0]*d[0] + d[1]*d[1] < 5.)) c[1] = 1 ;
            /* addestra la rete passando le coordinate del punto e il target
               output, cioè la label della classe di appartenenza */
            r = rele_Addestra(r,&par,d,c);
            eqm+=1./ITERAZIONI*r->EQ;
        }
    }
    /* Termina quando l'errore quadratico, mediato su 100 iterazioni è soddisfacente */
    while(eqm>EQM_ACCETTABILE);

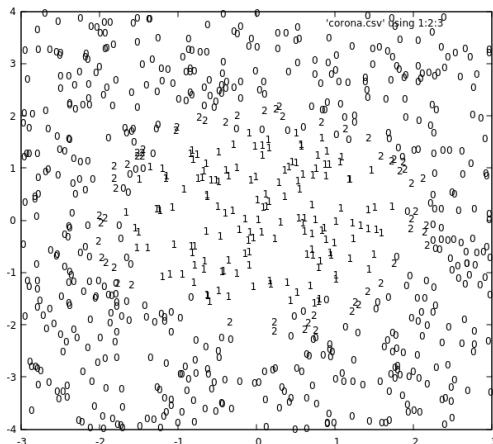
    printf("Terminato in %d iterazioni\n",iterazioni);
    for(int i=0;i<1000;i++)

```

```

{
    d[0] = 8.*(double)rand()/(double)RAND_MAX-4;
    d[1] = 8.*(double)rand()/(double)RAND_MAX-4;
    rele_Classifica(r, d);
    int cls = 0;
    if(r->strato_uscita[0]>r->strato_uscita[1] && r->strato_uscita[0]> 0.5)
        cls = 1;
    if(r->strato_uscita[1]>r->strato_uscita[0] && r->strato_uscita[1]> 0.5)
        cls = 2;
    printf("%lf %lf %d\n",d[0],d[1],cls);
}
rele_Libera_rete(r);
}

```



*Fig. 3.25 - Classificazione di dati generati dalla funzione generatrice del programma addestra\_tre\_strati.c.*

Per la compilazione e l'esecuzione si proceda come illustrato nell'esempio della rete ad un solo strato.

Con questi semplici esempi è stato mostrato l'uso di base della libreria ReLe, adesso ci sono tutti gli elementi per realizzare dei progetti propri basati sull'addestramento e sull'uso delle reti neurali.

## Il codice della libreria

### Listato 3.13 librele.h

```
#include <stdio.h>

typedef struct
{
    /** Interfacciamento */

    double * strato_ingresso;
    double * strato_uscita;

    /** Topologia rete */

    /* Distribuzione dei neuroni negli strati della rete */
    int N_neuroni_sensitivi;
    int N_neuroni_afferenti;
    int N_neuroni_primo_strato_intercalare;
    int N_neuroni_secondo_strato_intercalare;
    int N_strati_computazionali;

    /* neuroni e dendriti*/
    int l1_np, l1_nd;
    int l2_np, l2_nd;
    int l3_np, l3_nd;

    /** Qualità della rete */

    /* Errore quadratico */
    double EQ;

    /* Iterazioni di addestramento */
    int iterazioni;

    /* Errore quadratico medio = EQ/iterazioni*/
    double EQM;

    /** Rappresentazione in memoria */

    /* Risultato atteso, o target output */
    double * v_d;

    /* Strato 1: dall'ingresso 0 all'uscita 1 */
    double * v_x0;
    double * v_t;
    double * v_s1;
    double * v_y1;
```

```

/* Strato 2: dall'ingresso 1 all'uscita 2 */
double * v_x1;
double * v_u;
double * v_u_tmp;
double * v_s2;
double * v_y2;
double * v_dy2;

/* Strato 3: dall'ingresso 2 all'uscita 3 */
double * v_x2;
double * v_v;
double * v_v_tmp;
double * v_s3;
double * v_y3;
double * v_dy3;
} rele_rete;

typedef struct
{
    double fattore_apprendimento;
    double max_per_normalizzazione;
    int seme_pseudocasuale;
    int epoche;
    int campioni;
}rele_parametri;

/* crea una rete neurale che può essere addestrata.
   Inizializza i pesi dendritici (pesi delle connessioni)
   tra -0.1 e 0.1
*/
rele_rete * rele_Crea_rete(
    int N_neuroni_sensitivi,
    int N_neuroni_fferenti,
    int N_neuroni_primo_strato_intercalare,
    int N_neuroni_secondo_strato_intercalare);

/* libera le risorse della rete */
void rele.Libera_rete(rele_rete * rete);

/*
   addestra una rete neurale usando i dati e le classi passate
   ogni chiamata viene eseguita una singola iterazione
*/
rele_rete * rele_Addestra(rele_rete * rn,
                           rele_parametri * par,
                           double * dati,
                           double * classi);
/*

```

```

    Usa una rete già addestrata per classificare i dati
    il risultato è puntato da rete->strato_uscita
*/
void rele_Classifica(rele_rete * rete, double * dati);

void rele_Salva(rele_rete * rete, FILE * f);

rele_rete * rele_Apri(FILE * f);

/*
*|
*| v_s: vettore delle somme dei canali dendritici per gli n_perc percetroni
*| v_y: vettore degli output per gli n_perc percetroni
*| v_w: vettore dei pesi dendritici per gli n_perc percetroni
*| v_x: vettore degli input al percettrone (uguale per tutti gli n_perc percetroni)
*| n_perc: numero di percetroni nello strato
*| n_dend: numero di dendriti per percettrone */
void layer_feed_forward
(double v_s[],double v_y[],double v_w[],double v_x[],int n_perc, int n_dend);

/*
v_w vettore di n_dend+1. Il primo elemento è 1, il resto sono i pesi sinaptici
v_x vettore dell'input del percettrone. Il primo elemento è 1
z somma pesata dell'input: v_w < . v_x
d Valore Atteso - Valore Calcolato
rate learning rate
n_dend numero dei dendriti del percettrone
*/
void perc_correzione
(double v_w[],double v_x[],double z,double d,double rate,int n_dend);

/* Mappa l'output v_y in v_x aggiungendo prima l'elemento v_x[0]=1*/
void layer_map_out_in(double v_x[],double v_y[], int n_dend);

/*
v_w: vettore di dimensione n_dend+1 di pesi dendritici
v_x: vettore degli n_dend+1 (c'è il bias) input al percettrone
n_dend numero di dendriti
*/
double perc_calc_output(double v_w[],double v_x[],int n_dend);

/* Corregge i pesi del percettrone */
void perc_correzione
(double v_w[],double v_x[],double z,double d,double rate,int n_dend);

/* Calcola il valore della risposta del percettrone*/
double activ_function(double summed_input);

```

```

/* Calcola il valore della derivata della risposta del percettrone*/
double Dactiv_function(double summed_input);

/* Calcola e aggiorna l'EQM in base all'ultimo EQ e il numero di iterazioni eseguite */
void aggiorna_EQM(rele_rete * rn);

```

### Listato 3.14 librele.c

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include "librele.h"
#define MAX_PESO 0.1
#define MIN_PESO -0.1

/***
INTERFACCIA:
SEZIONI FUNZIONI PUBBLICHE
***/

rele_rete * rele_Apri(FILE * f)
{
    int N_neuroni_sensitivi;
    int N_neuroni_afferenti;
    int N_neuroni_primo_strato_intercalare;
    int N_neuroni_secondo_strato_intercalare;
    int N_strati_computazionali;

    char buff[1000];
    fscanf(f,"%[^n]s",buff);

    fscanf(f,"t%d\tn%d\tn%d\tn%d\tn",
           &N_neuroni_sensitivi,
           &N_neuroni_afferenti,
           &N_neuroni_primo_strato_intercalare,
           &N_neuroni_secondo_strato_intercalare,
           &N_strati_computazionali);

    rele_rete * rete = rele_Crea_rete(N_neuroni_sensitivi,
                                       N_neuroni_afferenti,
                                       N_neuroni_primo_strato_intercalare,
                                       N_neuroni_secondo_strato_intercalare);
}

```

```

fscanf(f,"%[\n]s",buff);
fscanf(f,"\\t%d\\t%d\\t%d\\t%d\\t%d\\t%d\\n",
       &rete->l1_np,&rete->l1_nd,&rete->l2_np,
       &rete->l2_nd,&rete->l3_np,&rete->l3_nd);

fscanf(f,"%[\n]s",buff);
fscanf(f,"\\t%lf\\t%d\\t%lf\\n",&rete->EQ,&rete->iterazioni,&rete->EQM);

/* Salva strato 1 */

fscanf(f,"%[\n]s",buff);
for(int i = 0; i<rete->l1_np; i++)
{
    fscanf(f,"\\t");
    for(int j=0;j<rete->l1_nd+1;j++)
    {
        fscanf(f,"%lf\\t",&rete->v_t[i*(rete->l1_nd+1)+j]);
    }
    fscanf(f,"\\n");
}

/* Salva strato 2 */

fscanf(f,"%[\n]s",buff);
for(int i = 0; i<rete->l2_np; i++)
{
    fscanf(f,"\\t");
    for(int j=0;j<rete->l2_nd+1;j++)
    {
        fscanf(f,"%lf\\t",&rete->v_u[i*(rete->l2_nd+1)+j]);
    }
    fscanf(f,"\\n");
}

/* Salva strato 3 */
fscanf(f,"%[\n]s",buff);
for(int i = 0; i<rete->l3_np; i++)
{
    fscanf(f,"\\t");
    for(int j=0;j<rete->l3_nd+1;j++)
    {
        fscanf(f,"%lf\\t",&rete->v_v[i*(rete->l3_nd+1)+j]);
    }
    fscanf(f,"\\n");
}

return rete;
}

void rete_Salva(rete_rete * rete, FILE * f)
{

```

```

fprintf(f, "#Distribuzione dei neuroni sulla rete\n");
fprintf(f, "\t");
fprintf(f, "%d\t%d\t%d\t%d\t%d\n",
    rete->N_neuroni_sensitivi,
    rete->N_neuroni_fferenti,
    rete->N_neuroni_primo_strato_intercalare,
    rete->N_neuroni_secondo_strato_intercalare,
    rete->N_strati_computazionali);
fprintf(f, "#Neuroni e dendriti\n");
fprintf(f, "\t");
fprintf(f, "%d\t%d\t%d\t%d\t%d\t%d\n",
    rete->l1_np, rete->l1_nd, rete->l2_np,
    rete->l2_nd, rete->l3_np, rete->l3_nd);

fprintf(f, "#Qualità rete\n");
fprintf(f, "\t");
fprintf(f, "%lf\t%d\t%lf\n", rete->EQ, rete->iterazioni, rete->EQM);

/* Salva strato 1 */
fprintf(f, "#Strato t\n");
for(int i = 0; i<rete->l1_np; i++)
{
    fprintf(f, "\t");
    for(int j=0;j<rete->l1_nd+1;j++)
    {
        fprintf(f, "%lf\t", rete->v_t[i*(rete->l1_nd+1)+j]);
    }
    fprintf(f, "\n");
}

/* Salva strato 2 */
fprintf(f, "#Strato u\n");
for(int i = 0; i<rete->l2_np; i++)
{
    fprintf(f, "\t");
    for(int j=0;j<rete->l2_nd+1;j++)
    {
        fprintf(f, "%lf\t", rete->v_u[i*(rete->l2_nd+1)+j]);
    }
    fprintf(f, "\n");
}

/* Salva strato 3 */
fprintf(f, "#Strato v\n");
for(int i = 0; i<rete->l3_np; i++)
{
    fprintf(f, "\t");
    for(int j=0;j<rete->l3_nd+1;j++)
    {
        fprintf(f, "%lf\t", rete->v_v[i*(rete->l3_nd+1)+j]);
    }
}

```

```

        }
        fprintf(f,"\\n");
    }

}

rele_rete * rele_Addestra(
    rele_rete * rn,
    rele_parametri * par,double * dati,double * classi)
{
    int l1_np = rn->l1_np;
    int l2_np = rn->l2_np;
    int l3_np = rn->l3_np;
    int l1_nd = rn->l1_nd;
    int l2_nd = rn->l2_nd;
    int l3_nd = rn->l3_nd;

    rn->iterazioni++;

    /* preparazione degli ingressi e delle label delle classi */
    rn->v_x0[0] = 1;
    memcpy(rn->v_x0+1,dati,rn->N_neuroni_sensitivi*sizeof(double));
    /* Carica il target-output (l'output voluto) in memoria */
    memcpy(rn->v_d,classi,rn->N_neuroni_afferenti*sizeof(double));

    /** RETE A UN SOLO STRATO ***/
    if(rn->N_strati_computazionali == 1)
    {
        /* Feed Forward: Input->L1->output to L2 */
        layer_feed_forward(rn->v_s1,rn->v_y1,rn->v_t,rn->v_x0,l1_np,l1_nd);

        rn->EQ = 0;
        /* Propagazione inversa dell'errore in L1 (v_t <- v_y1) */
        for(int i=0;i<l1_np;i++)
        {
            double errore = rn->v_d[i]- rn->v_y1[i];
            /* aggiornamento EQ */
            rn->EQ += (1.0/(double)l1_np)*errore*errore;
            /* correzione dei pesi (v_t) del percettore i-esimo */
            perc_correzione(rn->v_t+i*(l1_nd+1),rn->v_x0,
                            rn->v_s1[i],errore,
                            par->fattore_apprendimento,l1_nd);
        }
        aggiorna_EQM(rn);
        return rn;
    }

    /** RETE A DUE STRATI ***/
    if(rn->N_strati_computazionali == 2)

```

```

{
    /* Feed Forward: Input->L1->output to L2 */
    layer_feed_forward( rn->v_s1, rn->v_y1, rn->v_t, rn->v_x0, l1_np, l1_nd);

    /* Mappa y1 in x1 aggiungendo l'elemento x1_0=1*/
    layer_map_out_in(rn->v_x1, rn->v_y1, l2_nd);

    /* Feed Forward: L2->output */
    layer_feed_forward(rn->v_s2, rn->v_y2, rn->v_u, rn->v_x1, l2_np, l2_nd);

    rn->EQ = 0;
    /** Propagazione inversa dell'errore in L2 (v_u <- v_y1) */
    for(int i=0;i<l2_np;i++)
    {
        double errore = rn->v_d[i]- rn->v_y2[i];
        rn->v_dy2[i] = errore;
        rn->EQ += (1./(double)l2_np)*errore*errore;
        /* correzione dei pesi (v_u) del percettrone i-esimo
           La correzione viene memorizzata su un array temporaneo
           per completare la backpropagation
        */
        perc_correzione(
            rn->v_u_tmp+i*(l2_nd+1),
            rn->v_x1,
            rn->v_s2[i],
            errore,
            par->fattore_apprendimento,
            l2_nd);
    }

    /** Propagazione inversa dell'errore in L1 (v_t <- v_y2)*/
    for(int i=0;i<l1_np;i++)
    {
        double dd = 0;
        for(int j=0;j<l2_np;j++)
        {
            /* w: peso del i-esimo dendrite del j-esimo percettrone
               dello strato più esterno */
            double w = rn->v_u[j*(l2_nd+1)+i];
            /* correzione */
            dd=dd+w*(rn->v_dy2[j])*Dactiv_function(rn->v_s2[j]);
        }

        /* correzione del percettrone i-esimo dello strato t */
        perc_correzione( rn->v_t+i*(l1_nd+1),
                        rn->v_x0, rn->v_s1[i],
                        dd, par->fattore_apprendimento, l1_nd);
    }
}

```

/\*

```

Aggiornamento dello strato L2: i pesi della rete dello
strato u temporaneo
vengono copiati sullo strato u
*/
memcpy(rn->v_u,rn->v_u_tmp,sizeof(double)*l2_np*(l2_nd+1));
aggiorna_EQM(rn);
return rn;
}

/** TRE STRATI */
if(rn->N_strati_computazionali == 3)
{
/* Feed Forward: x0->L1->y1 */
layer_feed_forward(rn->v_s1,rn->v_y1,rn->v_t,rn->v_x0,l1_np,l1_nd);

/* Mappa y1 in x1 aggiungendo l'elemento x1_0=1*/
layer_map_out_in(rn->v_x1, rn->v_y1,l2_nd);

/* Feed Forward: x1->L2->y2 */
layer_feed_forward(rn->v_s2,rn->v_y2,rn->v_u,rn->v_x1,l2_np,l2_nd);

/* Mappa y2 in x2 aggiungendo l'elemento x3_0=1*/
layer_map_out_in(rn->v_x2, rn->v_y2,l3_nd);

/* Feed Forward: x2->L3->y3 */
layer_feed_forward(rn->v_s3,rn->v_y3,rn->v_v,rn->v_x2,l3_np,l3_nd);

rn->EQ = 0;

/** Propagazione inversa dell'errore in L3 (v_v <- v_y3) */
for(int i=0;i<l3_np;i++)
{
    double errore = rn->v_d[i]- rn->v_y3[i];
    rn->v_dy3[i] = errore;
    rn->EQ += (1./(double)l3_np)*errore*errore;
    /* correzione dei pesi (v_v_tmp) del perceptron i-esimo */
    perc_correzione(
        rn->v_v_tmp+i*(l3_nd+1),
        rn->v_x2,
        rn->v_s3[i],
        errore,
        par->fattore_apprendimento,
        l3_nd);
}

/** Propagazione inversa dell'errore in L2 (v_u <- v_y3)*/

for(int i=0;i<l2_np;i++)
{
    double dd=0;
}

```

```

    for(int j=0;j<l3_np;j++)
    {
        /* w: peso del i-esimo dendrite del j-esimo percepitrone
           dello strato più esterno */
        double w = rn->v_v[j*(l3_nd+1)+i];
        /* correzione */
        dd=dd+w*(rn->v_dy3[j])*Dactiv_function(rn->v_s3[j]);
    }
    rn->v_dy2[i] = dd;
    /* correzione del percepitrone i-esimo*/
    perc_correzione( rn->v_u_tmp+i*(l2_nd+1),
                      rn->v_x1,rn->v_s2[i],
                      dd, par->fattore_apprendimento,l2_nd);
}

/** Propagazione inversa dell'errore in L2 (v_t <- v_y2) */
for(int i=0;i<ll_np;i++)
{
    double dd=0;
    for(int j=0;j<l2_np;j++)
    {
        /* w: peso del i-esimo dendrite del j-esimo percepitrone
           dello strato più esterno */
        double w=rn->v_u[j*(l2_nd+1)+i];
        /* correzione */
        dd=dd+w*(rn->v_dy2[j])*Dactiv_function(rn->v_s2[j]);
    }
    /* correzione del percepitrone i-esimo*/
    perc_correzione( rn->v_t+i*(ll_nd+1),
                      rn->v_x0,rn->v_s1[i],
                      dd, par->fattore_apprendimento,l1_nd);
}
/* Aggiornamento rete */
memcpy(rn->v_v,rn->v_v_tmp,sizeof(double)*l3_np*(l3_nd+1));
memcpy(rn->v_u,rn->v_u_tmp,sizeof(double)*l2_np*(l2_nd+1));
aggiorna_EQM(rn);
return rn;
}
}

void rele_Classifica(rele_rete * rn, double * dati)
{
    int l1_np = rn->l1_np;
    int l2_np = rn->l2_np;
    int l3_np = rn->l3_np;
    int l1_nd = rn->l1_nd;
}

```

```

int l2_nd = rn->l2_nd;
int l3_nd = rn->l3_nd;

/* preparazione degli ingressi e delle label delle classi*/
rn->v_x0[0] = 1;
memcpy(rn->v_x0+1,dati,rn->N_neuroni_sensitivi*sizeof(double));
/* UNO STRATO */
if(rn->N_strati_computazionali == 1)
{
    layer_feed_forward(
        rn->v_s1,
        rn->v_y1,
        rn->v_t,
        rn->v_x0,
        l1_np,
        l1_nd);
    rn->strato_uscita = rn->v_y1;

}

/* DUE STRATI */
if(rn->N_strati_computazionali == 2)
{
    layer_feed_forward(
        rn->v_s1,
        rn->v_y1,
        rn->v_t,
        rn->v_x0,
        l1_np,
        l1_nd);

    /** Mappa y1 in x1 aggiungendo l'elemento x1_0=1*/
    layer_map_out_in(rn->v_x1,
                      rn->v_y1,
                      l2_nd);

    /** Feed Forward: L1->L2->output*/
    layer_feed_forward(rn->v_s2,
                      rn->v_y2,
                      rn->v_u,
                      rn->v_x1,
                      l2_np,
                      l2_nd);

    rn->strato_uscita = rn->v_y2;
}

/* TRE STRATI */
if(rn->N_strati_computazionali == 3)

```

```

{

    layer_feed_forward(
        rn->v_s1,
        rn->v_y1,
        rn->v_t,
        rn->v_x0,
        l1_np,
        l1_nd);

    /** Mappa y1 in x1 aggiungendo l'elemento x1_0=1*/
    layer_map_out_in(rn->v_x1,
                      rn->v_y1,
                      l2_nd);

    /** Feed Forward: L1->L2->output*/
    layer_feed_forward(rn->v_s2,
                       rn->v_y2,
                       rn->v_u,
                       rn->v_x1,
                       l2_np,
                       l2_nd);

    /** Mappa y2 in x2 aggiungendo l'elemento x2_0=1*/
    layer_map_out_in(rn->v_x2,
                      rn->v_y2,
                      l3_nd);

    /** Feed Forward: L1->L2->output*/
    layer_feed_forward(rn->v_s3,
                       rn->v_y3,
                       rn->v_v,
                       rn->v_x2,
                       l3_np,
                       l3_nd);

    rn->strato_uscita = rn->v_y3;
}

}

```

```

void rele.Libera_rete(rele_rete * rete)
{
    free(rete->v_x0);
    free(rete->v_t);
    free(rete->v_s1);

```

```

free(rete->v_y1);

if( rete->N_strati_computazionali >=2)
{
    free(rete->v_x1);
    free(rete->v_u);
    free(rete->v_u_tmp);
    free(rete->v_s2);
    free(rete->v_y2);
    free(rete->v_dy2);
}

if( rete->N_strati_computazionali >=3)
{
    free(rete->v_x2);
    free(rete->v_v);
    free(rete->v_v_tmp);
    free(rete->v_s3);
    free(rete->v_y3);
    free(rete->v_dy3);
}

free(rete);
return;
}

/*** rele_crea_rete ***/
rele_rete * rele_Crea_rete(
    int N_neuroni_sensitivi,
    int N_neuroni_afferenti,
    int N_neuroni_primo_strato_intercalare,
    int N_neuroni_secondo_strato_intercalare)
{
    rele_rete * rn = malloc(sizeof(rele_rete));

    rn->N_neuroni_afferenti = N_neuroni_afferenti;
    rn->N_neuroni_sensitivi = N_neuroni_sensitivi;
    rn-> N_neuroni_primo_strato_intercalare =
        N_neuroni_primo_strato_intercalare;
    rn-> N_neuroni_secondo_strato_intercalare =
        N_neuroni_secondo_strato_intercalare;

    if(rn == 0) exit (1);

    /* Errore quadratico*/
    rn->EQ = 0;
    rn->EQM = 0;
    rn->iterazioni = 0;
}

```

```

/* target output */
rn->v_d = malloc(N_neuroni_afferenti*sizeof(double));

int l1_np = 0, l1_nd = 0;
int l2_np = 0, l2_nd = 0;
int l3_np = 0, l3_nd = 0;

/* UN SOLO STRATO COMPUTAZIONALE */
if(N_neuroni_primo_strato_intercalare == 0 &&
   N_neuroni_secondo_strato_intercalare == 0)
{
    rn->N_strati_computazionali = 1;
    /* percettroni del primo strato */
    l1_np = N_neuroni_afferenti;
    /* dendriti del primo strato iintercalare */
    l1_nd = rn->N_neuroni_sensitivi;
}

/* DUE STRATI COMPUTAZIONALI */
if(N_neuroni_primo_strato_intercalare !=0 &&
   N_neuroni_secondo_strato_intercalare == 0)
{
    rn->N_strati_computazionali = 2;
    /* percettroni del primo strato */
    l1_np = rn-> N_neuroni_primo_strato_intercalare;
    /* dendriti del primo strato iintercalare */
    l1_nd = rn->N_neuroni_sensitivi;
    /* percettroni del secondo strato */
    l2_np = N_neuroni_afferenti;
    /* dendriti secondo strato */
    l2_nd = l1_np;
}

/* TRE STRATI COMPUTAZIONALI */
if(N_neuroni_primo_strato_intercalare != 0 &&
   N_neuroni_secondo_strato_intercalare != 0)
{
    rn->N_strati_computazionali = 3;
    /* percettroni del primo strato */
    l1_np = rn-> N_neuroni_primo_strato_intercalare;
    /* dendriti del primo strato iintercalare */
    l1_nd = rn->N_neuroni_sensitivi;
    /* percettroni del secondo strato */
    l2_np = N_neuroni_secondo_strato_intercalare;
    /* dendriti secondo strato */
    l2_nd = l1_np;
    /* neuroni del terzo strato (computazionale) intercalare */
    l3_np = N_neuroni_afferenti;
    /* dendriti del terzo strato iintercalare */
}

```

```

    l3_nd = l2_np;
}

rn->l1_np = l1_np;
rn->l2_np = l2_np;
rn->l3_np = l3_np;
rn->l1_nd = l1_nd;
rn->l2_nd = l2_nd;
rn->l3_nd = l3_nd;

/* input dei percettroni dello strato 0*/
rn->v_x0 = malloc((l1_nd+1)*sizeof(double));
/* un vettore di l1_nd dendriti per ognuno dei l1_np neuroni*/
rn->v_t = malloc((l1_nd+1)*l1_np*sizeof(double));
/*vettore della somma pesata dei canali dendritici per ogni neurone*/
rn->v_s1 = malloc(l1_np*sizeof(double));
/* vettore dell'output di ogni neurone*/
rn->v_y1 = malloc(l1_np*sizeof(double));

/* inizializzazione pseudocasuale delle connessioni
   da -MAX_PESO a + MAX_PESO */
for(int i=0;i<(l1_nd+1)*l1_np;i++)
    rn->v_t[i]=2*MAX_PESO*(double)rand()/(double)RAND_MAX-MIN_PESO;

/* RETURN si tratta si un percettrone a singolo strato*/
if(rn->N_strati_computazionali == 1) return rn;

/* input dei percettroni dallo strato 1*/
rn->v_x1 = malloc((l2_nd+1)*sizeof(double));
/* un vettore di l1_nd dendriti per ognuno dei l1_np neuroni*/
rn->v_u = malloc((l2_nd+1)*l2_np*sizeof(double));
/* vettore temporaneo per l'update*/
rn->v_u_tmp = malloc((l2_nd+1)*l2_np*sizeof(double));
/*vettore della somma pesata dei canali dendritici per ogni neurone*/
rn->v_s2 = malloc(l2_np*sizeof(double));
/* vettore dell'output di ogni neurone*/
rn->v_y2 = malloc(l2_np*sizeof(double));
/* errore: differenza tra target e ottenuto */
rn->v_dy2 = malloc(l2_np*sizeof(double));

/* inizializzazione pseudocasuale delle connessioni */
for(int i=0;i<(l2_nd+1)*l2_np;i++)
    rn->v_u[i]=MAX_PESO*(double)rand()/(double)RAND_MAX-MIN_PESO;
/* copia i pesi inizializzati nel vettore temporaneo */
memcpy(rn->v_u,rn->v_u_tmp,sizeof(double)*l2_np*(l1_nd+1));

/* RETURN si tratta si un percettrone a due strati*/
if(rn->N_strati_computazionali == 2) return rn;

/* input dei percettroni dello strato 2*/

```

```

rn->v_x2 = malloc((l3_nd+1)*sizeof(double));
/* un vettore di l3_nd dendriti per ognuno dei l1_np neuroni*/
rn->v_v = malloc((l3_nd+1)*l3_np*sizeof(double));
/* un vettore temporaneo di l3_nd dendriti per ognuno dei l1_np neuroni*/
rn->v_v_tmp = malloc((l3_nd+1)*l3_np*sizeof(double));
/* vettore della somma pesata dei canali dendritici per ogni neurone*/
rn->v_s3 = malloc(l3_np*sizeof(double));
/* vettore dell'output di ogni neurone*/
rn->v_y3 = malloc(l3_np*sizeof(double));
/* errore: differenza tra target e ottenuto */
rn->v_dy3 = malloc(l3_np*sizeof(double));

/* inizializzazione pseudocasuale delle connessioni */
for(int i=0;i<(l3_nd+1)*l3_np;i++)
    rn->v_v[i]=2*MAX_PESO*(double)rand()/(double)RAND_MAX-MIN_PESO;

/* RETURN */
return rn;

}

/***
IMPLEMENTAZIONI:
SEZIONI FUNZIONI PRIVATE
***/

void layer_feed_forward
(double v_s[],double v_y[],double v_w[],double v_x[],int n_perc, int n_dend)
{
    for(int i=0;i<n_perc;i++)
    {
        /*calcola l'output per ogni perceptrone*/
        v_s[i]=perc_calc_output(v_w+i*(n_dend+1),v_x,n_dend);
        v_y[i]=activ_function(v_s[i]);
    }
}

void perc_correzione
(double v_w[],double v_x[],double z,double d,double rate,int n_dend)
{
    /* ciclico sui dendriti */
    for(int i=0;i<n_dend+1;i++)
    {
        v_w[i]=v_w[i]+rate*v_x[i]*(d)*Dactiv_function(z);
    }
}

void layer_map_out_in(double v_x[],double v_y[], int n_dend)
{

```

```

v_x[0]=1;
for(int i=1;i<n_dend+1;i++) v_x[i]=v_y[i-1];
}

double perc_calc_output(double v_w[],double v_x[],int n_dend)
{
    double a=0;
    /*somma pesata degli stimoli di ingresso*/
    for(int i=0;i<n_dend+1;i++) a=a+v_w[i]*v_x[i];

    /*Attivazione del percettrone*/
    return a;
}

double activ_function(double summed_input)
{
    /* Come scelta alternativa della funzione di attivazione */
    // double r=tanh(summed_input);

    double r=1/(1+exp(-summed_input));
    return r;
}

double Dactiv_function(double summed_input)
{
    /* Se si scegli la tanh come attivazione */
    //double r=tanh(summed_input);
    //return 1-r*r;

    double r=activ_function(summed_input);
    return r*(1-r);
}

void aggiorna_EQM(rele_rete * rn)
{
    rn->EQM = ((rn->iterazioni-1)*rn->EQM+rn->EQ)/(double)rn->iterazioni;
}

```

## 4 Il cognitrone e il neocognitron di Fukushima

All'inizio di questa sezione abbiamo provato a comprendere le basi generali del funzionamento della trasmissione di un impulso nervoso tra i neuroni collegati insieme in una rete. Ci sono aspetti che non abbiamo trattato, per esempio non abbiamo visto il ruolo dei neurotrasmettitori come la serotonina, la dopamina ecc... e delle connessioni inibitorie. Per chi si vuole interessare alla ricerca in questo campo, la conoscenza approfondita del modello biologico, cioè di come funziona davvero il cervello e in generale il sistema nervoso è un dovere imprescindibile. In questo paragrafo discuterò l'evoluzione del modello del percettrone estendendo la modellazione matematica anche alle connessioni inibitorie.

### Le sinapsi

Abbiamo visto che una sinapsi in matematica può essere modellata come un elemento di una matrice o di un vettore, e che aumentarne o diminuirne il peso è una questione di poco conto, lo possiamo fare con una semplice operazione. In questo modo possiamo modellare delle reti che realizzano il numero di variazioni sinaptiche che riteniamo necessario. In natura però le cose non stanno esattamente così. Quando un neurone riceve un segnale nervoso da un neurone ad esso collegato (diremo pre-sinaptico) per rinforzare il detto collegamento esso (il neurone post-sinaptico) deve secernere una sostanza che raggiungendo la connessione sinaptica mediante reazioni chimiche la rinforzerà, oppure, come ora vedremo la inibirà. Bene, potete immaginare che, trattandosi di sostanze reali, che hanno una massa, non sono disponibili all'infinito, la loro disponibilità dipende da diverse condizioni sia fisiologiche che patologiche. Nel modello informatico non c'è un limite al numero di connessioni che un neurone può rinforzare, ma in quello biologico invece c'è eccome. Immaginate che un neurone post-sinaptico debba rinforzare mille cellule pre-

sinaptiche, sicuramente avrà a disposizione meno sostanza rispetto al caso in cui debba rinforzarne solo una.

A questo quadro, che va complicandosi, aggiungiamo anche che un neurone può "decidere" di inibire la connessione con un altro neurone, per esempio un neurone pre-sinaptico (potrebbe essere il caso di un neurone afferente) potrebbe decidere di aumentare la connessione con il neurone post-sinaptico che si è attivato maggiormente in risposta al suo stimolo, e di inibire le altre connessioni verso gli altri neuroni, ottenendo in questo modo di "risparmiare" sostanza da usare in modo più efficace.

La mia descrizione del fenomeno è assolutamente naif, ma il concetto che sto cercando di delineare è invece molto importante. Le reti neurali artificiali traggono la loro ispirazione dai modelli biologici, perciò la conoscenza e lo studio della biologia deve essere parte fondante di un informatico che se ne voglia occupare.

## Il modello

Il modello del cognitron fu proposto da Kunihiko Fukushima già nel 1975. Come Rosenblatt anche il modello originale di Fukushima si ispirava ai meccanismi neurali legati alle proprietà della vista alla capacità di rappresentare le immagini nel cervello. Il modello neurale del cognitron è sempre un modello a cellula con più ingressi ed una uscita. La novità rispetto al percettrone visto in precedenza è nel modo in cui vengono sommati i segnali afferenti dai canali di input (dendriti). Nel percettrone questi venivano sommati in base al peso della connessione, lo stesso accade nel cognitron ma accanto ai segnali eccitatori vengono introdotti anche dei segnali inibitori. In pratica l'equazione (3.7) che descrive il calcolo della somma degli ingressi, diventa per il cognitron la seguente:

$$s = \frac{1 + \mathbf{w} \cdot \mathbf{x}}{1 + \mathbf{g} \cdot \mathbf{z}} - 1 \frac{1 + \sum_{i=1}^N w_i x_i}{1 + \sum_{i=1}^M g_i z_i} - 1$$

dove  $\mathbf{w}$  e  $\mathbf{x}$  sono i pesi sinaptici e gli input agli  $N$  dendriti attivatori come nella (3.7), mentre  $\mathbf{g}$  e  $\mathbf{z}$  rappresentano gli  $M$  pesi e i segnali inibitori. La funzione di attivazione è la stessa della (3.8) dove alla disuguaglianza stretta ( $<$ ), Fukushima preferisce la disuguaglianza ( $\leq$ ).

Le idee proposte nel lavoro di Fukushima, prima nel 1975, poi nel 1980 dove con il neo-cognitron superò le difficoltà incontrate dal cognitron nel riconoscere un dettaglio in un'immagine quando questo veniva traslato, portarono allo sviluppo delle ben note CNN, cioè convolution neural network.

Dopo la prima edizione di questo libro, ho scritto *Reti neurali non supervisionate: il cognitron di Fukushima*, lo consiglio a chi vuole approfondire questo argomento.

# E da qui? Come andare avanti?

## Python e TensorFlow®

Se siete arrivati fin qui leggendo tutto con attenzione e facendo gli esempi di reti che ho proposto, una certa idea ve la sarete fatta e non sarete più degli estranei in questo affascinante mondo del machine learning. Se però pensate di essere già arrivati allora non avete capito bene il problema! Ora che avete imparato il C, in pochi giorni potreste acquisire le basi del linguaggio Python e usare un ambiente come TensorFlow® per compiere veri miracoli sul riconoscimento delle immagini, sull'elaborazione del linguaggio naturale e quant'altro. Vi suggerisco di farlo per capire il livello tecnologico raggiunto, poi con calma cercare di capire qual è il vero modello biologico usato dalle librerie e studiarne l'implementazione come abbiamo fatto fin qui insieme.

## Giocare per comprendere!

Un modo molto interessante per migliorare rapidamente nella programmazione è quello di giocare. Il gioco aiuta in tutte le attività umane quando è fatto con serietà e nel rispetto delle regole. Per iniziare vi suggerisco i giochi di **Tuki e Giuli**. Si tratta di *coding game* scritti in C in cui il giocatore muove un personaggio usando il codice sorgente anziché usare i tasti, il joystick o che altro. Vi siete chiesti chi sono il bruco e il tucano in copertina? Beh, sono proprio loro, Tuki il tucano e Giuli il bruco, li trovate a questo indirizzo <https://pumar.it/tuki>

## Applicazioni attuali e future!

Approfitto di questo paragrafo per chiarire un concetto importante. Nel modo in cui sono implementate attualmente le ANN, cioè come Macchine di Turing, esse non possono far nulla di più di quello che già non si può fare con una Macchina di Turing. Non sono stato chiaro? Provo ad esserlo, qualsiasi risultato

otteniate con una ANN potreste ottenerlo con un task specific algorithm, cioè un programma scritto specificatamente per un compito. Per esempio, oggi le ANN vengono usate per classificare le immagini che carichiamo sui *social*. Con un po' di pazienza e di testa, un buon programmatore potrebbe scrivere un codice non basato su una ANN, ma appunto task specific che faccia lo stesso. Detto questo, è vero che le ANN semplificano e velocizzano diverse attività. Non riporto qui la lista completa delle attuali applicazioni, perché sarebbe sempre e comunque datata ed è meglio googolarla in rete. Piuttosto ne elenco qualcuna che è, o è stata, di mio interesse per ragioni professionali o per passione personale:

- Ricerca di dettagli e strutture anatomiche in immagini mediche.
- Ricerca di periodicità in segnali clinici (ECG, EEG ecc.)
- Analisi del linguaggio naturale.
- Calcoli di dinamica e astrodinamica

Detto questo è possibile che le ANN superino il limite della Macchina di Turing? Questa è una bella domanda a cui io non so rispondere in modo ferrato, però alcune considerazioni si possono fare. Mi pare che al momento sul piatto ci siano due pezzi interessanti: le reti neurali biologiche (o comunque non digitali) e la computazione quantistica. Credo che questi siano i terreni fertili per le idee del futuro, però torno a dire, è solo una chiacchierata.

## Saluti

Carissimi lettori, giovani e meno giovani, donne e uomini e perché no, sistemi artificiali con capacità di apprendimento. Mi ha fatto estremo piacere scrivere questo libro, spero che ne abbiate trovato la lettura interessante, per qualsiasi commento o richiesta fate riferimento al profilo facebook della Scuola Sisini, che ri porto qui: <https://www.facebook.com/scuolasisini/>

## Riconoscimenti

- La copertina è una realizzazione di Valentina Sisini
- Edit delle formule matematiche: <https://www.mathjax.org/>
- Stili: <https://www.w3schools.com/css>
- Porte logiche nelle Fig. 2.2, 2.3, 2.4: adattate da [https://commons.wikimedia.org/wik/File:Circuit\\_elements.svg](https://commons.wikimedia.org/wik/File:Circuit_elements.svg)
- Circuito Half Adder in Fig. 2.5: adattato da [https://commons.wikimedia.org/wik/File:Half\\_Adder.svg](https://commons.wikimedia.org/wik/File:Half_Adder.svg)
- Circuito Half Adder in Fig. 2.6: adattato da <https://commons.wikimedia.org/wik/File:Full-adder.svg>
- Neurone in Fig.3.0 :adattato da [https://it.wikipedia.org/wik/File:Neuron\\_-\\_annotated.svg](https://it.wikipedia.org/wik/File:Neuron_-_annotated.svg)

## Bibliografia essenziale

- *Analisi matematica -I: funzioni di una variabile*, G. Zwirner, CEDAM, 2017.
- *Architettura e organizzazione dei calcolatori elettronici*, Giacomo Bucci, McGraw-Hill, 2005.
- *Fondamenti di Reti Neurali*, Tarun Khanna, Addison-Wesley, 1991.
- *Low-Level Programming*, Igor Zhirkiv, Apress, 2017.
- *Linguaggio C*, Brian W. Kernighan e Dennis M. Ritchie, Jakson, 1995.
- *Strutture, Logica, Linguaggi*, Luigia Carlucci Aiello e Flora Pirri, Addison Wesley, 2005.

## Link utili

- Gruppo facebook di supporto al libro:  
<https://www.facebook.com/groups/1229389267230026>
- Progetto Rele: <https://github.com/francescosisisini/ReLe>
- Altri listati del libro: <https://github.com/francescosisisini/LIBRO-Introduzione-alle-reti-neurali-codice>
- Canale youtube: <https://www.youtube.com/channel/UCDwLIFqa0xZ71PEOdHA0aSQ>
- Pagina facebook di scuolasisisini: <https://www.facebook.com/scuolasisisini>
- Sito ufficiale: <https://pumar.it>

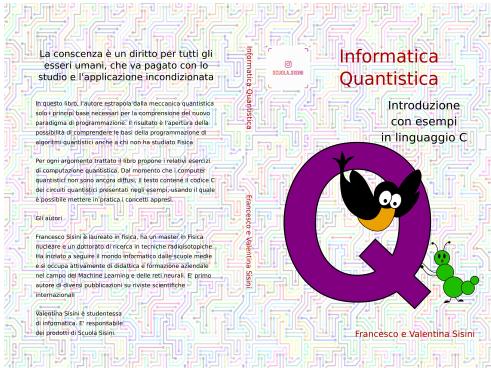
## Altre pubblicazioni di Scuola Sisini



## *Reti neurali non supervisionate: il cognitron di Fukushima*



## *Sfidare gli algoritmi: 5 videogiochi in C su Linux*



*Informatica quantistica: introduzione con esempi in linguaggio C*