Francesco Sisini

Introduzione alle reti neurali

con esempi in linguaggio C

Terza edizione

Edizioni: i Sisini Pazzi, 2020

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

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:

$$\sigma(x) = tanh(x) \tag{3.2}$$

  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

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 "mastica", 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' retró, 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 memorizzazione, 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/segnale/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/segnale 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.

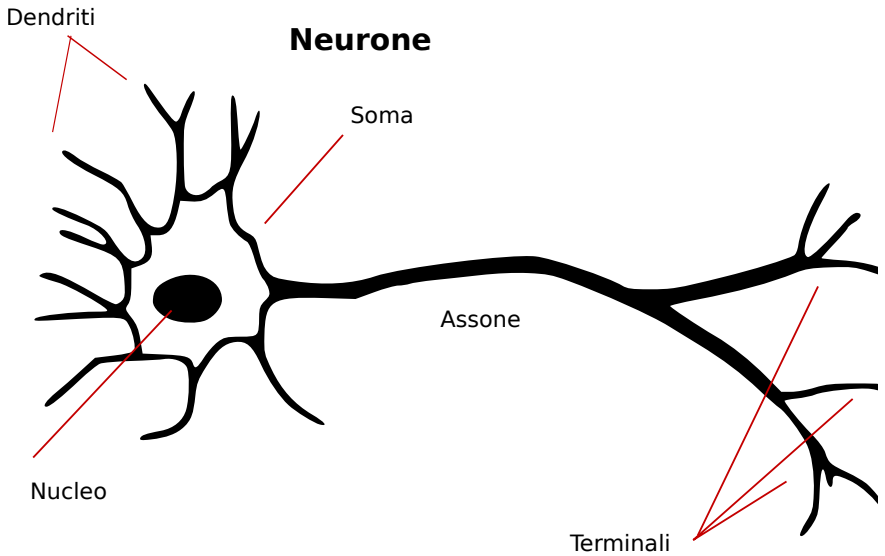

Fig. (a) - Schema di una cellula neurale (da Wikimedia)

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.

1.2 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:

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

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 4×3. 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.

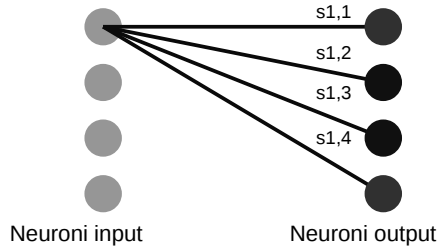

Fig.1.1 - Esempio di collegamento tra neuroni.

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 dicasi 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:

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

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

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*.

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:

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: (molto importante): $x + (x \cdot y) = x,\ x \cdot (x + y) = x,$ 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 · 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 I. 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 ($f(x, y, x)$) 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

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 li, 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.

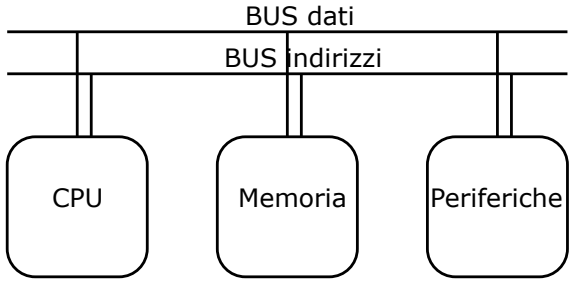

Fig. 2.8 - Collegamento a BUS tra memoria, CPU e periferiche.

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 architetturali 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 architetturali, 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 tornare 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 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 programmatori 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 li 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

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 uamano. 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 paragrfi introduttivi in modo da focalizzare l'attenzione sul concetto di cellula nervosa (neurone) sull'idea che essa possa essere *attivata* in conseguenza di un inpulso 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 configuraziona 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:

$$\Delta \mathbf{A}\, \mathbf{f} = \eta \mathbf{g} \tag{3.1}$$

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 η 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 $\mathbf{f}^{(1)}$:

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

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 $\Delta\mathbf{A}^{(3)}$:

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

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

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 Δt 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 (Δ), 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 λ 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 abbiamo 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 λ:

$$k = \lfloor \frac{1}{\lambda} \rfloor$$

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:

$$\mathbf{M}(t + k\Delta t) = 0 + \lambda \mathbf{f}\mathbf{s}^T + \ldots \lambda \mathbf{f}\mathbf{s}^T = k\lambda \mathbf{f}\mathbf{s}^T = \mathbf{f}\mathbf{s}^T \quad (3.6)$$

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 λ 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 $\mathbf{fs}^T$ e quindi il prodotto $k\lambda$ deve essere uguale ad 1. Questo impone o un vincolo su λ o, fissato λ, 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 i età, 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:

$$\mathbf{r}^{(1)} = \mathbf{M}\mathbf{s}^{(1)} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{s}^{(1)} = \begin{bmatrix} 0 \\ 2 \\ 0 \\ 0 \end{bmatrix} = 2 \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = 2\mathbf{f}^{(1)}$$

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 $\mathbf{r} = \mathbf{M}\mathbf{p}$ quindi si ha:

$$\mathbf{r} = \mathbf{M}\mathbf{p} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \mathbf{p} = \begin{bmatrix} 0 \\ 2 + \delta\alpha + \delta\epsilon \\ 0 \\ 0 \end{bmatrix} =$$

$$= (2 + \delta\alpha + \delta\epsilon) \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$$= (2 + \delta\alpha + \delta\epsilon)\mathbf{f}^{(1)}$$

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

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 venagano 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.

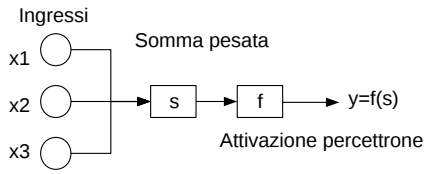

Fig. 3.4 - Schema a blocchi del percettrone di Rosenblatt.

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:

$$s = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^{n} w_i \, x_i + b \tag{3.7}$$

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

$$f(s) = \begin{cases} 1 & \text{se} \quad s > 0 \\ 0 & \text{se} \quad s \leq 0 \end{cases} \tag{3.8}$$

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:

$$s = \mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^{n} w_i \, x_i = b \times 1 + \sum_{i=1}^{n} w_i \, x_i \qquad (3.9)$$

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 $y = f(\mathbf{s})$
    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.

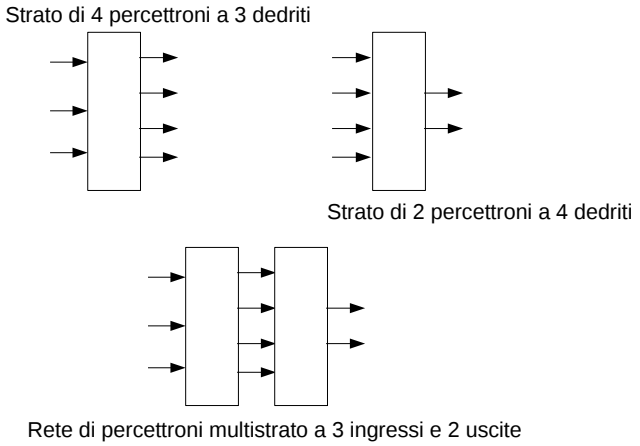

Fig. 3.8 - Schematizzazione della rete multistrato di percettroni in due strati.

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).

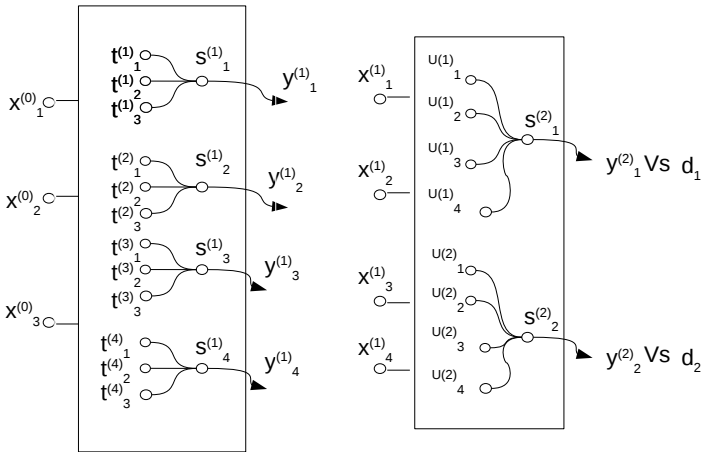

Fig. 3.7 bis - Rete a due strati di percettroni completamente connessi. Il primo strato (deep) presenta 4 percettroni a 3 ingressi, il scondo 2 percettroni a 4 ingressi. La figura evidenzia sia il modello a percettroni che la sua astrazione in strati. Il nome dei paramteri del modello è messo in evidenza.

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:

$$\Delta \mathbf{u}^{(i)} = \begin{bmatrix} 1 \times (d_i - y_i^{(2)}) f'(s_i^{(2)}) \\ y_1^{(1)} (d_i - y_i^{(2)}) f'(s_i^{(2)}) \\ y_2^{(1)} (d_i - y_i^{(2)}) f'(s_i^{(2)}) \\ y_3^{(1)} (d_i - y_i^{(2)}) f'(s_i^{(2)}) \\ y_4^{(1)} (d_i - y_i^{(2)}) f'(s_i^{(2)}) \end{bmatrix} \tag{3.10}$$

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 $\left(d - y^{(2)}\right)$. 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 $\left(d - y^{(2)}\right)$ 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:

$$\mathbf{u}^{(i)} = \begin{bmatrix} u_1^{(i)} \\ u_2^{(i)} \\ u_3^{(i)} \\ u_3^{(i)} \end{bmatrix}$$

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 η, 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:
    - Modifica i pesi delle connessioni dello strato 1 e 2 come: $\mathbf{t} = \mathbf{t} + \eta\Delta\mathbf{t}$ e $\mathbf{u} = \mathbf{u} + \eta\Delta\mathbf{u}$.
    - 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 percettroni, 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 percettroni. 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.

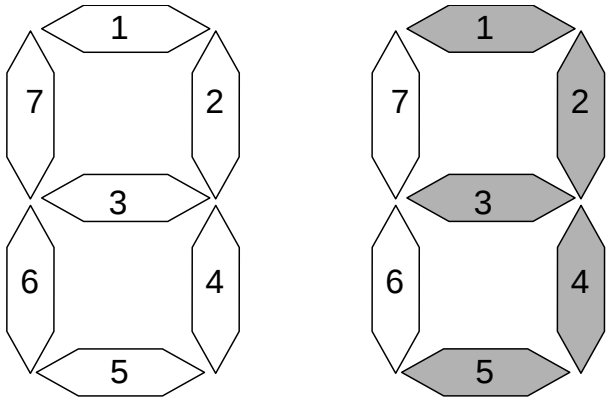

Fig. 3.9 - Display a sette segmenti.

Analisi Top Down

Il programma usa due file di configurazione, in uno sono registrati i pesi iniziali delle connessioni sinaptiche dei percettroni 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 percettroni del primo strato, il numero di dendriti dei percettroni, il numero di percettroni del secondo strato e il numero di dendriti dei percettroni 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 percettrone per mostrare il suo ruolo nella rete. Lo scopo di questo "design" applicativo è quello di evidenziare la presenza del percettrone e di mostrare la rete neurale come l'insieme connesso di più percettroni.

Configurazione e funzionamento

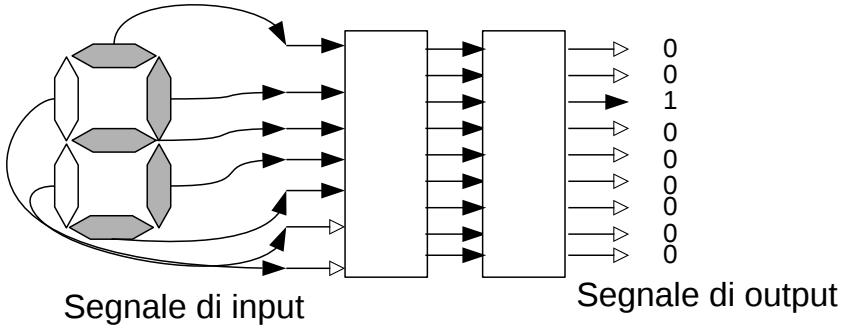

Fig. 3.10 - Configurazione della rete per il riconoscimento delle cifre digitali. Il terzo neurone di output dall'alto è attivo in corrispondenza del numero tre presentato ai neuroni di input.

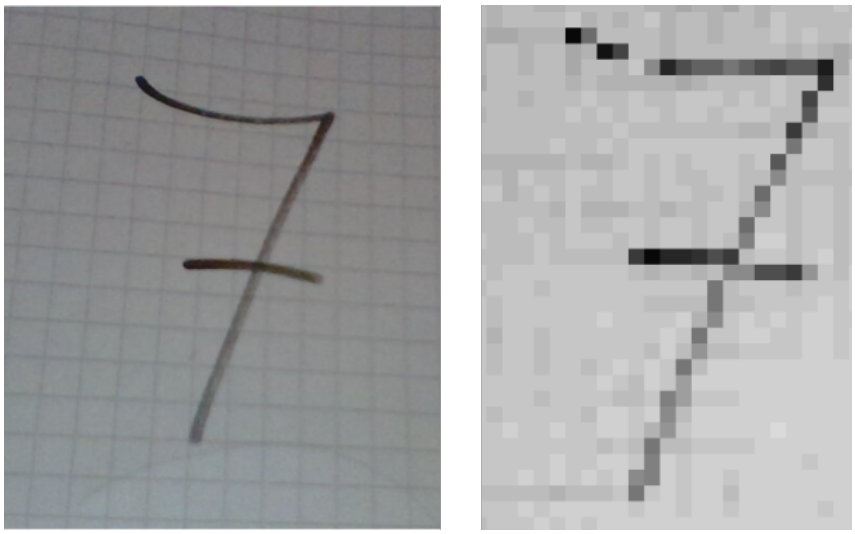

Fig. 3.11 - Esempio di digitalizzazione della cifra 7 scritta a mano.

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.

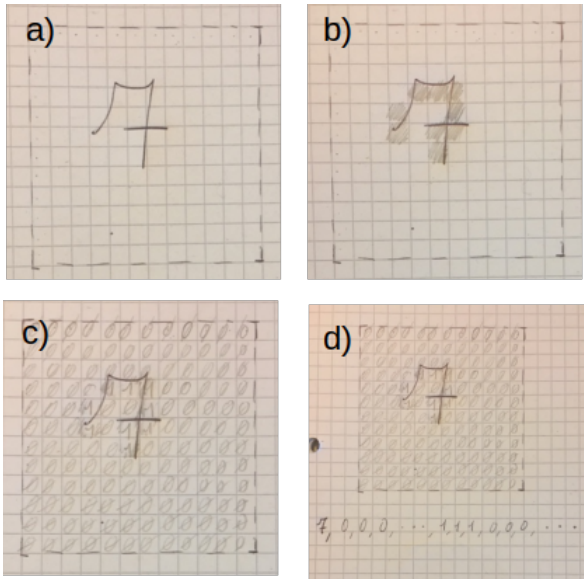

Fig. 3.12 - Procedura per la produzione dei dati di addestramento per una MLP atta a riconoscere le cifre. a) scrittura manuale di una cifra, b) selezione dei quadretti di interesse, c) valorizzazione dei quadretti a 0 e 1, d) scrittura della sequenza di 0 e 1 rappresentati la cifra, preceduta dal valore decimale della cifra stessa.

Il dabase 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

percettroni 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:

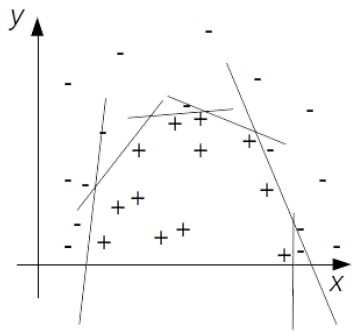

Fig. 3.19 - La regione dei dati indicati dalle crocette è definita attraverso le intersezioni di sei rette nel piano.

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 percettroni dello strato intercalare, cioè il secondo strato di percettroni.

Come abbiamo anticipato all'inizio del capitolo, due soli strati di percettroni 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:

$$\begin{cases} y > x + 5 \\ y < 6 - x \end{cases}$$

$$\begin{cases} y > x + 10 \\ y < 11 - x \end{cases}$$

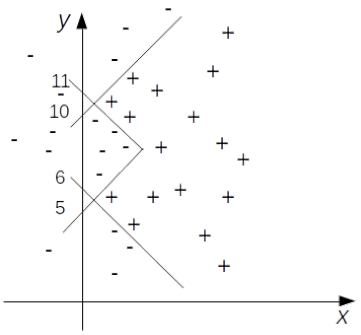

Fig. 3.20 - La regione dei dati indicati dalle crocette è definita attraverso le intersezioni di quattri rette nel piano. La regione presenta una concavità.

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:

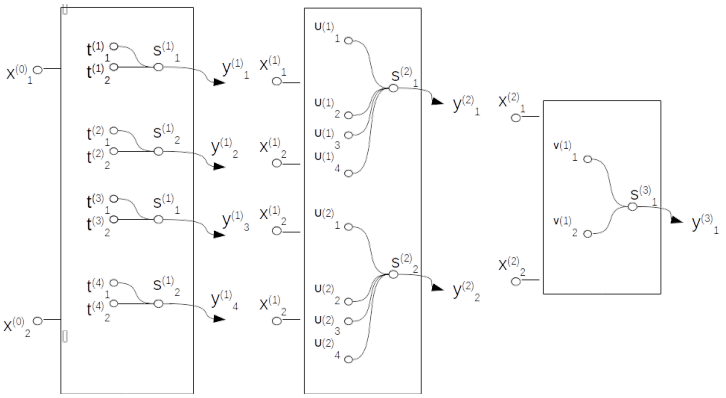

Fig. 3.21 - rete neurale a tre strati di percettroni.

Tre strati di percettroni 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.

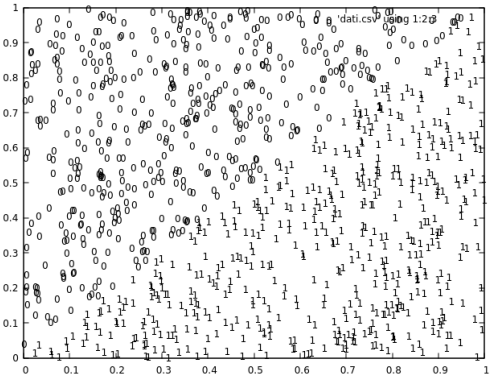

Fig. 3.23 - Classificazione di dati generati dalla funzione generatrice del programma addestra_uno_strato.c.

Addestramento di una rete a due e a tre strati

Per addestrare una rete a due strati è necessario definire il numero dei percettroni 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 cooridnate 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 = 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]=4.*(double)rand()/(double)RAND_MAX-2;
      d[1]=5*(double)rand()/(double)RAND_MAX;
      rele_Classifica(r, d);
      printf("%lf %lf %d\n",d[0],d[1],r->strato_uscita[0]>0.5);
    }
  rele_Libera_rete(r);
}
```

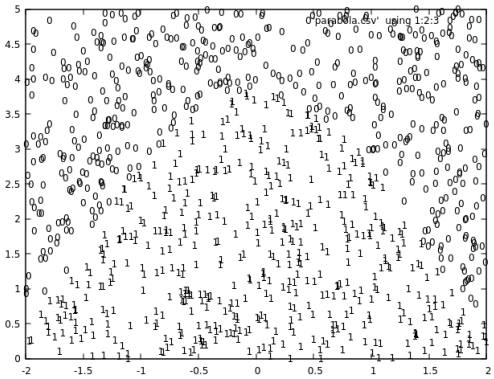

Fig. 3.24 - Classificazione di dati generati dalla funzione generatrice del programma `addestra_due_strati.c`.

L'esempio che segue, oltre ad implementare una rete a tre strati