

# DISPENSA DI INTELLIGENZA ARTIFICIALE

ROBERTA PRENDIN

Appunti del corso di Intelligenza Artificiale  
Corso di Laurea Magistrale in Computer Science  
Università Ca' Foscari di Venezia, Settembre 2015

Parte I

INTELLIGENZA ARTIFICIALE



## IL MODELLO NEURALE

---

Le reti neurali artificiali sono state create per **riprodurre processi biologici** tipici del cervello umano come la percezione di immagini, il riconoscimento di forme, la comprensione del linguaggio e il coordinamento senso-motorio. Le reti neurali biologiche presentano due interessanti caratteristiche:

- Sono **addestrabili e adattabili**, cioè possono imparare e adattarsi a nuove circostanze e stimoli esterni. Posta di fronte ad un problema, la rete neurale **non vuole diventarne un'esperta** – studiarne le caratteristiche per individuare un algoritmo che porti ad una soluzione; al contrario, la soluzione al problema nasce dopo una fase di *apprendimento* in cui la rete impara a partire da un *training set* fornito dal problema stesso.
- Sono architetture **massicciamente parallele**, efficienti e *fault tolerant*. Nel cervello non esiste un elemento che, al pari di una CPU, si occupi del controllo centralizzato delle cellule nervose: le diverse aree funzionano insieme, influenzandosi reciprocamente e contribuendo alla realizzazione di uno specifico compito. L'elemento fondamentale di ogni rete neurale, il neurone, è tra l'altro molto semplice, tanto che il mancato funzionamento di uno di essi non ha conseguenze sull'intero sistema. Di conseguenza, se molti neuroni smettono di funzionare le **prestazioni cerebrali degradano gradualmente** man mano che il numero di neuroni non funzionanti aumenta (*graceful degradation*).

Ne deriva che qualsiasi studio sulle reti neurali artificiali non possa prescindere dal **contributo fornito delle neuroscienze**, che si occupano dello studio del sistema nervoso umano con particolare riguardo per il cervello.

### 1.1 LA STRUTTURA NEURALE

Il sistema nervoso è composto da miliardi di cellule nervose interconnesse tra loro: si stima che ogni neurone sia connesso a circa

10-100'000 altri neuroni. L'unità base di organizzazione è la **corteccia cerebrale**, una colonna di tessuto nervoso contenente circa 20'000 neuroni e spessa 4mm. Ogni neurone può essere visto come una **struttura di calcolo elementare** composta da tre parti fondamentali<sup>1</sup>:

- **Soma** o corpo cellulare: l'unità di calcolo del neurone (5/10 micron) contenente il citoplasma e il nucleo cellulare;
- **Dendriti**: fibre di media lunghezza che ricevono in input dei segnali provenienti da altri neuroni;
- **Assone**: singola fibra molto lunga che si ramifica all'estremità per inviare l'output del neurone agli altri a cui è collegato.

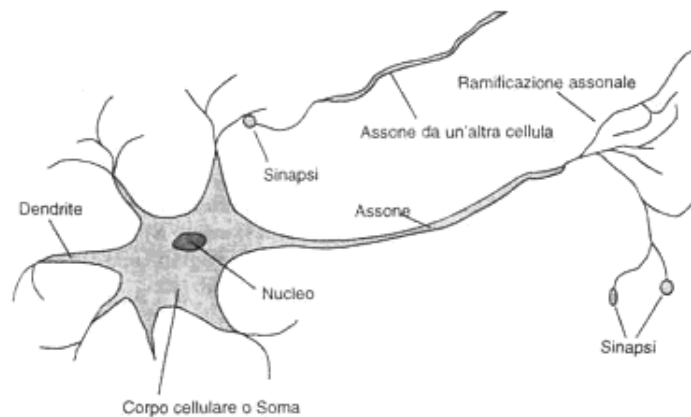


Figura 1: Schema generale di un neurone biologico.

Il punto di connessione tra ramificazione assonale e dendrite di neuroni diversi è detto **sinapsi**. La comunicazione vera e propria tra neuroni è un processo complesso che avviene tramite sostanze chimiche dette *neurotrasmettitori*; tale comunicazione è composta da queste fasi:

1. Il corpo cellulare esegue la **somma pesata** dei segnali in ingresso, provenienti dagli altri neuroni;
2. Se la somma pesata supera il **valore-soglia** il neurone *spara*: viene cioè prodotto un segnale elettrico, detto **potenziale d'azione**, che viene inviato all'assone. Se la somma pesata non supera il valore-soglia, invece, il neurone rimane *inattivo*.

<sup>1</sup> Questa struttura generale può variare a seconda della funzione del neurone e della specie animale considerata.

3. Quando il segnale elettrico raggiunge la sinapsi il **bottone pre-sinaptico** rilascia dei neurotrasmettitori che, combinandosi con il bottone post-sinaptico, provocano una diffusione del segnale elettrico nel neurone post-sinaptico.

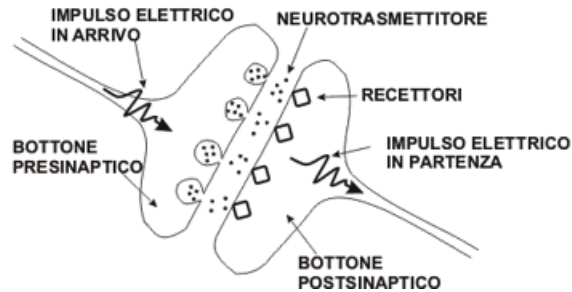


Figura 2: Sinapsi tra due neuroni.

Ogni sinapsi ha associata una certa **efficacia sinaptica**, l'ammontare cioè di segnale elettrico che entra nel neurone post-sinaptico rispetto al potenziale d'azione del neurone pre-sinaptico. Le sinapsi possono perciò favorire la generazione di potenziale d'azione nel neurone post-sinaptico (sinapsi eccitatorie) o inibire tale generazione (sinapsi inibitorie). L'**apprendimento** si attua proprio modificando l'efficacia sinaptica.

Concludendo, per riprodurre artificialmente il cervello umano occorre realizzare una rete di elementi molto semplici; tale struttura dev'essere **distribuita**, massicciamente parallela, capace di apprendere e di generalizzare, cioè di produrre output in corrispondenza di input non incontrati durante la fase d'addestramento. Anche se di recente introduzione, le reti neurali trovano valida applicazione in settori quali predizione, classificazione, riconoscimento e controllo, portando spesso contributi significativi alla soluzione di problemi difficilmente trattabili con metodologie classiche.



## MODELLO DI MCCULLOCH & PITTS

Nel 1943 W. McCulloch e W. Pitts presentarono un **modello artificiale di neurone biologico** modellato come un *binary threshold unit*. Ogni neurone è dotato di **più ingressi** ma presenta **una sola uscita**; ciascun ingresso ha inoltre associato un peso,  $w_{in}$ , che ne determina la conducibilità, e un valore-soglia  $\mu_i$ .

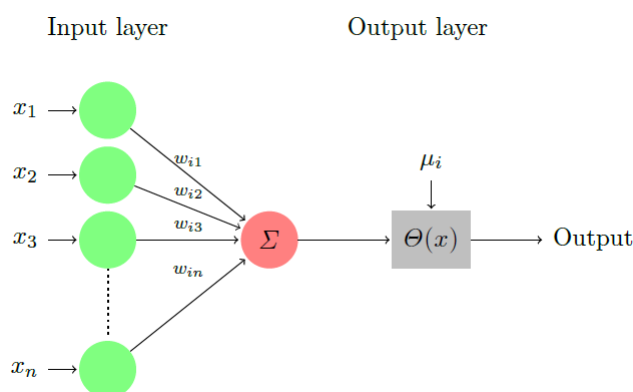


Figura 3: Esempio di perceptrone.

L'**attivazione del neurone** dipende dal valore della somma pesata degli ingressi (*net input*) che viene valutato dalla funzione d'attivazione  $\Theta$ : se il net input supera il valore-soglia  $\mu_i$  allora il neurone si attiva e produce un segnale, altrimenti rimane inerte. Nella versione più semplice la funzione d'attivazione è una **step function discreta**, come ad esempio la funzione *segno*:

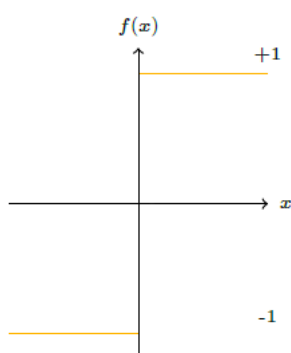


Figura 4: Funzione segno, una step function discreta.

In formule:



$$\text{Output} = \begin{cases} 0 & \text{se } \sum_{j=0}^n (w_{ij}x_j) > \mu_i \\ 1 & \text{altrimenti} \end{cases}$$

...dove  $h = \sum_j w_{ij}x_j$  è il net input dell' $i$ -esimo neurone. Nelle versioni più complesse e realistiche, invece, la funzione d'attivazione è *continua*: tra le più utilizzate troviamo la *sigmoidea* e la *tangente iperbolica* che mappano il risultato della somma pesata entro intervalli  $[0, 1]$  e  $[-1, 1]$  rispettivamente.

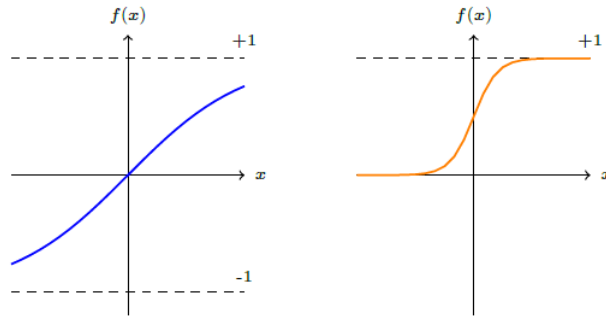


Figura 5: Funzioni continue non lineari: tangente iperbolica e sigmoidea.

Combinando opportunamente i neuroni di McCulloch & Pitts è possibile costruire reti capaci di funzionare come un **dispositivo booleano**, realizzando cioè AND, OR e NOT:

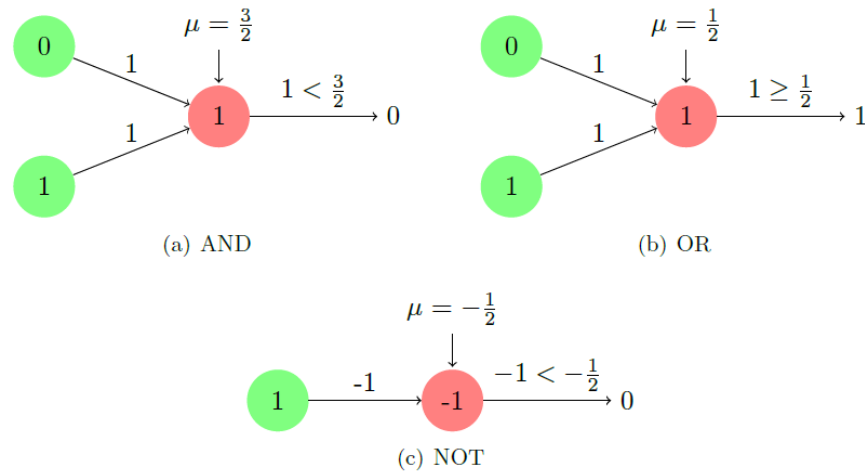


Figura 6: Operatori booleani ottenuti attraverso neuroni di McCulloch & Pitts.

È importante notare che in questo modello non avviene ancora alcun tipo di apprendimento: i pesi sono infatti **fissi e non aggiustati** di volta in volta a seconda dell'output prodotto dalla rete.

## 2.1 TIPI DI ARCHITETTURE DELLA RETE

Una insieme di neuroni artificiali forma una **rete neurale** rappresentabile tramite un **grafo diretto pesato**  $N = (V, E, w)$  dove  $V$  è l'insieme dei neuroni,  $E \subset V \times V$  è l'insieme di connessioni e  $w : E \rightarrow \mathbb{R}$  è una funzione che assegna un peso con valore reale  $w(i, j)$  per ogni connessione  $(i, j) \in E^1$ . I pesi rappresentano l'efficacia sinaptica: pesi positivi amplificano il segnale, pesi negativi lo inibiscono. In generale le reti neurali vengono classificate sulla base di **tre caratteristiche**:

- **presenza di cicli**: reti *feedforward* (o acicliche, dove cioè le connessioni avvengono in una sola direzione) e reti *feedback* (o ricorrenti o cicliche);
- **connettività**: reti fortemente o scarsamente connesse;
- **stratificazione**: reti a strato singolo e multistrato.

La scelta della rete neurale da utilizzare dipende dalla tipologia di problema che si vuole risolvere. In generale si identificano tre classi di reti:

- **reti *feedforward* ad uno strato**: reti di questo tipo contengono una serie di unità d'input e un singolo strato di neuroni d'output. Il segnale nella rete si propaga in avanti senza cicli, partendo dallo strato di input ed arrivando in quello di output. Si tratta di reti **statiche**, in cui cioè l'input originale non viene ricombinato con l'output via via prodotto dalla rete.

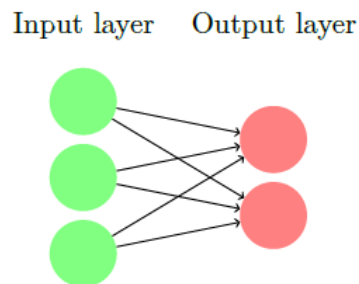


Figura 7: Rete *feedforward* a singolo strato.

- **reti *feedforward* a più strati**: tra lo strato d'input e lo strato neurale di output sono collocati uno o più strati di neuroni nascosti (*hidden layers*). Questo tipo di architettura fornisce alla re-

<sup>1</sup> Come notazione sarà utilizzata  $w_{ji}$  anziché  $w(i, j)$ .

te una **prospettiva globale** in quanto aumentano le interazioni tra neuroni. Anche in questo caso si tratta di reti **statiche**.

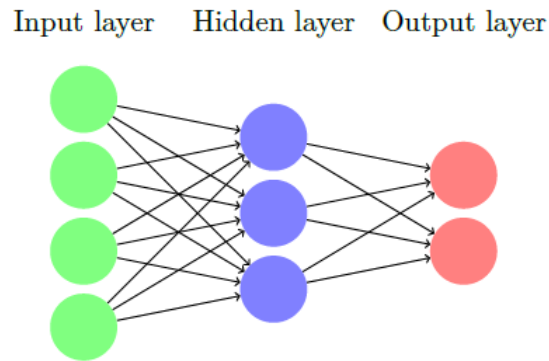


Figura 8: Rete *feedforward* multistrato.

- **reti feedback:** sono reti contenenti cicli grazie ai quali l'input fornito alla rete si ricombina con l'output fornito da quest'ultima. I cicli hanno un impatto profondo sulle capacità di apprendimento della rete e sulle sue performance, in quanto la rendono un sistema **dinamico**.

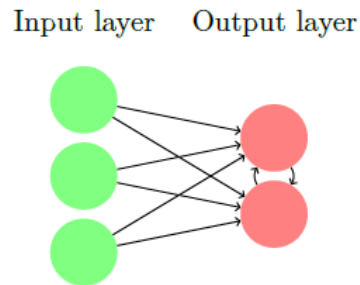


Figura 9: Rete ricorrente.

## PROBLEMA DI CLASSIFICAZIONE E PERCETTRONE

---

In un tipico problema di classificazione si hanno a disposizione due insiemi di dati:

- Un **insieme predefinito**<sup>1</sup> di **features** o caratteristiche tipiche dell'essenza di un oggetto:  $f_1, f_2, \dots, f_n$ ;
- Un **insieme di classi**:  $c_1, c_2, \dots, c_m$ ;

L'obiettivo è classificare correttamente gli oggetti – chiamati anche *pattern* – in base alle loro caratteristiche. Un tipico problema di classificazione può ad esempio riguardare le classi *melone*, *mela*, *arancia* e studiare le caratteristiche *peso*, *colore* e *dimensione* di determinati oggetti.

Poniamo ora di avere un problema di classificazione con  $n$  features: supponendo che queste siano misurabili o comunque rappresentabili numericamente, **ogni oggetto diventa un vettore** contenente  $n$  valori che, geometricamente, possono essere rappresentati come un punto in uno spazio  $n$ -dimensionale. I problemi di classificazione possono quindi essere trattati attraverso metodi geometrici. Per esempio, date le classi *mela* (0), *pera* (1) e le features *peso*, *altezza* definite nell'intervallo reale  $[0, +\infty)$ , otteniamo questa rappresentazione geometrica:

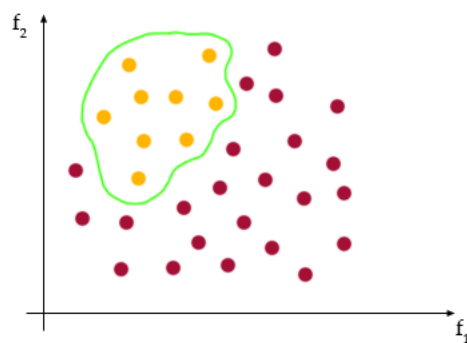


Figura 10: Un insieme di dati in uno spazio bidimensionale.

---

<sup>1</sup> In sistemi più elaborati la rete neurale è capace di individuare autonomamente le features, senza fissarle nella fase di progettazione.

Gli oggetti sono cioè rappresentati tramite un vettore (peso, altezza). Quelli appartenenti alla stessa classe ricadono all'interno della stessa area dello spazio  $n$ -dimensionale, che risulta così diviso in **regioni di decisione**. Risolvere un problema di classificazione significa quindi **determinare i confini** di queste regioni tramite uno specifico algoritmo; ogni nuovo oggetto è classificato semplicemente inserendolo in una regione di decisione.

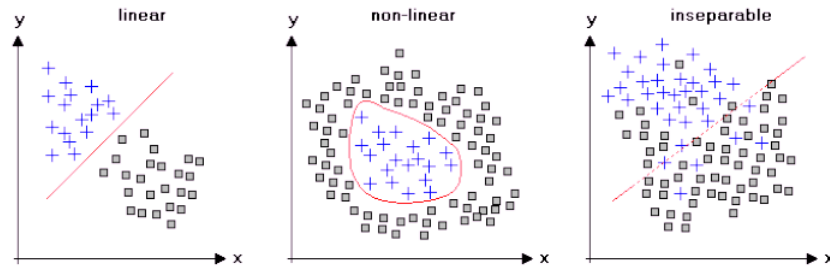


Figura 11: Rappresentazione geometrica di problemi di classificazione

Le regioni di decisione possono assumere forme diverse: in alcuni casi è possibile separare due classi semplicemente attraverso un piano mentre in altre abbiamo regioni contenute in altre regioni o situazioni ancora più complesse in cui la linea di separazione non è lineare.

### 3.1 PERCETTRONE A SINGOLO STRATO

La prima rete neurale costruita per risolvere problemi di classificazione è il *percettrone* di Rosenblatt (1958). Si tratta di una rete *feedforward* formata da un **singolo strato di neuroni di McCulloch & Pitts** e avente tante unità di input quante features e tanti neuroni d'output quante classi. Originariamente solo binario, nel tempo ha assunto forme più complesse.

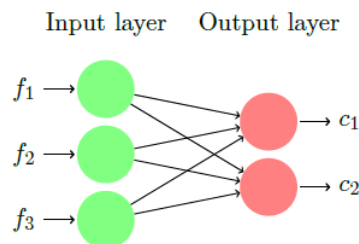


Figura 12: Rete per problema di classificazione (3 features, 2 classi).

Il percettrone risolve il problema di classificazione attraverso una *fase di apprendimento* durante la quale ha accesso ad un insieme di coppie (pattern, classe) detto *training set*; la rete **aggiusta i propri**

**pesi** per poter classificare correttamente i pattern forniti; quando sarà presentato un pattern nuovo appartenente ad una classe già appresa, la rete potrà classificarlo grazie alle informazioni estratte durante la fase di addestramento.

### 3.1.1 *L'incorporazione del valore-soglia*

Nel modello di McCulloch & Pitts ogni neurone è dotato di un valore-soglia  $\mu_i$ : tale valore soglia può però essere assimilato ai pesi della rete se aggiungiamo un'ulteriore unità di input bloccata  $-1$ . Per capire perché ricordiamoci che l'output di ogni neurone dipende dalla differenza tra il net input ricevuto e il suo valore soglia:

$$\text{Output} \left( \sum_j w_{ij} V_j - 1\mu_i \right)$$

Incorporiamo  $-1\mu_i$  nella somma e otteniamo:

$$\text{Output} \left( \sum_{j=1} w_{ij} V_j \right)$$

dove  $w_{i,j-1} = \mu_i$ . In questo modo la rete impara i valori-soglia esattamente come impara i pesi: possiamo quindi dimenticarci dei vari  $\mu_i$ .

### 3.1.2 *Limiti e potenzialità del Percettrone*

Il percettrone è un'idea semplice ed elegante: imita il neurone umano ed è in grado di imparare dagli esempi che gli vengono presentati. Tuttavia, come dimostrato da M. Minsky e S. Papert (1969), ha una **pesante limitazione**: è in grado di **risolvere solo problemi linearmente separabili**, cioè problemi in cui le regioni di decisione si possono separare con un iperpiano. Esempi giocattolo di regioni di decisione linearmente separabili sono gli **operatori logici** AND e OR: dati due input (0 o 1) e due possibili output (0 o 1) è possibile separarne i punti su un piano cartesiano attraverso una semplice retta. Al contrario l'operatore logico XOR **non è linearmente separabile**: sono necessari due iperpiani per separarne i punti appartenenti a classi diverse. Non è quindi risolvibile dal percettrone.

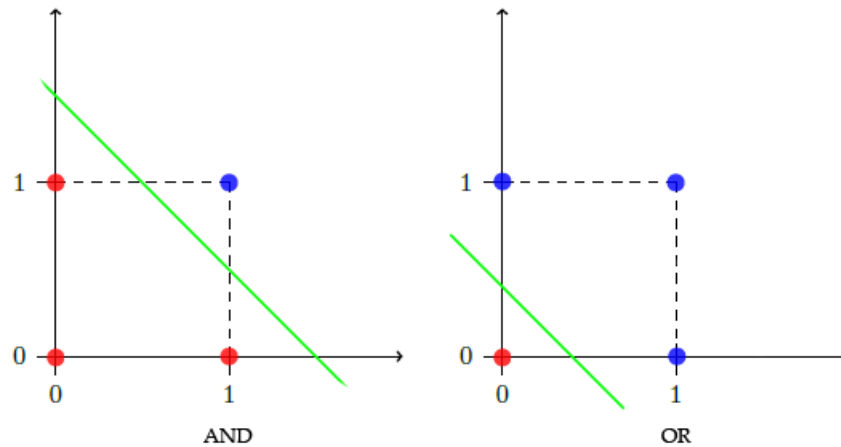


Figura 13: Operatori AND e OR linearmente separabili.

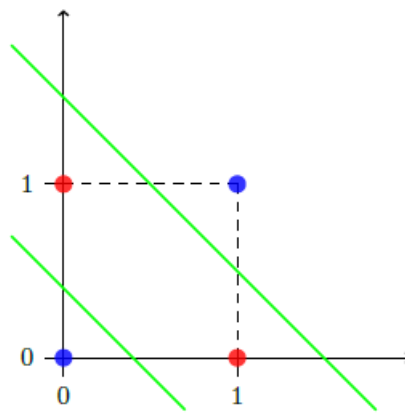


Figura 14: Operatore XOR non linearmente separabile.

A difesa di Rosenblatt, il percettrone è comunque un classificatore molto potente perché in grado di risolvere *ogni* problema linearmente separabile trovando il miglior iperpiano in grado di separare due classi. La convergenza del percettrone è *garantita* teoricamente dal *teorema di convergenza del percettrone* di N. J. Nilsson (1965) secondo cui se il problema di classificazione è linearmente separabile allora la fase di apprendimento converge ad un'appropriata impostazione dei pesi in un numero finito di passi. Nella pratica, però, **non è dato sapere a priori** se un problema sia o meno linearmente separabile. La convergenza si può comunque ottenere aggiustando i parametri del percettrone come il numero di iterazioni o il fattore di apprendimento: in questo caso, tuttavia, la convergenza è artificiale.

## 3.2 PERCETTRONE A PIÙ STRATI

È possibile sopperire alle limitazioni dei percettroni a singolo strato **aggiungendo strati di neuroni nascosti**: dal numero di strati e di neuroni per ciascun strato dipende la forma che possono assumere le regioni di decisione. Per esempio:

- le reti a singolo strato trovano regioni di decisione separabili linearmente;
- le reti a due strati individuano regioni convesse aperte o chiuse;
- le reti a tre strati identificano regioni di forma arbitraria, la cui complessità dipende dal numero di neuroni nascosti.

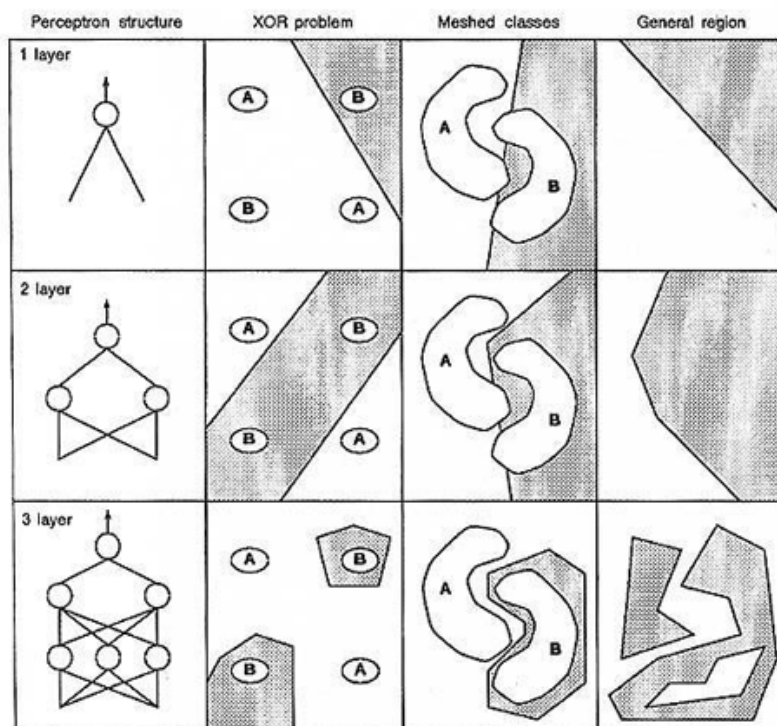


Figura 15: Percettroni e corrispondenti risultati.

Le potenzialità dei percettroni multistrato sono note da tempo ma gli algoritmi per l'apprendimento (come l'algoritmo di *backpropagation*) sono stati ideati solo recentemente (1986).

## 3.3 PROBLEMI DI CLASSIFICAZIONE E DI REGRESSIONE

Una rete neurale può essere vista come una scatola nera dotata di diversi input  $x_1, x_2, \dots, x_n$  e capace di produrre un output  $y$  (la



classe di un oggetto, per esempio).

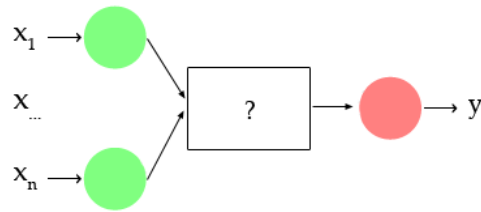


Figura 16: La rete neurale come una scatola nera.

Un problema di regressione è sostanzialmente identico se non per un particolare: l'output  $y$  non è discreto o categorico ma continuo, reale. Una seconda differenza deriva alla cosiddetto *potere d'approssimazione universale delle reti neurali* (situazioni continue). Da un punto di vista matematico una rete neurale può essere vista come una funzione  $f$  che accetta un insieme di input e produce un insieme di output:

$$f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

$$f(x_1, \dots, x_n) \in \mathbb{R}^m$$

Supponiamo d'avere una funzione arbitraria  $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$ ; sotto quali condizioni esiste una configurazione di pesi  $w$  tale da approssimare il più possibile  $g$  o, in altre parole, tale per cui la distanza tra l'output della rete e l'output di  $g$  è inferiore ad un certo  $\epsilon$ ? La differenza sostanziale tra problemi di classificazione e di regressione sta proprio qui: i problemi di regressione possono essere approssimati tramite una rete neurale a 2 strati; per approssimare le regioni di decisione di problemi di classificazione, invece, sono necessari almeno 2 strati.

## ALGORITMO DI BACKPROPAGATION

I problemi di apprendimento possono essere suddivisi in tre gruppi:

- **Apprendimento supervisionato.** Richiede l'esistenza di un esperto che etichetti correttamente *ogni* elemento del training set;
- **Apprendimento non supervisionato** o *clustering*. Gli elementi del training set non sono etichettati: si procede raggruppandoli in cluster i cui elementi appartengono alla stessa classe;
- **Apprendimento semi-supervisionato.** Solo alcuni degli elementi del training set sono etichettati da un esperto. Si tratta di un metodo nato per ridurre i costi dell'apprendimento supervisionato.

L'algoritmo di *backpropagation* permette a reti neurali multistrato di risolvere problemi di apprendimento supervisionato.

### 4.1 L'APPRENDIMENTO SUPERVISIONATO

Supponiamo d'avere una rete neurale multistrato formata da  $n$  unità d'input,  $m$  neuroni d'output e un certo numero di strati nascosti:

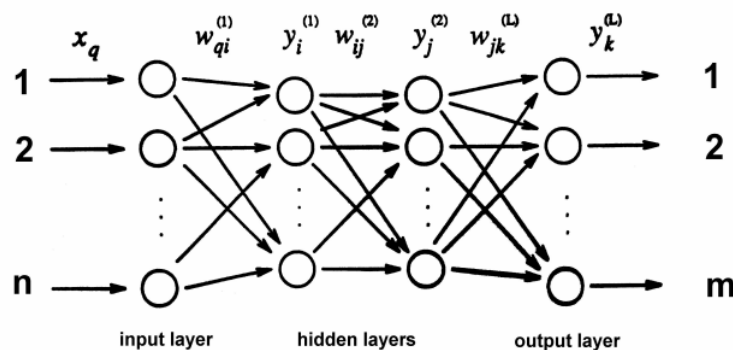


Figura 17: Rete neurale multistrato d'esempio.

Formalmente, il training set di un problema d'apprendimento supervisionato è formato da coppie di elementi  $(x_n, y_n)$ , dove cioè figurano un elemento dato in input alla rete e la corrispondente etichetta fornita dall'esperto.

$$\text{TrainingSet} = \{(x_1, y_1), \dots, (x_n, y_n)\}$$

Attraverso la fase d'apprendimento la rete aggiusta i propri pesi in modo che l'output prodotto corrisponda quanto più possibile all'output desiderato per ogni elemento del training set. Risolvere un problema d'apprendimento supervisionato significa quindi modificare il comportamento della rete – aggiustare i suoi pesi – per ottenere un output quanto più vicino a quello richiesto. Il problema di classificazione diventa quindi un **problema di ottimizzazione**: dobbiamo minimizzare una funzione che rappresenti la *qualità* dei pesi usati nella rete. La funzione scelta è l'**errore quadratico medio**:

$$E(w) = \frac{1}{2} \sum_h \sum_k (\text{out}_k^h - y_k^h)^2$$

dove:

- $\text{out}_k^h$  è l'output prodotto dal neurone  $k$  quando viene dato l'input  $h$  alla rete;
- $y_k^h$  è l'output desiderato dal neurone  $k$  quando viene dato l'output  $h$  alla rete;

La funzione  $E$  dipende solo dai pesi della rete,  $w$ ; in particolare,  $\text{out}_k^h$  dipende dai pesi mentre  $y_k^h$  ne prescinde. Dobbiamo inoltre evidenziare come minimizzare questa funzione d'errore non sia un'operazione semplice: il vettore dei pesi può infatti essere molto grande (situazioni realistiche possono presentare migliaia di pesi da aggiustare) e la funzione è inoltre altamente non-lineare (ogni neurone compie operazioni non-lineari, secondo il modello di McCulloch & Pitts).

#### 4.2 L'ALGORITMO DI BACKPROPAGATION

Gli anni '70 sono caratterizzati da un **generale disinteresse** verso le reti neurali, dettato dalla scoperta delle limitazioni dei perceptron a singolo strato, capaci di risolvere solamente problemi linearmente separabili, e dall'incapacità di addestrare reti multistrato usando la stessa tecnica impiegata per aggiornare i pesi delle reti a singolo strato. Supponiamo infatti d'avere una rete neurale a singolo strato:

La funzione  $E(w)$  è calcolata come la somma delle distanze euclidee tra l'output desiderato e l'output effettivamente prodotto per ciascun

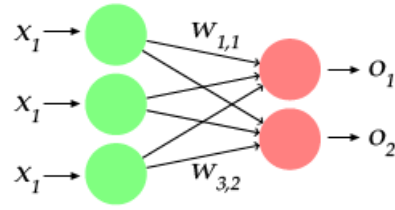


Figura 18: Una semplice rete *feedforward* a singolo strato.

elemento del training set, e permette di aggiornare il vettore dei pesi  $w = (w_{11}, w_{12}, \dots)$ . Non è possibile compiere la stessa operazione se aggiungiamo strati di neuroni nascosti: questo metodo permette di aggiornare solo i pesi relativi ai neuroni d'uscita, di cui si conosce l'output desiderato (è il secondo elemento delle coppie che costituiscono il training set), mentre nulla si sa invece in merito all'output desiderato per i neuroni nascosti.

Il problema è stato risolto da D. E. Rumelhart, G. E. Hinton e R. J. Williams, che nel 1986 hanno introdotto l'**algoritmo di backpropagation** per l'addestramento di reti neurali *feedforward* multistrato. L'algoritmo prevede di calcolare l'errore commesso da un neurone dell'ultimo strato nascosto **propagando all'indietro l'errore** calcolato sui neuroni di uscita collegati ad esso; lo stesso procedimento è poi ripetuto per tutti i neuroni del penultimo strato nascosto e così via. Come dicevamo, non è possibile risolvere un problema di ottimizzazione non-lineare così complesso attraverso metodi analitici: dobbiamo quindi appoggiarci alla tecnica della *discesa del gradiente*.

#### 4.2.1 Il gradiente

Data la funzione derivabile  $f$ , il suo gradiente  $\nabla f$  è il vettore che ha come componenti le derivate parziali di  $f$ .

$$\nabla f(x) = \left( \frac{df(x)}{dx_1}, \dots, \frac{df(x)}{dx_n} \right)$$

Per esempio, il gradiente della funzione  $E(x, y) = 2xy + y^2$  è composto da due derivate parziali, una rispetto ad  $x$  e una rispetto a  $y$ ; nel primo caso  $y$  è una costante, nel secondo lo è  $x$ .

$$\nabla E(x, y) = (2y, 2x + 2y)$$

Il gradiente può essere calcolato in qualsiasi punto dello spazio in cui è definita la funzione ed è interpretabile geometricamente come la **direzione massima di crescita di  $f$** .

#### 4.2.2 Discesa del gradiente

La *discesa del gradiente* è una **tecnica di ottimizzazione locale** che consente di trovare un minimo locale<sup>1</sup> di una data funzione  $f$ . Tale tecnica si basa sul significato geometrico del gradiente: visto che il gradiente è la direzione massima di crescita di  $f$ , per minimizzare  $f$  e trovarne un minimo locale è sufficiente scegliere punti nella **direzione opposta al gradiente**. L'algoritmo, di impostazione *greedy*, è:

1. Scegliere un punto casuale in cui è definita la funzione  $f$ ;
2. Calcolarvi il gradiente: se è diverso dal vettore nullo scegliere un nuovo punto nella direzione opposta al gradiente (con un opportuno *step size*), altrimenti l'algoritmo si arresta.

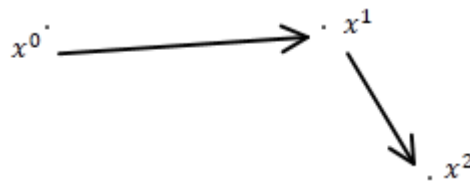


Figura 19: Esempio di discesa del gradiente.

Come vediamo, l'algoritmo è ripetuto finché il gradiente non si annulli: solo in questo caso l'algoritmo ha raggiunto un punto di stazionarietà (minimo locale o globale). In formule, se applicato alla nostra funzione obiettivo  $E(w)$ , la discesa del gradiente aggiorna il vettore dei pesi in questo modo:

$$w^{(new)} = w^{(old)} - \eta \nabla E(w^{(old)})$$

dove  $\eta$  è lo **step size**, il cui valore può teoricamente variare ad ogni iterazione; come vedremo nelle prossime pagine  $\eta$  ha una forte influenza sul comportamento dell'algoritmo (velocità di convergenza, oscillazioni, ...).  $-\mu \nabla E(w^{(old)})$  è invece l'aggiornamento dei pesi, che può essere descritto come:

<sup>1</sup> Esiste anche l'equivalente tecnica per trovare un massimo locale della funzione.

$$\Delta w^{(\text{old})} = -\eta \nabla E(w^{(\text{old})})$$

L'aggiornamento del singolo peso  $w_{ij}$  tra l'unità d'input  $j$  e il neurone d'output  $i$  è infine:

$$\Delta w_{ij} = -\eta \nabla E(w_{ij}) = -\mu \frac{dE}{dw_{ij}}$$

#### 4.3 LA BACKPROPAGATION DELL'ERRORE

L'algoritmo di backpropagation è un algoritmo locale il cui scopo è quello di calcolare  $\Delta w^{(\text{old})}$ . L'algoritmo è strutturato in **due passi**:

- **forward pass**: l'input dato alla rete è propagato in avanti, strato dopo strato, fino al livello di uscita dove viene calcolato l'errore commesso per ciascun neurone d'output;
- **backward pass**: l'errore è propagato all'indietro e i pesi sono aggiornati in maniera appropriata.

Supponiamo di avere la seguente rete:

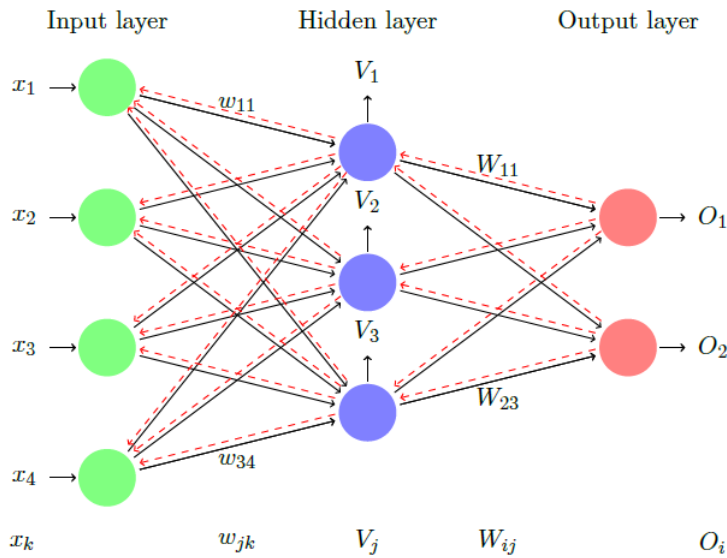


Figura 20: Backpropagation: le linee nere indicano il segnale propagato in avanti, mentre quelle rosse indicano l'errore propagato all'indietro

Indichiamo con  $k$  le unità d'input, con  $i$  le unità d'output, con  $j$  le unità appartenenti allo strato nascosto. I pesi che collegano lo strato

d'input allo strato nascosto sono indicati con  $w_{jk}$  mentre i pesi che collegano lo strato nascosto allo strato d'output sono  $W_{ij}$ . L'input ricevuto dal j-esimo neurone dello strato nascosto è il seguente:

$$h_j^\mu = \sum_k w_{jk} x_k^\mu$$

Si tratta cioè della somma pesata dei valori d'input per i pesi che collegano tali input al neurone j. L'output prodotto dal neurone j è invece:

$$v_j^\mu = g(h_j^\mu) = g\left(\sum_k w_{jk} x_k^\mu\right)$$

dove g è la **funzione di attivazione** non-lineare del neurone j.

#### 4.3.1 Aggiornamento dei pesi strato nascosto - strato d'output

Vogliamo ora calcolare l'aggiornamento dei pesi che congiungono lo strato d'output e lo strato nascosto della rete,  $\Delta W_{ij}$ . Come sappiamo,  $\Delta W_{ij}$  è uguale alla derivata della funzione E rispetto al peso  $W_{ij}$ :

$$\Delta W_{ij} = -\eta \frac{dE}{dW_{ij}}$$

Per prima cosa ricordiamo la **regola della catena** secondo cui, date due funzioni f e g, la derivata della loro composizione è:

$$df(g(x)) = f'(g(x))g'(x)$$

La derivata della funzione  $E(w)$  applica questa regola:

$$= -\eta \frac{d\left(\frac{1}{2} \sum_\mu \sum_k (O_k^\mu - y_k^\mu)^2\right)}{dW_{ij}} = \eta \sum_\mu \sum_k (O_k^\mu - y_k^\mu) \frac{d(O_k^\mu)}{dW_{ij}}$$

In questa formula k rappresenta i neuroni d'output; la sommatoria su k può quindi essere eliminata visto che siamo interessati solamente all'i-esimo neurone d'output:

$$= -\eta \sum_\mu (O_i^\mu - y_i^\mu) \frac{d(O_i^\mu)}{dW_{ij}}$$

Deriviamo  $\frac{d(O_i^\mu)}{dW_{ij}}$ . L'output dell'i-esimo neurone  $i$  è:

$$O_i^\mu = g(h_i^\mu)$$

Riapplichiamo la regola della catena e otteniamo:

$$= -\eta \sum_{\mu} (O_i^\mu - y_i^\mu) g'(h_i^\mu) \frac{dh_i^\mu}{dW_{ij}}$$

Dobbiamo ora calcolare  $\frac{dh_i^\mu}{dW_{ij}}$ .  $h_i^\mu$  è il net input dell'i-esimo neurone:

$$h_i^\mu = \sum_l V_l^\mu W_{il}$$

Di nuovo, siamo interessati solamente al  $j$ -esimo neurone: la sommatoria può essere eliminata per  $l = j$ . Ma la derivata di  $V_j^\mu W_{ij}$  è semplicemente  $V_j^\mu$ , da cui:

$$\Delta W_{ij} = -\eta \sum_{\mu} (y_i^\mu - O_i^\mu) g'(h_i^\mu) V_j^\mu$$

dove  $(y_i^\mu - O_i^\mu) g'(h_i^\mu)$  è l'errore del neurone  $i$ -esimo, noto come **gradiente locale**.

#### 4.3.2 Aggiornamento dei pesi strato input - strato nascosto

L'aggiornamento del peso tra lo strato d'input e lo strato nascosto,  $\Delta w_{jk}$ , non è concettualmente diverso da quanto appena visto ma presenta qualche complicazione. Prima di tutto individuiamo il significato della notazione usata:  $j$  indica un neurone dello strato nascosto,  $k$  indica un'unità input;  $i$  rappresenta sempre un neurone dello strato d'output.

$$\Delta w_{jk} = -\eta \frac{dE}{dw_{jk}}$$

Come sopra, deriviamo la funzione d'errore  $E(w)$  applicando la regola della catena:

$$-\eta \sum_i \sum_{\mu} (O_i^\mu - y_i^\mu) \frac{dO_i^\mu}{dw_{jk}}$$



A differenza di quanto visto precedentemente **non possiamo eliminare la sommatoria** su  $i$  perché l'output di ogni neurone dello strato d'output dipende indirettamente da  $w_{jk}$ . Occupiamoci quindi di  $\frac{dO_i^\mu}{dw_{jk}}$ . Come prima, sappiamo che  $O_i^\mu$  si definisce come  $g(h_i^\mu)$ , da cui:

$$-\eta \sum_i \sum_\mu (O_i^\mu - y_i^\mu) g'(h_i^\mu) \frac{dh_i^\mu}{dw_{jk}}$$

Calcoliamo ora  $\frac{dh_i^\mu}{dw_{jk}}$ .  $h_i^\mu$  è il net input ricevuto dall' $i$ -esimo neurone, e perciò:

$$\frac{dh_i^\mu}{dw_{jk}} = \frac{\sum_l V_l^\mu W_{il}}{dw_{jk}}$$

La sommatoria può essere eliminata dato che siamo interessati solo al caso in cui  $l = j$ . Spostiamo la costante  $W_{ij}$ :

$$\frac{\sum_l V_l^\mu W_{il}}{dw_{jk}} = \frac{dV_j^\mu W_{ij}}{dw_{jk}} = W_{ij} \frac{dV_j^\mu}{dw_{jk}}$$

Occupiamoci quindi di  $\frac{dV_j^\mu}{dw_{jk}}$ .  $V_j^\mu$  è definito come l'output prodotto dal neurone  $j$ -esimo dato l'input  $\mu$ ; formalmente:

$$V_j^\mu = g(h_j^\mu)$$

Perciò la derivata di  $V_j^\mu$  è di nuovo composta:

$$\frac{dV_j^\mu}{dw_{jk}} = g'(h_j^\mu) \frac{dh_j^\mu}{dw_{jk}}$$

Per quanto riguarda invece  $\frac{dh_j^\mu}{dw_{jk}}$ , dato che  $h_j^\mu = \sum_m w_{jm} x_m^\mu$  allora la nostra derivata diventa:

$$\frac{dh_j^\mu}{dw_{jk}} = \frac{d \sum_m w_{jm} x_m^\mu}{dw_{jk}}$$

La sommatoria può essere tolta: ci interessa lavorare solo sul caso in cui  $m = k$ .

$$\frac{dh_j^\mu}{dw_{jk}} = \frac{dw_{jk} x_k^\mu}{dw_{jk}}$$

Ora la derivata è banale: semplicemente  $x_k^\mu$ . Riassumendo abbiamo:

$$\Delta w_{jk} = -\eta \sum_i \sum_\mu (O_i^\mu - y_i^\mu) g'(h_i^\mu) W_{ij} g'(h_j^\mu) x_k^\mu$$

Facciamo ordine chiamando:

- $\delta_i^\mu = (O_i^\mu - y_i^\mu) g'(h_i^\mu)$
- $\delta_j^\mu = \sum_i \delta_i^\mu W_{ij} * g'(h_j^\mu)$ ; è il gradiente locale del neurone nascosto j.

Ne deriva che l'aggiornamento del peso tra j e k è:

$$\Delta w_{jk} = \eta \sum_\mu \delta_j^\mu x_k^\mu$$

Notare che  $\sum_i \delta_i^\mu W_{ij}$  è l'**errore medio** sullo strato di output causato dal j-esimo neurone dello strato nascosto.

#### 4.3.3 Osservazioni e caso generale

In generale, l'aggiornamento dei pesi **non necessita di informazioni globali ma solo locali**; questo comportamento ha senso visto che stiamo simulando il comportamento del cervello umano: che senso avrebbe per la singola connessione tra due neuroni aver bisogno di dati proveniente da altre aree del cervello? Le formule d'aggiornamento di peso calcolate finora presentano inoltre una **struttura molto simile**, che possiamo riassumere come segue. Dati due neuroni p e q appartenenti a due strati diversi, l'aggiornamento del corrispondente peso  $w_{pq}$  è:

$$\Delta w_{pq} = \eta \sum_\mu \delta_p^\mu V_q^\mu$$

...cioè l'output prodotto dal neurone q moltiplicato per  $\delta_p^\mu$ , che diventa a seconda dei casi:

$$\delta_p^\mu = \begin{cases} g'(h_p^\mu)(O_p^\mu - y_p^\mu) & \text{se p è un neurone di output} \\ g'(h_p^\mu) \sum_i \delta_i^\mu W_{ip} & \text{altrimenti} \end{cases}$$

## 4.3.4 Implementazione offline e online

L'implementazione dell'algoritmo di backpropagation può essere sia *online* che *offline*:

- **Offline:** per aggiornare un particolare peso la rete usa tutti gli elementi del training set. Non è molto pratico perché necessita che l'intero training set sia analizzato prima procedere all'aggiornamento.
- **Online:** per aggiornare un particolare peso la rete prende un esempio casuale dal training set, esegue il *forward pass* e il *backward pass*, quindi aggiorna i pesi immediatamente; la procedura viene ripetuta fino alla convergenza o fino a che non si verificano determinate condizioni. Si tratta di un metodo più semplice, pratico e logico, e che in genere fornisce risultati migliori<sup>2</sup>.

In pratica, la differenza sta tutta nel modo in cui calcoliamo l'aggiornamento dei pesi. In formule questa differenza è evidente:

$$\Delta w_{pq} = \begin{cases} \eta \sum_{\mu} \delta_p^{\mu} V_p^{\mu} & \text{offline} \\ \eta \delta_p^{\mu} V_p^{\mu} & \text{online} \end{cases}$$

## 4.4 L'ALGORITMO ONLINE DI BACKPROPAGATION

Consideriamo una generica rete neurale con  $M$  strati in cui:

- $V_i^m$  è l'output del neurone  $i$ -esimo del livello  $m$ ;
- $w_{ij}^m$  è il peso della connessione tra il  $j$ -esimo neurone dello strato  $m-1$  e l' $i$ -esimo neurone dello strato  $m$ .

Con  $m = 0, \dots, M$  l'algoritmo è il seguente:

1. Inizializzo i pesi della rete a (piccoli) valori casuali;
2. Scelgo un esempio  $x^m$  dal training set e lo passo allo strato d'input;

<sup>2</sup> Non si tratta di una contraddizione. In base al punto di partenza, l'algoritmo arriva sempre ad un minimo locale; può trattarsi però del peggior minimo locale. Usando l'approccio online inseriamo del *rumore* nell'algoritmo, che può essere ottimo per evitare di rimanere bloccati in un minimo locale inadatto.

3. Propago il segnale in avanti:

$$V_i^m = g(h_i^m) = g\left(\sum_j w_{ij}^m V_j^{m-1}\right)$$

4. Calcolo gli errori per lo strato d'output:

$$\delta_i^M = g'(h_i^M)(y_i - V_i^M)$$

5. Calcolo gli errori per tutti gli strati rimanenti:

$$\delta_i^{m-1} = g'(h_i^{m-1}) \sum_j \delta_j^m w_{ji}^m$$

6. Aggiorno i pesi:

$$w_{ij}^{\text{new}} = w_{ij}^{\text{old}} - \Delta w_{ij}^{\text{old}}$$

$$\text{dove } \Delta w_{ij}^{\text{old}} = \eta \delta_i^m V_j^{m-1}.$$

7. Torna al passo 2 fino alla convergenza.

L'algoritmo può terminare quando la differenza tra i nuovi e i vecchi pesi è inferiore ad un certo  $\epsilon$  oppure dopo un certo numero di iterazioni (se la condizione precedente non è rispettata).

#### 4.4.1 Applicazione dell'algoritmo di Backpropagation

Una celebre applicazione dell'algoritmo appena visto è la rete neurale costruita per imparare a parlare chiamata **NETtalk**: dato un training set contenente testi e relativa pronuncia fonetica, la rete riesce a leggere questi testi a seguito della fase d'addestramento (*text-to-speech synthesis*).

### 4.5 IL MOMENTO

La scelta del fattore di apprendimento  $\eta$  influenza il comportamento dell'algoritmo. Per capire perché possiamo vedere due esempi:

- Se scegliamo valori troppo piccoli la discesa del gradiente sarà lenta;
- Se scegliamo valori troppo alti la discesa del gradiente rischia d'avere un comportamento oscillatorio e non convergere mai.

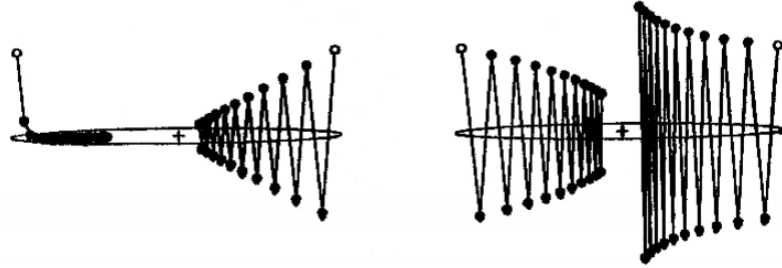


Figura 21: Convergenza lenta (a sinistra) e comportamenti oscillatori.

Un metodo semplice per incrementare  $\eta$  senza il rischio di rendere l'algoritmo instabile consiste nel modificare la regola per l'update dei pesi inserendo un nuovo termine, il *momento*, proporzionale alla precedente variazione dei pesi. La regola di aggiornamento diventa:

$$\Delta w_{pq}^{\text{NEW}} = \alpha \Delta w_{pq}^{\text{OLD}} - \eta \frac{dE}{dw_{pq}}$$

In questo modo ci è possibile usare valori alti di  $\eta$  evitando fenomeni d'oscillazione. In genere **si sceglie**  $\alpha = 0.9$  e  $\eta = 0.5$ .

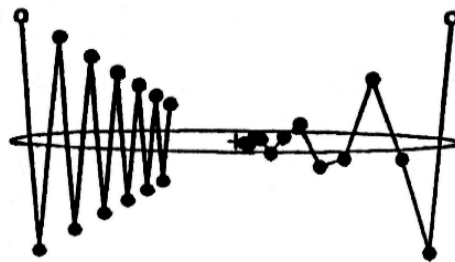


Figura 22: Sulla sinistra non c'è il momento, sulla destra  $\alpha = 0.5$

## 4.6 MINIMI LOCALI

L'algoritmo di backpropagation non è sempre in grado di trovare il minimo globale. Il problema risiede nell'esistenza di *buoni* e *cattivi* punti di minimo.

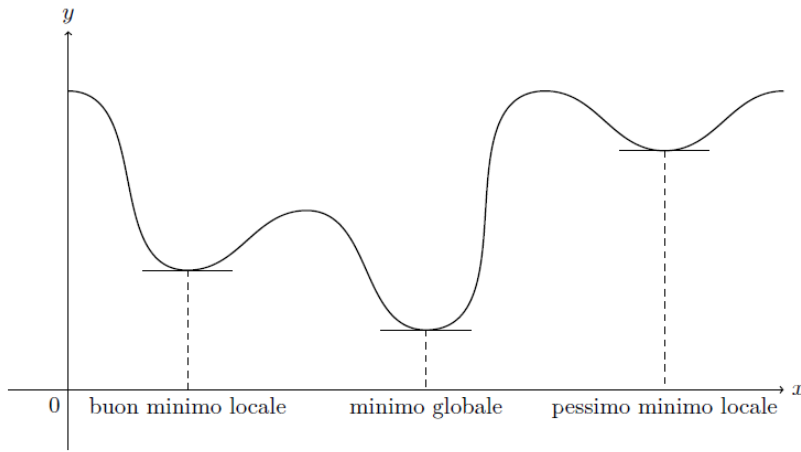


Figura 23: Vari esempi di minimi locali.

Per evitare il problema è importante **scegliere una configurazione iniziale adeguata dei pesi**. Nella pratica si utilizza la seguente euristica per impostare i pesi iniziali:

$$w_{ij} = \frac{1}{\sqrt{k_i}}$$

dove  $k_i$  è il numero di unità entranti nell'unità  $i$  (*fan-in* di  $i$ ). In gener, si consiglia di scegliere **pesi iniziali casuali e piccoli**. Il motivo è presto detto: se consideriamo la regola d'aggiornamento di backpropagation,

$$\Delta w_{ij} = \eta \delta_i^m V_j^{m-1}$$

$\delta_i^{m-1} = g'(h_i^{m-1}) \sum_j \delta_j^m w_{ji}^m$ . Ora, se scegliamo dei pesi molto grandi ne deriva che il net input  $h_i^{m-1}$  sarà a sua volta grande. La funzione  $g$  diventerà così quasi **parallela all'asse  $x$** : la sua derivata e  $\delta$  saranno vicine allo 0. L'aggiornamento del peso sarà quindi **quasi nullo**.



## COSTRUZIONE DI UNA RETE E APPRENDIMENTO

---

Quando si intende costruire una rete neurale per risolvere un particolare problema ci sono diverse questioni da affrontare:

- Di **quanti strati** dev'essere composta la rete? Sappiamo che per problemi di classificazione sono sufficienti 2 strati nascosti e che per problemi di regressione 1 strato è sufficiente – a condizione però che il numero di neuroni di questi strati sia sufficientemente elevato.
- Quante **unità per ciascun strato**? Esistono diverse regole per determinare il numero di neuroni, ma non tutte sono praticabili. Per esempio, è dimostrabile che per risolvere correttamente problemi di classificazione è sufficiente avere **tanti neuroni quante sono gli esempi del training set**; si tratta però di una soluzione poco pratica, visto che un training set può contenere migliaia di elementi.
- Fino a che punto ci interessa **rappresentare il problema**? Supponiamo, ad esempio, d'avere un problema di classificazione su 4 features e 8 classi. Siamo sicuri di dover avere tanti neuroni d'output quante sono le classi e tante unità d'input quante sono le features, o possiamo ridurne il numero? Per esempio, potremmo impostare che ogni neurone d'output generi una stringa binaria: 000 = classe 1, 001 = classe 2, .... In questo modo sarebbero sufficienti solo 3 neuroni d'output.

Per rispondere a queste domande dobbiamo introdurre il concetto di *generalizzazione*.

### 5.1 GENERALIZZAZIONE

Con il termine *generalizzazione* si intende la capacità della rete di fornire risposte corrette ad esempi tratti dal *test set*, cioè non incontrati durante la fase di addestramento<sup>1</sup>. La **dimensione del training set**

---

<sup>1</sup> Si assume che training e test set siano formati da elementi estratti dalla stessa popolazione.



è fondamentale affinché la rete compia una buona generalizzazione: un training set troppo grande e un conseguente numero troppo alto di pesi possono portare a casi di *overfitting* in cui la rete memorizza non solo il training set ma anche tutto il **rumore** (classificazioni errate, ...) contenuto in esso. La rete diventa così **troppo rigida e incapace di generalizzare**: probabilmente si comporterà perfettamente con il training set ma darà risultati mediocri con il test set. Esistono anche casi di *underfitting* dove la rete non impara sufficientemente bene i dati del training set perché, ad esempio, dotata di pochi pesi: per esempio, un perceptrone che tenta di riprodurre l'operatore logico XOR sotto-impara i dati.

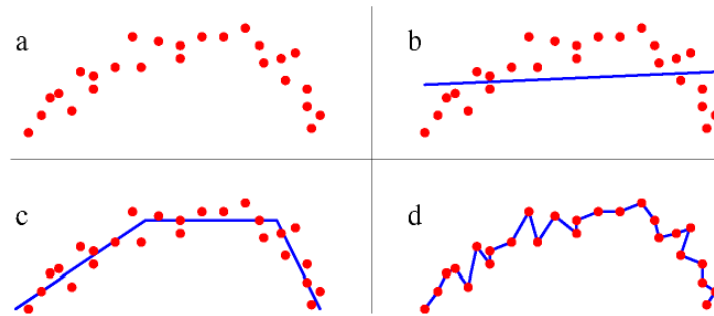


Figura 24: (a) dati del training set, (b) underfitting, (c) una buona stima sui dati, (d) overfitting: la curva di apprendimento è perfettamente disposta sul training set.

Si noti peraltro la **contraddizione**: da un lato vogliamo che la rete classifichi correttamente gli esempi del training set minimizzando la funzione d'errore, dall'altro che sia performante anche quando è posta di fronte ad esempi mai visti prima. Per riassumere, una rete troppo piccola impara poco, mentre una rete troppo grande impara molto, ma non generalizza abbastanza. In entrambi i casi si rischia di non riuscire a risolvere il problema: è necessario **trovare un giusto compromesso**.

La capacità di generalizzazione è influenzata da **tre fattori principali**: le dimensioni del training set, l'architettura della rete neurale e la complessità del problema. Dal momento che non si ha alcun controllo sulla complessità del problema è possibile affrontare la generalizzazione sotto due punti di vista:

- Si fissa l'architettura della rete e si determina la dimensione del training set ottimale per una buona generalizzazione;
- Si fissa la dimensione del training set e si determina la migliore architettura di rete per una buona generalizzazione.

## 5.2 DETERMINARE IL TRAINING SET

Supponiamo di voler risolvere un problema di classificazione utilizzando una rete neurale. Prima di tutto vogliamo fissare uno strumento per valutare la capacità di generalizzare della rete, cioè una misura degli errori commessi:

- **Misclassification error**, cioè il rapporto tra il numero di classificazioni errate e il numero di dati del training set;
- **Errore quadratico medio**, che abbiamo già visto in precedenza:

$$E = \frac{1}{2} \sum_{\mu} \sum_k (O_k^{\mu} - y_k^{\mu})^2$$

Possiamo poi passare alla costruzione di training e test set. Il modo più semplice per determinare il contenuto di entrambi è **dividere l'intero data set** in due parti: 60 – 70% dei dati per il training set, il resto per il test set<sup>2</sup>. Esistono però tecniche più sofisticate per gestire la definizione del training set senza dover tagliare l'intero data set in due. Una delle più popolari è la *cross-validation*.

## 5.3 CROSS-VALIDATION

Cross-validation è una tecnica usata per **selezionare il miglior modello** tra quelli utilizzabili per risolvere un problema. La sua versione più diffusa è la *k-fold* cross-validation, che consiste nel suddividere l'intero data set in  $k$  diversi training set. La rete viene poi addestrata su  $k - 1$  insiemi, che costituiscono il training set, e testata sul rimanente  $k$ -esimo insieme. Per ciascuna iterazione è possibile ottenere il corrispondente errore quadratico medio,  $E_i$ , e calcolare quindi l'errore medio:

$$E = \frac{1}{k} \sum_i E_i$$

La strategia migliore, ma applicabile raramente a causa delle enormi dimensioni raggiunte dal training set, consiste nell'utilizzare un training set formato da  $k = N - 1$  elementi, dove  $N$  è la cardinalità dell'insieme dei dati. In questo modo l'unico restante elemento viene usato come test set.

<sup>2</sup> Esistono situazioni in cui il numero di dati disponibili è molto basso: considerarne il 70% solo per il training set rende il test set quasi vuoto.

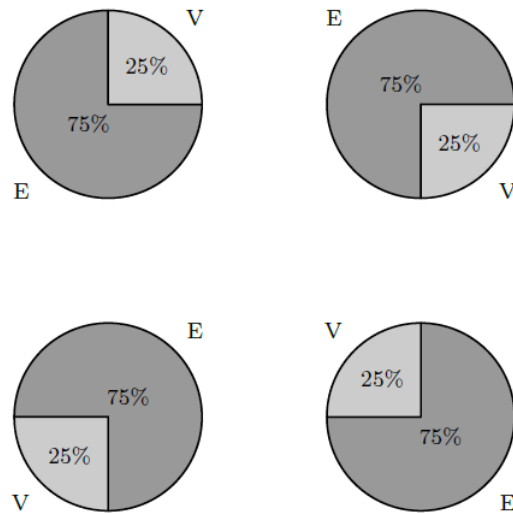


Figura 25: Esempio di cross validation.

#### 5.4 METODO DI TRAINING EARLY-STOPPING

Se l'obiettivo è ottenere una buona generalizzazione è molto difficile decidere quand'è il momento di bloccare la fase d'addestramento: se non si ferma l'addestramento al punto giusto c'è il rischio di overfitting. Per evitare questo problema siamo costretti ad adottare un **compromesso**. Se monitoriamo il comportamento della rete nel tempo notiamo il seguente comportamento al passare delle epoche<sup>3</sup>:

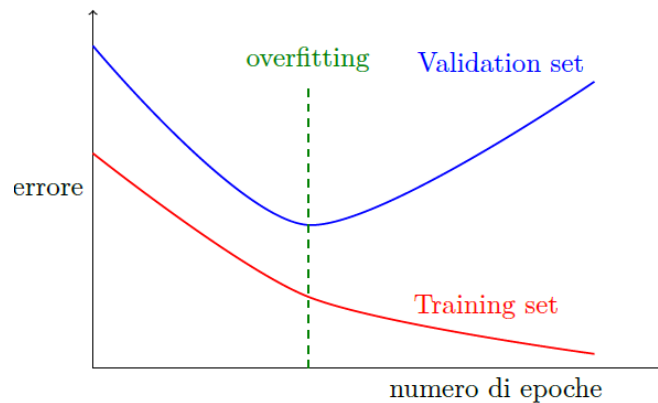


Figura 26: Early stopping: fermare la fase d'addestramento prima dell'overfitting.

La curva rossa mostra l'andamento dell'errore nel classificare il training set mentre la curva blu mostra l'errore nel classificare il test

<sup>3</sup> Una epoca è il tempo necessario alla rete per esaminare l'intero training set.

set. Se l'errore sul test set aumenta mentre l'errore sul training set diminuisce allora siamo in presenza di un possibile caso di overfitting. Ne deriva che per evitare overfitting possiamo **fermare l'algoritmo di training** prima che la performance della rete si deteriori, cioè **prima di superare la linea verde**. Il processo di addestramento early-stopping è il seguente:

- dopo un periodo di addestramento sul training set si calcola l'errore di validazione per ogni esempio del test set;
- quando la fase di test è completa, si riprende la fase di addestramento per un altro periodo.

Notare che potremmo essere tentati di proseguire l'addestramento anche oltre il minimo della curva del *test set*: in realtà ciò che la rete apprende dopo quel punto è il **rumore contenuto nei dati del training set**. L'euristica suggerisce quindi di **fermare l'addestramento in corrispondenza del minimo** della curva relativa al test set.

## 5.5 TECNICHE DI PRUNING

La capacità di generalizzazione di una rete è influenzata dalla sua dimensione, ovvero dal **numero di neuroni nascosti**: con una rete troppo piccola si rischia di non riuscire a risolvere il problema mentre con una troppo grande si rischia di apprendere il rumore deteriorando la capacità di generalizzare. Per scegliere la dimensione corretta abbiamo due strategie:

- **pruning**: si parte da una rete sovradimensionata per poi ridurla eliminando connessioni o neuroni.
- **growing**: si parte da una rete piccola per poi espanderla.

A sua volta, gli algoritmi di pruning si dividono in due classi:

- Algoritmi che funzionano *durante* la fase di training, forzando alcuni pesi a 0;
- Algoritmi che funzionano *dopo* la fase di training. In questo caso non c'è interferenza tra l'algoritmo di pruning e l'algoritmo di training.

In generale l'approccio di pruning è il più adottato nonostante richieda più tempo (si devono addestrare più unità): si è infatti rivelato

particolarmente efficiente, evitando per esempio minimi locali inadatti all'algoritmo di *backpropagation*; è inoltre indipendente dall'algoritmo di addestramento utilizzato.

Supponiamo ora di avere una rete sovradimensionata composta da tre unità d'input, 3 neuroni d'output e un certo numero di neuroni nascosti.

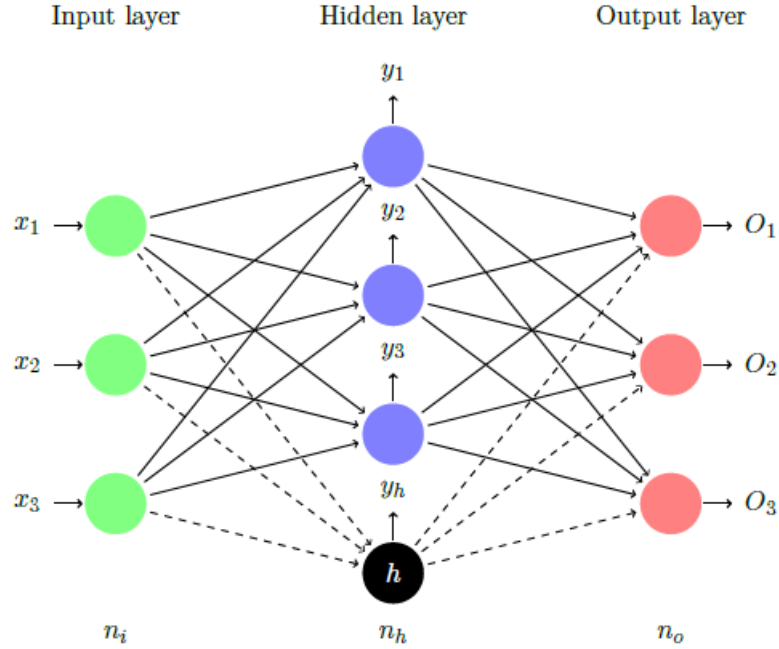


Figura 27: Una rete sovradimensionata su cui applicare il pruning.

Per rimuovere il neurone  $n_h$  e le sue connessioni in entrata e in uscita dobbiamo aggiustare i rimanenti pesi della rete in modo che le prestazioni della rete pre e post eliminazioni rimangano costanti. L'opzione banale, che prevede di rimuovere  $n_h$  e riaddestrare la rete, non garantisce d'ottenere una rete performante tanto quanto la precedente. La strategia corretta è quella **aggiungere un certo  $\delta$  a ciascuno dei pesi rimanenti** in modo che il net input dei neuroni d'output *prima* della rimozione sia identico al net input dei neuroni d'output *dopo* la rimozione:

$$\text{NetInput}_i^\mu(\text{con } h) = \text{NetInput}_i^\mu(\text{senza } h)$$

Riutilizziamo la notazione vista precedentemente:  $i$  è l'indice che scorre i neuroni d'output,  $j$  è l'indice che scorre i neuroni nascosti.  $w_{ij}$  è il generico peso tra strato nascosto e strato d'output;  $y_j^\mu$  è l'output prodotto dal  $j$ -esimo neurone nascosto quando è fornito il pattern  $\mu$  alla rete. L'uguaglianza tra net input diventa quindi:

$$\sum_{j=1}^{n_h} w_{ij} y_j^\mu = \sum_{j=1, j \neq h}^{n_h} (w_{ij} + \delta_{ij}) y_j^\mu$$

dove  $\delta_{ij}$  è il valore di cui parlavamo precedentemente e che dobbiamo sommare ai pesi per riequilibrarli dopo la rimozione di  $n_h$ . Riscriviamo la precedente formula in questo modo:

$$\sum_{j=1}^{n_h} w_{ij} y_j^\mu = \sum_{j=1, j \neq h}^{n_h} w_{ij} y_j^\mu + \sum_{j=1, j \neq h}^{n_h} \delta_{ij} y_j^\mu$$

L'unica differenza tra le due sommatorie è che la prima contiene  $n_h$ , la seconda non lo contiene. Perciò:

$$w_{ih} y_h^\mu = \sum_{j=1, j \neq h}^{n_h} \delta_{ij} y_j^\mu$$

Si tratta di un sistema lineare sparso del tipo  $Ax = b_h$ , dove  $A$  è una matrice contenente gli output dei neuroni mentre  $b$  è un vettore contenente invece  $w_{ih} y_h^\mu$ . Il problema di questo sistema lineare è che non è detto che esista un  $\delta$  che lo soddisfi; si può però **trasformare il problema in un problema di minimo**: cerchiamo cioè il valore  $x$  che minimizza le distanze tra  $A$  e  $b$ .

$$\min_x \|Ax - b\|$$

In questo modo anche se il sistema lineare ha una soluzione sarà possibile ottenerne un'approssimazione. L'idea generale per poi risolvere questo sistema è la seguente:

- Iniziamo con un valore casuale  $x_0$  e calcoliamo il suo residuo, la distanza cioè tra  $x_0$  e  $b$ :  $r_0 = \|Ax_0 - b\|$ ;
- Si seleziona un nuovo punto,  $x_1$  il cui residuo è minore o uguale al residuo precedente:  $r_1 = \|Ax_1 - b\|$ , con  $r_1 \leq r_0$
- Si prosegue in questo modo fino a minimizzare questi residui, convergendo a minimizzare  $x$  di conseguenza.

A conferma di ciò, la curva con tutti i residui mostra un andamento discendente.

$$x_0 : r_0 = \|Ax_0 - b\| \quad x_1 : r_1 = \|Ax_1 - b\| \quad r_1 \leq r_0 \quad \dots$$

5.5.1 *Quali neuroni rimuovere?*

Come determiniamo quali sono i neuroni nascosti da rimuovere? La **scelta ideale** dovrebbe ricadere sul neurone con il **minor residuo finale** calcolato sul sistema corrispondente, cioè sul neurone che ha il minore impatto sul comportamento della rete. In pratica, tuttavia, calcolare il residuo minimo finale ha un costo computazionale elevato a causa del numero di sistemi da risolvere, uno per ogni neurone nascosto; si sceglie quindi la via del compromesso, eliminando l'unità con il minor residuo iniziale. Dato che il punto di partenza scelto solitamente è il **vettore nullo**, allora il residuo iniziale è  $\|b\|$ : la regola quindi è **scegliere il neurone con il minimo  $\|b\|$** . Naturalmente questo criterio non garantisce la soluzione ottimale, perché non è detto che esista una corrispondenza tra residuo iniziale minimo e residuo finale minimo; tuttavia, nella pratica, si è dimostrato efficace.

Riassumendo, data una rete sovradimensionata i passi dell'algoritmo di pruning sono:

1. Trovare l'unità  $h$  con  $\|b\|$  minimo.
2. Risolvere il sistema corrispondente (aggiustare i pesi).
3. Rimuovere l'unità  $h$ .
4. Torna al punto 1 se la differenza tra le performance della rete è entro un certo  $\epsilon$ , altrimenti scarta la rete ridotta.

$$\text{Perf}(\text{pruned}) - \text{Perf}(\text{originale}) < \epsilon$$

## RETI DI HOPFIELD

Finora abbiamo studiato reti neurali statiche, simili a scatole nere: dato un input esse semplicemente producono un output. Esistono tuttavia reti il cui output è propagato all'indietro come nuovo input, dinamicamente: si tratta delle reti *feedback* o ricorrenti. Il primo modello di rete feedback è stato presentato nei primi anni '80 da J. J. Hopfield.

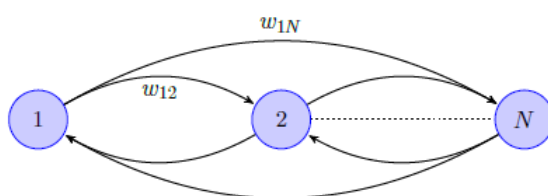


Figura 28: Esempio di rete ricorrente

Le reti di Hopfield nascono dal tentativo di costruire una rete neurale che **simuli il funzionamento della memoria associativa umana**: data cioè un'informazione parziale (una porzione d'immagine, una parziale sequenza di bit, ...) la rete è in grado di recuperarne la parte restante. Dal punto di vista tecnico le reti di Hopfield hanno queste caratteristiche:

- **ricorrenti a singolo strato**, cioè dotate di un singolo strato di neuroni d'output;
- tutti i neuroni sono **interconnessi tra loro** (escludiamo il caso in cui un neurone sia connesso con se stesso);
- **simmetriche**: dati due neuroni  $a$  e  $b$ , allora  $a \rightarrow b$ ,  $b \rightarrow a$  e le due connessioni hanno lo stesso peso.

Le reti di Hopfield possono servirsi di neuroni discreti o continui, da cui dipendono **due diversi modelli**:

- Modello discreto, con unità binarie (neuroni standard di McCulloch & Pitts);
- Modello continuo, con unità continue.



## 6.1 RETI DI HOPFIELD: MODELLO DISCRETO

Si tratta del primo modello, introdotto nel 1982. La rete è formata da neuroni standard di McCulloch e Pitts. Descriviamo prima di tutto il **comportamento del singolo neurone**:

- **Input.** L'input dell' $i$ -esimo neurone,  $H_i$ , equivale al classico net input dei neuroni di McCulloch e Pitts più un **elemento esterno**,  $I_i$ :

$$H_i = \sum_{j \neq i} w_{ij} V_j^\mu + I_i$$

L'input equivale cioè alla sommatoria dei pesi in entrata a  $i$  per l'output dei neuroni collegati a  $i$ , più un elemento esterno caratteristico di  $i$ . Notare che  $j \neq i$  reitera l'idea che un neurone non possa essere connesso a se stesso.

- **Output.** Trattandosi di un'unità binaria discreta l'output  $V_i$  dell' $i$ -esimo neurone è binario, indicato con *high* o *low*. In genere si scelgono i valori  $V_i^h = +1$  e  $V_i^l = -1$ .
- **Funzione d'attivazione.** Si usa la **funzione segno** applicata all'input del neurone:

$$V_i = \text{sign}(H_i) = \begin{cases} -1 & H_i = \sum_{j \neq i} w_{ij} V_j^\mu + I_i < 0 \\ +1 & H_i = \sum_{j \neq i} w_{ij} V_j^\mu + I_i \geq 0 \end{cases}$$

Passiamo quindi al **comportamento collettivo** dei neuroni, cioè al modo in cui aggiornano il proprio stato. Esistono due modalità, entrambe basate sull'esistenza del tempo:

- **Aggiornamento asincrono:** ad ogni unità di tempo si aggiorna un neurone alla volta, indipendentemente dai rimanenti. Si tratta della modalità più diffusa e realistica, ed è scelta dalle reti di Hopfield. L'unità da aggiornare per ogni unità di tempo è scelta **casualmente** oppure attraverso l'**assegnazione ad ogni neurone di una probabilità** costante ad ogni istante.
- **Aggiornamento sincrono:** basato sull'assunzione dell'esistenza di un *time clock*, tutte le unità si aggiornano in parallelo, nello stesso istante. Soluzione poco conveniente e irrealistica.

### 6.1.1 La rete neurale come una traiettoria

Supponiamo di avere una rete di Hopfield di  $n$  neuroni; poniamo di raccogliere gli stati di tutti i neuroni al tempo  $t$ : otteniamo un vettore di  $n$  valori,  $V(t) = (V_1(t), \dots, V_n(t))^T$ , che possiamo rappresentare come un punto in uno spazio  $n$ -dimensionale<sup>1</sup>. L'evoluzione della rete di Hopfield può quindi essere rappresentata tramite una traiettoria da punto a punto, che evolve fino al raggiungimento di un **punto stazionario** – detto anche *attrattore* – in cui  $V(t+1) = V(t)$  (all'unità di tempo successiva non avremo nessun nuovo vettore: l'evoluzione della rete si stabilizza).

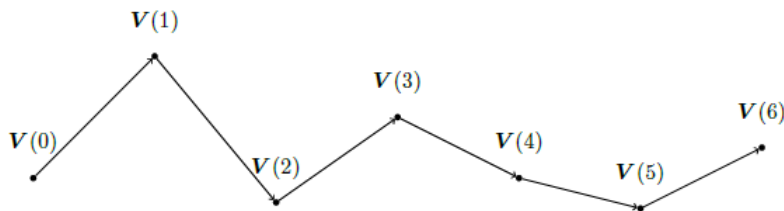


Figura 29: Un esempio di traiettoria discreta per una rete di Hopfield.

Non sempre la rete è in grado di raggiungere un punto stazionario: nella pratica abbiamo infatti **comportamenti convergenti**...

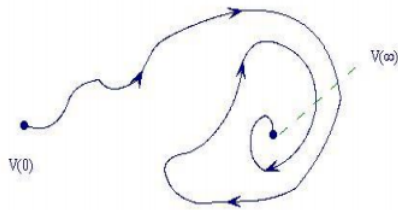


Figura 30: Comportamento convergente di una traiettoria.

...**ciclici**<sup>2</sup> e addirittura **caotici**, in cui non c'è né ciclicità né convergenza. I sistemi dall'evoluzione caotica presentano tra l'altro una particolarità: la minima perturbazione delle loro condizioni iniziali provoca gravi cambiamenti sull'evoluzione dell'intero sistema - come a dire che una lieve modifica del punto di partenza della traiettoria del sistema fa sì che il punto finale della traiettoria sia molto diverso da quello del sistema senza perturbazione. Questo non si verifica nel-

<sup>1</sup> Per esempio, una rete con tre neuroni e tutte le possibili connessioni produce geometricamente uno spazio tridimensionale in cui è collocato un ipercubo; ogni neurone ha  $2^3$  possibili connessioni.

<sup>2</sup> A volte è la nostra stessa mente ad avere comportamenti simili (si pensi ad esempio al cubo di Necker, interpretabile in due modi diversi a seconda della prospettiva).

le reti convergenti, dove cambiare il punto di partenza non impatta l'andamento generale della traiettoria. Va da sé che il comportamento caotico non è quello che cerchiamo per le nostre reti; alcuni studi, tuttavia, dimostrano che il nostro cervello ha a volte comportamenti caotici, specialmente nel **sistema olfattorio**.

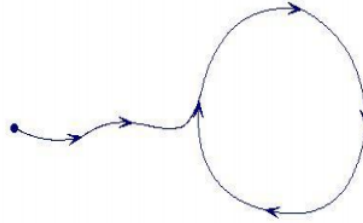


Figura 31: Traiettoria dal comportamento ciclico.

#### 6.1.2 Lyapunov e la funzione d'energia

Per capire come possano le reti di Hopfield convergere ad un punto stazionario dobbiamo appoggiarci alla teoria proposta dal fisico russo Lyapunov. Supponiamo d'avere un sistema dinamico la cui evoluzione sia rappresentata attraverso una traiettoria. Ad ogni punto della traiettoria, allora, possiamo **associare un valore reale detto energia**. Esiste quindi una funzione  $E$  tale per cui, dato un punto in uno spazio  $n$ -dimensionale, restituisce l'energia associata ad esso:

$$E : \mathbb{R}^n \rightarrow \mathbb{R}$$

Ad ogni unità di tempo la funzione d'energia può avere comportamenti molto diversi:

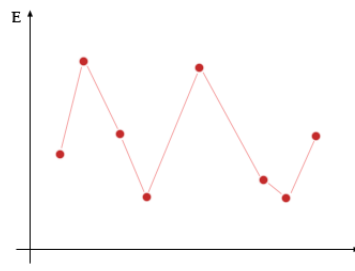
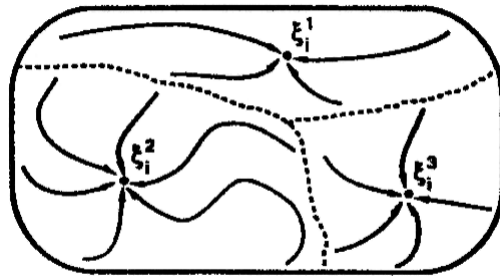


Figura 32: Andamento di una potenziale funzione d'energia.

Lyapunov ha dimostrato che se tale funzione d'energia ha un andamento **monotono decrescente** allora il sistema evolve fino a **convergere necessariamente ad un punto stazionario**. Con un'analogia,

se lanciamo una biglia in una ciotola, la biglia scenderà lungo le superfici della ciotola fino a posizionarsi sul fondo, il nostro punto stazionario. Piccole perturbazioni sul punto di partenza della biglia non cambiano il punto d'arrivo (punto stazionario *stabile*). Se la ciotola ha una struttura più complessa con più attrattori la situazione non cambia: in base al punto di partenza, la biglia convergerà su uno di questi attrattori. Per inciso, ogni attrattore ha associata una certa *forza d'attrazione* che determina l'area su cui l'attrattore ha influenza, come mostra questo schema:



L'idea di Hopfield è molto semplice: per simulare una memoria associativa come quella umana è sufficiente rendere ciò che vogliamo memorizzare un attrattore stabile. In questo modo il sistema può convergere su questi attrattori se viene fornita in input un'informazione parziale.

### 6.1.3 Hopfield e la teoria di Lyapunov

Hopfield usò la teoria sviluppata da Lyapunov per provare che le proprie reti discrete e asincrone convergono sempre in uno punto stazionario a patto che due condizioni siano rispettate:

- I **pesi sono simmetrici**, cioè  $w_{ij} = w_{ji}, \forall i, j = 1, \dots, n$ . Possono anche essere rappresentati in una matrice  $W = (w_{ij})$ , che assumiamo tale per cui  $W = W^T$ .
- **Non esistono connessioni di neuroni con se stessi**. Questo significa che  $\text{diag}(W) = 0$ .

Se queste condizioni sono valide allora associamo alla rete una funzione d'energia  $E$  **monotona decrescente** tale per cui  $E(t+1) \leq E(t)$ , con uguaglianza se si raggiunge un punto stazionario. La funzione individuata da Hopfield è:

$$E = -\frac{1}{2} \sum_i \sum_{j \neq i}^n w_{ij} V_i V_j - \sum_i I_i V_i$$

Dati due neuroni  $i, j$ , la formula è la sommatoria dei pesi che congiungono i due neuroni per lo stato di  $i$  al tempo  $t$  e lo stato di  $j$  al tempo  $t$ , a cui sottraiamo la sommatoria dello stato di  $i$  per l'elemento esterno in input a  $i$ .

#### 6.1.4 Dimostrazione della convergenza (caso discreto)

Supponiamo di avere una rete di Hopfield discreta di  $n$  neuroni e che, dal tempo  $t$  al tempo  $t + 1$ , il neurone  $h$  cambi il proprio stato. Per dimostrare che  $\Delta E = E(t + 1) - E(t) \leq 0$ , iniziamo semplicemente riscrivendo la funzione d'energia al tempo  $t$  e al tempo  $t + 1$ :

$$E(t + 1) = -\frac{1}{2} \sum_i \sum_{j \neq i} w_{ij} V_i(t + 1) V_j(t + 1) - \sum_{i \neq j} V_i(t + 1) I_i$$

$$E(t) = -\frac{1}{2} \sum_i \sum_{j \neq i} w_{ij} V_i(t) V_j(t) - \sum_{j \neq i} V_i(t) I_i$$

Calcoliamo  $\Delta E = E(t + 1) - E(t)$  raccogliendo  $w_{ij}$ :

$$\Delta E = -\frac{1}{2} \sum_i \sum_{j \neq i} w_{ij} [V_i(t + 1) V_j(t + 1) - V_i(t) V_j(t)] - \sum_{i \neq j} I_i [V_i(t + 1) - V_i(t)]$$

$V_i(t + 1) - V_i(t)$  può essere semplicemente scritto come  $\Delta V_i$ ; tuttavia tale valore è pari a 0 se  $i \neq h$ , dato che  $h$  è l'unico neurone ad aggiornare il proprio stato. L'ultima sommatoria può quindi essere eliminata, sostituendo l'indice  $i$  con  $h$ :

$$\Delta E = -\frac{1}{2} \sum_i \sum_{j \neq i} w_{ij} [V_i(t + 1) V_j(t + 1) - V_i(t) V_j(t)] - I_h \Delta V_h$$

Occupiamoci quindi della prima parte della formula, che dividiamo in due parti: una in cui  $i \neq h$  (la formula si occupa di tutti i neuroni diversi da  $h$ , l'altra in cui  $i = h$ ).

$$A_i = -\frac{1}{2} \sum_{i \neq h} \sum_{j \neq i} w_{ij} [V_i(t + 1) V_j(t + 1) - V_i(t) V_j(t)]$$

$$A_h = -\frac{1}{2} \sum_{j \neq h} w_{hj} [V_h(t+1)V_j(t+1) - V_h(t)V_j(t)]$$

**Primo termine.** Il solo neurone ad essere aggiornato è  $h$ : di conseguenza, dato che  $i \neq h$ ,  $V_i(t) = V_i(t+1)$ . Raccogliendo  $V_i(t)$  e chiamando  $V_j(t+1) - V_j(t) = \Delta V_j$  otteniamo:

$$\begin{aligned} A_i &= -\frac{1}{2} \sum_{i \neq h} \sum_{j \neq i} w_{ij} [V_i(t)V_j(t+1) - V_i(t)V_j(t)] = \\ &= -\frac{1}{2} \sum_{i \neq h} \sum_{j \neq i} w_{ij} [V_i(t)\Delta V_j] \end{aligned}$$

Ma  $\Delta V_j$  sarà uguale a 0 in tutte le situazioni in cui  $j \neq h$ , perciò eliminiamo la sommatoria su  $j$  e sostituiamo l'indice  $j$  con  $h$ :

$$A_i = -\frac{1}{2} \sum_{i \neq h} w_{ih} [V_i(t)\Delta V_h]$$

**Secondo termine.** Per le stesse ragioni di prima, dato che  $j \neq h$  allora  $V_j(t) = V_j(t+1)$ . Di conseguenza:

$$\begin{aligned} A_h &= -\frac{1}{2} \sum_{j \neq h} w_{hj} [V_h(t+1)V_j(t) - V_h(t)V_j(t)] \\ &= -\frac{1}{2} \sum_{j \neq h} w_{hj} \Delta V_h V_j(t) \end{aligned}$$

Dato che i pesi sono simmetrici, i due termini appena visti sono equivalenti:

$$-\frac{1}{2} \sum_{i \neq h} w_{ih} [V_i(t)\Delta V_h] - \frac{1}{2} \sum_{i \neq h} w_{hi} \Delta V_h V_i(t) = -\Delta V_h \sum_{i \neq h} w_{ih} V_i(t)$$

Ritorniamo quindi a  $\Delta E$ , che diventa:

$$\Delta E = -\Delta V_h \sum_{i \neq h} w_{ij} V_i(t) + I_h \Delta V_h = -\Delta V_h \left[ \sum_{i \neq h} w_{ij} V_i(t) + I_h \right]$$

Ma  $\sum_{i \neq h} w_{ij} V_i(t) + I_h$  è l'input ricevuto dal neurone  $h$  al tempo  $t+1$ . Perciò otteniamo:

$$\Delta E = -\Delta V_h H_h(t+1)$$

A questo punto dobbiamo dimostrare che  $\Delta E$  sia sempre negativo, ossia che  $\Delta V_h H_h(t+1)$  sia sempre **non negativo**. Abbiamo due casi:

- Se  $H_h \geq 0$  allora lo stato del neurone passa da  $-1$  a  $+1$ . Ma allora  $\Delta V_h = V_h(t+1) - V_h(t) = +1 - (-1) = 2$ , da cui deriva che  $\Delta E$  è negativo.
- Se  $H_h < 0$ , lo stato del neurone passa da  $+1$  a  $-1$ . Ma allora  $\Delta V_h = V_h(t+1) - V_h(t) = -1 - (+1) \leq 0$ , da cui  $\Delta E$  sicuramente negativo.

Il caso d'uguaglianza, in cui  $E(t) = E(t+1)$ , si ottiene solo quando i due punti considerati sono equivalenti, cioè quando si giunti ad un punto stazionario; dato che la dimostrazione considera un neurone che cambia stato allora è vero che  $E(t+t) < E(t+1)$ , cioè che la funzione d'energia  $E$  è *strettamente* decrescente. In questo modo escludiamo la possibilità di avere una rete con comportamenti ciclici o caotici.

## 6.2 MEMORIA ASSOCIATIVA E REGOLA DI HEBB

Dobbiamo ora capire quanti neuroni avere e come impostare i pesi della rete di Hopfield per ricreare una memoria associativa.

### 6.2.1 Quanti neuroni considerare?

Sia dato alla rete un pattern binario formato da  $N$  bit; secondo Hopfield la rete deve allora avere un singolo strato composto da  $N$  neuroni. Per esempio, per memorizzare pattern di 3 bit abbiamo bisogno di una rete con 3 neuroni.

### 6.2.2 Come impostare i pesi?

La risposta a questa domanda viene dalla cosiddetta **Regola di Hebb**, usata per descrivere il meccanismo associativo di accesso alle informazioni. Secondo questa regola se due neuroni si attivano contemporaneamente e comunicano reciprocamente, allora la loro **interconnessione viene rafforzata**. In accordo a ciò, i pesi sono così settati:

$$w_{ij} = \frac{1}{N} \sum_{\mu=1}^P x_i^{\mu} x_j^{\mu}$$

dove  $N$  è il numero di neuroni della rete. Come si nota, il peso tra i neuroni  $i$  e  $j$  dipende dal prodotto dell' $i$ -esimo e  $j$ -esimo bit dei

pattern dati in input alla rete: se i due bit sono concordi allora il peso sarà positivo e darà un contributo alla rete, altrimenti il peso sarà nullo. La formula è inoltre simmetrica, in linea con l'idea che i pesi stessi della rete siano simmetrici. Il meccanismo di **recall** è invece il seguente:

$$s_i = \text{sign}\left(\sum_j w_{ij} s_j - \theta_i\right)$$

### 6.2.3 Un esempio pratico e gli attrattori spuri

Supponiamo di voler memorizzare questi pattern:

$$x^1 = (-1, -1, -1, +1) \quad x^2 = (+1, +1, +1, +1)$$

Calcoliamo i pesi secondo la formula:

$$w_{ij} = \frac{1}{4} \sum_{\mu=0}^2 x_i^{\mu} x_j^{\mu}$$

...e otteniamo questa matrice:

$$W = \frac{1}{4} \begin{bmatrix} 2 & 2 & 2 & 0 \\ 2 & 2 & 2 & 0 \\ 2 & 2 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

Supponiamo di dare questi tre pattern in input alla rete: otteniamo come risultato i pattern più a destra.

$$\text{Input}(-1, -1, -1, +1) \rightarrow (-1, -1, -1, -1)$$

$$\text{Input}(-1, +1, +1, +1) \rightarrow (+1, +1, +1, +1)$$

$$\text{Input}(-1, -1, -1, -1) \rightarrow (-1, -1, -1, -1)^*$$

I due pattern senza asterisco sono *stabili*, cioè appartenenti ai pattern che effettivamente volevamo memorizzare; possono però esistere dei **pattern spuri**, che non appartengono alla lista dei pattern da memorizzare ma che risultano comunque memorizzati. Va da sé che molto spesso, nel memorizzare uno o due pattern, **se ne memorizzano in realtà molti di più**.



### 6.3 ALCUNI ESEMPI E APPLICAZIONI

Le reti di Hopfield discrete presentano diverse applicazioni:

#### 6.3.1 Riconoscimento di immagini

Un primo esempio d'applicazione di una rete di Hopfield discreta è il seguente: se forniamo delle immagini binarie ( $130 \times 180$  pixels, a sinistra) alla rete, il sistema riesce a recuperare le immagini più a destra (le immagini centrali sono stati intermedi). In altre parole la memoria è rappresentata da un insieme di  $P$  patterns  $x^\mu$ , con  $\mu = 1, \dots, P$ : quando viene presentato un nuovo pattern  $x$  - ad esempio un'immagine - la rete risponde producendo il pattern salvato in memoria che più somiglia a  $x$ .



Figura 33: Riconoscimento di immagini. Notare come si tratti di tre esperimenti diversi: nel primo caso diamo alla rete un'immagine rumorosa, nel secondo metà immagine, nel terzo solo un quarto dell'immagine.

#### 6.3.2 Riconoscimento di lettere o numeri

Un altro esempio di prova fatta con una rete di Hopfield di 120 neuroni consiste nella memorizzazione e riconoscimento di 7 pattern d'e-

sempio (numeri). In 7 iterazioni la rete era riuscita a ricostruire il numero 3 dopo che le era stato fornito in input un 3 rumoroso.

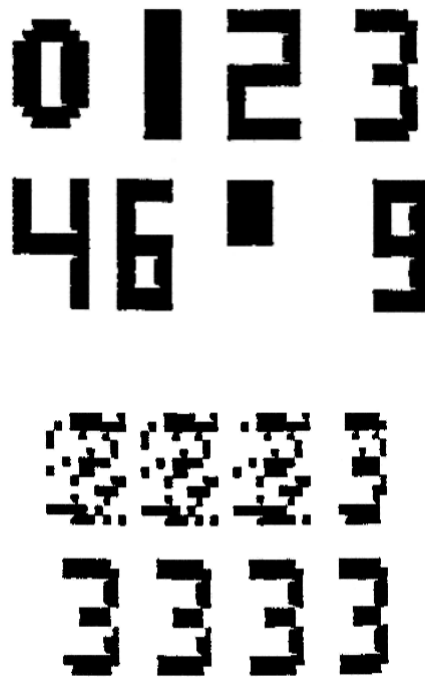


Figura 34: Riconoscimento di numeri con una rete di Hopfield discreta.

### 6.3.3 Ricapitolando...

Le reti di Hopfield discrete presentano alcuni problemi nel simulare una memoria associativa:

- La **capacità della memoria è limitata**: il numero massimo di pattern è  $0.15N$ ;
- Come visto prima, talvolta la rete produce degli stati spuri;
- Il pattern evocato non è necessariamente il più simile a quello di input;
- I pattern non sono richiamati tutti con la stessa enfasi.

## 6.4 RETI DI HOPFIELD: CASO CONTINUO

Finora abbiamo visto reti di Hopfield *discrete* sia nel tempo (presentano dei *time steps* discreti:  $t, t + \Delta t, t + 2\Delta t \dots$ ) che nello spazio (ciascun neurone ha un numero discreto di stati). Le reti di Hopfield continue,

che generalizzano le precedenti, mostrano la propria continuità sia nel tempo che nello spazio.

#### 6.4.1 Continuità spaziale delle reti di Hopfield continue

Pur seguendo il modello di McCulloch e Pitts i neuroni delle reti di Hopfield continue producono un valore reale compreso tra 0 e 1 o tra  $-1$  e  $+1$  a seconda della funzione d'attivazione scelta,  $g_\beta$ . Tale funzione è **continua, crescente e non lineare**, e lavora come sempre sul *net input* ricevuto dal neurone:

$$V_i = g_\beta(\mu_i) = g_\beta \left( \sum_j w_{ij} V_j + I_i \right)$$

Il **parametro**  $\beta$  indica la **stickiness della funzione**  $g$  all'asse delle  $y$  o, in altre parole, la derivata della funzione  $g$  nel punto  $(0,0)$ ; per  $\beta \rightarrow +\infty$  la funzione diventa sempre più ripida fino a trasformarsi nella funzione segno.

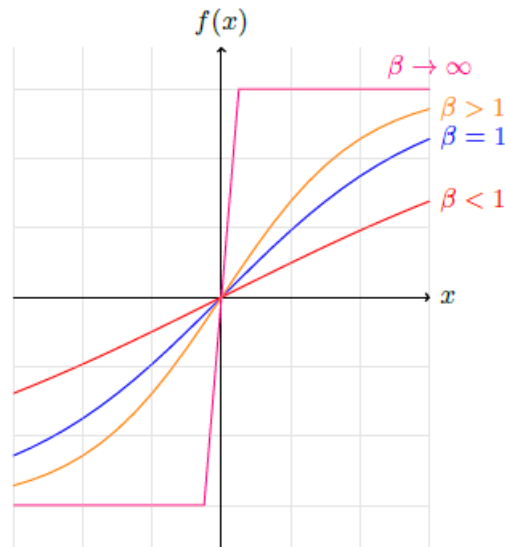


Figura 35: Possibili andamenti della funzione  $g$ , per  $\beta \rightarrow \infty$

Le funzioni più usate sono la *tangente iperbolica* e la *sigmoidea*:

$$\tanh_\beta(\mu) = \frac{e^{\beta\mu} - e^{-\beta\mu}}{e^{\beta\mu} + e^{-\beta\mu}} \in ]-1, +1[$$

$$g_\beta(\mu) = \frac{1}{1 + e^{-2\beta\mu}} \in ]0, 1[$$

### 6.4.2 Continuità temporale delle reti di Hopfield continue

Per capire cosa s'intenda per continuità temporale dobbiamo prima comprendere la differenza tra *tempo discreto* e *tempo continuo* di un sistema. Supponiamo di avere una rete di due neuroni i cui stati possono essere rappresentati come un punto in uno spazio bidimensionale.

- **Tempo discreto:** per conoscere completamente l'evoluzione di un sistema discreto basta conoscerne il punto di partenza  $x(0)$  della traiettoria e il vettore contenente la differenza tra due vettori,  $\Delta x(t)$ : in questo modo posso sempre calcolare il punto successivo a cui passare.
- **Tempo continuo:** per conoscere completamente l'evoluzione di un sistema continuo dobbiamo conoscerne il punto di partenza  $x(0)$  e il *tangent vector*, la differenza cioè tra due vettori successivi quando  $\Delta t \rightarrow 0$ .

Le reti di Hopfield continue hanno quindi bisogno del concetto di derivata – tanto più che la stessa **definizione di punto stazionario** diventa la seguente:

Un punto è stazionario se **ogni derivata in tale punto va a 0** o, equivalentemente, se la tangente in quel punto è il vettore nullo.

L'uso di valori continui permette la modalità di **aggiornamento continuo** dei neuroni. Hopfield ha proposto due modelli per descrivere l'evolversi della sua rete continua, ma si tratta di formulazioni equivalenti; entrambe pongono l'accento su uno dei due elementi fondamentali di ciascun neurone  $i$ , il **net input** o l'**output**.

### 6.4.3 Modello basato su $V_i$

Il modello che descrive l'evolversi della rete neurale basandosi sull'**output del  $i$ -esimo neurone**,  $V_i$ , è:

$$V_i + \tau_i \left( \frac{dV_i}{dt} \right) = g_\beta(\mu_i) = g_\beta \left( \sum_j w_{ij} V_j + I_i \right)$$

...dove  $\tau_i$  è una costante positiva che rappresenta la resistenza elettrica. Il sistema si stabilizza e raggiunge un punto stazionario quando, per ogni neurone  $i$ ,  $\frac{dV_i}{dt} = 0$ . Ne deriva quindi che:

$$V_i + \tau_i(0) = g_\beta(\mu_i) \rightarrow V_i = g_\beta(\mu_i)$$

...o, in altre parole, che quando la rete converge ad un punto stazionario allora l'output di ogni neurone è uguale alla funzione d'attivazione  $g_\beta$  sul suo *net input*.

#### 6.4.4 Modello basato su $\mu_i$

Il modello basato sul **net input** dell' $i$ -esimo neurone non è molto diverso dal precedente:

$$\mu_i + \tau_i \left( \frac{d\mu_i}{dt} \right) = \sum_j w_{ij} V_j + I_i$$

Il sistema si stabilizza e raggiunge un punto stazionario quando, per ogni neurone  $i$ ,  $\frac{d\mu_i}{dt} = 0$ , cioè quando:

$$\mu_i = \sum_j w_{ij} V_j + I_i$$

Per questioni di semplicità è questo il **modello scelto da Hopfield** per descrivere l'aggiornamento di stato continuo di ciascun neurone.

#### 6.4.5 L'apporto di Lyapunov e la convergenza di una rete continua

Come già fatto per il precedente modello, per comprendere in che modo convergano le reti continue di Hopfield dobbiamo basarci sulla teoria di Lyapunov. Supponiamo d'avere una rete di Hopfield continua in cui:

- i pesi sono simmetrici;
- i neuroni non sono connessi a se stessi (la diagonale della matrice dei pesi è 0).

Allora la rete ha associata una **funzione d'energia**  $E$  **monotona decrescente** tale per cui  $\frac{dE}{dt} \leq 0$ , con uguaglianza quando il sistema ha

raggiunto un punto stazionario. La funzione individuata da Hopfield è:

$$E(t) = -\frac{1}{2} \sum_i \sum_j w_{ij} V_i(t) V_j(t) + \sum_i \int_0^{V_i(t)} g_\beta^{-1}(V) dV - \sum_i I_i V_i(t)$$

Si tratta cioè della stessa funzione d'energia vista per il caso discreto con l'**aggiunta di un nuovo elemento**, che si annulla quando  $\beta \rightarrow \infty$ . Vogliamo ora dimostrare il teorema. Visto che la derivata di una somma equivale alla somma delle sue derivate possiamo semplificare i calcoli suddividendo la funzione d'energia in tre termini, di cui calcoliamo le derivate separatamente.

- $A = -\frac{1}{2} \sum_i \sum_j w_{ij} V_i(t) V_j(t)$
- $B = + \sum_i \int_0^{V_i(t)} g_\beta^{-1}(V) d(V)$
- $C = - \sum_i I_i V_i(t)$

**Derivata di A.** Vogliamo calcolare la derivata del termine A:

$$A = -\frac{d}{dt} \left( -\frac{1}{2} \sum_i \sum_j w_{ij} V_i(t) V_j(t) \right)$$

Dato che  $w_{ij}$  è una costante spostiamo il calcolo della derivata in questo modo:

$$A = -\frac{1}{2} \sum_i \sum_j w_{ij} \frac{d}{dt} (V_i(t) V_j(t))$$

Per le regole di derivazione otteniamo:

$$A = -\frac{1}{2} \sum_i \sum_j w_{ij} \left[ \frac{dV_i}{dt} V_j(t) + \frac{dV_j}{dt} V_i(t) \right]$$

Separiamo le due parti della sommatoria: dato che i **pesi sono simmetrici**, possiamo ridurre il tutto a:

$$\begin{aligned} A &= -\frac{1}{2} \sum_i \sum_j w_{ij} \frac{dV_i}{dt} V_j(t) - \frac{1}{2} \sum_i \sum_j w_{ij} V_j(t) \frac{dV_j}{dt} \\ &= - \sum_i \frac{dV_i}{dt} \sum_j w_{ij} V_j(t) \end{aligned}$$

**Derivata di C.** La derivata di C non è complicata, visto che basta spostare la derivazione all'unico termine che ci interessa,  $V_i(t)$ :

$$C = \frac{d}{dt} \left( - \sum_i I_i V_i(t) \right) = - \sum_i I_i \frac{dV_i(t)}{dt}$$

**Derivata di B.** La derivata di B è forse il termine più complesso. Per prima cosa ricordiamoci che la derivata di una somma equivale alla somma delle sue derivate:

$$B = + \frac{d}{dt} \left( \sum_i \int_0^{V_i(t)} g_\beta^{-1} dV \right) = \sum_i \frac{d}{dt} \int_0^{V_i(t)} g_\beta^{-1}(V) dV$$

Quindi osserviamo che si tratta di calcolare la derivata di una funzione composta: abbiamo infatti  $t \rightarrow V_i(t) \rightarrow \int_0^{V_i(t)} g_\beta^{-1}(V) dV$ . Di conseguenza la derivata è:

$$\sum_i \frac{dV_i(t)}{dt} * \frac{d \int_0^{V_i(t)} g_\beta^{-1}(V) dV}{dt}$$

La derivata di un integrale altro non è che la funzione nell'integrale stesso. Otteniamo perciò:

$$\sum_i \frac{dV_i(t)}{dt} g_\beta^{-1}(V_i(t)) = \sum_i \frac{dV_i(t)}{dt} \mu_i$$

e questo perché  $g_\beta^{-1} V_i(t) = \mu_i$ , cioè è il **net input dell'i-esimo neurone**. Riassumendo,  $\frac{dE}{dt}$  è uguale a:

$$\frac{dE}{dt} = - \sum_i \frac{dV_i}{dt} \sum_j w_{ij} V_j + \sum_i \frac{dV_i}{dt} \mu_i - \sum_i I_i \frac{dV_i(t)}{dt}$$

Raccogliamo le parti in comune:

$$\frac{dE}{dt} = - \sum_i \frac{dV_i}{dt} \left[ \sum_j w_{ij} V_j - \mu_i + I_i \right]$$

Ma  $\sum_j w_{ij} V_j - \mu_i + I_i$  è  $\tau_i \frac{d\mu_i}{dt}$ , per cui:

$$\frac{dE}{dt} = - \sum_i \frac{dV_i(t)}{dt} \tau_i \frac{d\mu_i}{dt}$$

Passiamo quindi all'unica derivata rimasta. Dato che  $V_i(t) = g_\beta(\mu_i(t))$  è una funzione composta, applichiamo la regola della catena:

$$\frac{dV_i(t)}{dt} = \frac{d\mu_i}{dt} g'(\mu_i(t))$$

Perciò la nostra formula diventa:

$$\begin{aligned} \frac{dE}{dt} &= - \sum_i \frac{dV_i}{dt} \tau \frac{d\mu_i}{dt} = - \sum_i \tau \frac{d\mu_i}{dt} g'(\mu_i(t)) \frac{d\mu_i}{dt} = \\ &= - \sum_i \tau \left( \frac{d\mu_i}{dt} \right)^2 g'(\mu_i(t)) \end{aligned}$$

Dobbiamo ora dimostrare che  $\frac{dE}{dt}$  è sempre non positivo. Sicuramente  $\left( \frac{d\mu_i}{dt} \right)^2$  è sempre positivo; lo stesso vale per  $g'(\mu_i(t))$ : dato che  $g$  è una funzione monotona crescente, la sua derivata è sempre positiva. Di conseguenza  $\frac{dE}{dt} \leq 0$ , con uguaglianza quando  $\frac{d\mu_i}{dt} = 0$ . Il teorema è provato.

## 6.5 CORRISPONDENZA TRA I DUE MODELLI

Mettiamo a confronto le due funzioni d'energia:

- Caso continuo:

$$E = -\frac{1}{2} \sum_i \sum_j w_{ij} V_i(t) V_j(t) + \sum_i \int_0^{V_i(t)} g_\beta^{-1}(V) dV - \sum_i I_i V_i(t)$$

- Caso discreto:

$$E = -\frac{1}{2} \sum_i \sum_j w_{ij} V_i(t) V_j(t) - \sum_i I_i V_i(t)$$

Quando  $\beta \rightarrow \infty$ , il termine aggiuntivo  $\sum_i \int_0^{V_i(t)} g_\beta^{-1}(V) dV$  tende a 0 e, di conseguenza, ha un minimo impatto sulla funzione d'energia (che diventa uguale a quella del caso discreto).

Hopfield fu poi motivato ad approfondire l'applicazione di queste reti seguendo l'intuizione secondo cui, data una rete dai pesi simmetrici, è possibile risolvere problemi d'ottimizzazione in cui si cerca di minimizzare la funzione d'energia.





## OTTIMIZZAZIONE CON LE RETI NEURALI

---

Vediamo ora alcuni problemi **NP-difficili**<sup>1</sup>: il *problema del commesso viaggiatore* e il *problema della clique massima*. Si tratta di problemi **non risolvibili in tempo polinomiale**: se ne vedrà quindi una soluzione in termini di reti neurali.

### 7.1 IL PROBLEMA DEL COMMESSE VIAGGIATORE (TSP)

Sia data una rete di  $n$  città; il problema del commesso viaggiatore consiste nel trovare il percorso di lunghezza minima che il commesso viaggiatore deve seguire per visitare ogni città una e una sola volta e poi tornare alla città di partenza. Formalmente, abbiamo:

- $n$  città rappresentate come vertici di un grafo;
- $d_{xy}$  distanze rappresentate come archi di un grafo ( $d_{xy}$  è la distanza tra città  $x$  e città  $y$ ). Le distanze possono essere organizzate in una matrice  $D_{n \times n} = (d_{xy})$  **simmetrica**, dove cioè  $d_{xy} = d_{yx}$ .

Il problema del commesso viaggiatore diventa quindi un problema di **minimizzazione della distanza percorsa**. Non è possibile ottenerne una soluzione analiticamente dato che il numero di possibili percorsi è:

$$\frac{n!}{2n} = \frac{(n-1)!}{2}$$

A ben guardare, infatti, i percorsi non sono altro che il numero di permutazioni delle  $n$  città diviso per 2 (per eliminare percorsi equivalenti tra loro, come ABCD e BCDA). Si tratta di un numero di permutazioni estremamente alto: se per esempio consideriamo solo 20 città, arriviamo ad avere  $10^{18}$  possibili percorsi – un valore il cui calcolo è irrealizzabile (*unfeasible*).

---

<sup>1</sup> Un problema  $P$  è NP-difficile se ogni problema  $L \in NP$  è riconducibile in tempo polinomiale a  $P$ . Un problema  $P$  è invece NP-completo se  $P \in NP$  ed è NP-difficile.

### 7.1.1 Topologia della rete: tradurre le permutazioni

Per prima cosa dobbiamo **tradurre il problema del commesso viaggiatore in termini di reti neurali**. La traduzione presenta un ostacolo:

- Il problema del commesso viaggiatore, per com'è posto in origine, tratta **permutazioni di lettere** rappresentanti possibili percorsi (ABCD, BDCA, ...);
- La rete neurale tratta solo ed esclusivamente **valori numerici**.

L'idea di Hopfield è di usare una **matrice di permutazione**, quadrata e di elementi binari (0/1) dove **ogni riga e ogni colonna presentano un solo 1** (tutti gli altri elementi sono a 0). Un esempio di matrice di permutazione  $4 \times 4$  è:

	1	2	3	4
A	0	0	1	0
B	1	0	0	0
C	0	0	0	1
D	0	1	0	0

Le matrici di permutazione sono in grado di rappresentare i percorsi tra le città: le righe rappresentano infatti le città coinvolte mentre le colonne indicano le tappe del percorso. La tabella precedente, di conseguenza, mostra il percorso  $B \rightarrow D \rightarrow A \rightarrow C$ . Questo approccio presenta chiaramente sia vantaggi che svantaggi:

- **Vantaggi:** le matrici di permutazione traducono il problema del commesso viaggiatore in termini numerici, permettendoci di trattarlo tramite le reti di Hopfield.
- **Svantaggi:** le matrici di permutazione rappresentano un problema lineare ( $n$  città) in forma quadratica (matrice di permutazione  $n \times n$ ).

### 7.1.2 Topologia della rete: gli elementi fondamentali

Il problema del commesso viaggiatore viene tradotto da Hopfield come segue:

- Se  $X$  rappresenta una delle  $n$  città mentre  $i$  rappresenta una generica fermata nel percorso, allora la rete neurale deve avere

$n^2$  **neuroni**, ciascuno rappresentante la coppia  $(X, i)$ . Notare che l'indice  $i$  è sempre inteso **modulo**  $n$ .

- La coppia di neuroni  $X, i$  e  $Y, j$  è connessa da un peso, indicato con  $w_{X,i,Y,j}$ . Il peso presenta 4 indici.
- La distanza tra due città  $X$  e  $Y$  è rappresentata come  $d_{X,Y}$ ;
- $V_{X,i}$  è lo stato dell'unità  $X, i$ ; è 1 se la città  $X$  è visitata alla fermata  $i$ , 0 altrimenti. Notare che sebbene si ragioni in termini di zeri e uni il modello che useremo è *continuo*: ogni neurone ha cioè output compresi tra 0 e 1; quando il sistema converge, gli output dei neuroni sono *vicini* a 0 o 1.

### 7.1.3 Topologia della rete: la funzione obiettivo

Per prima cosa dobbiamo individuare una funzione d'energia adeguata; usiamo la funzione d'energia del modello discreto di rete di Hopfield, **ragionando però su 4 indici**.

$$E = -\frac{1}{2} \sum_x \sum_y \sum_i \sum_j w_{x,i,y,j} V_{x,i} V_{y,j} - \sum_x \sum_i I_{x,i} V_{x,i}$$

### 7.1.4 Vincolo sulle distanze e calcolo dei pesi

Consideriamo ora il percorso BCDA; la distanza complessiva del percorso è la somma delle distanze della matrice:

$$d(\text{BCDA}) = d_{bc} + d_{cd} + d_{da} + d_{ab}$$

La rete di Hopfield ha accesso solo alla matrice delle distanze e alla matrice di permutazione corrispondente a questo percorso:

	1	2	3	4
A	0	0	0	1
B	1	0	0	0
C	0	1	0	0
D	0	0	1	0

Per calcolare la distanza complessiva dobbiamo allora procedere per passi, moltiplicando la prima colonna per la seconda, la seconda

per la terza, la terza per la quarta e la quarta per la prima. Dato che ciascuna colonna conterrà solo un 1 allora il risultato del prodotto sarà 1 in un solo caso, 0 nei rimanenti. Hopfield, in realtà, ha proposto una versione più generalizzata di questo schema, calcolando la distanza **non solo tra  $i$  e  $i + 1$  ma anche tra  $i$  e  $i - 1$** . In formule:

$$\sum_x \sum_i \sum_{y \neq x} d_{xy} (V_{y,i-1} + V_{y,i+1}) V_{xi}$$

Supponendo per esempio che  $i = 2$ : con questa formula otteniamo la somma delle distanze tra le tappe 2,1 e tra le tappe 2,3. Solo in un caso otteniamo 1 (da cui si ottiene la distanza) per entrambe le coppie: negli altri casi il risultato è sempre 0.

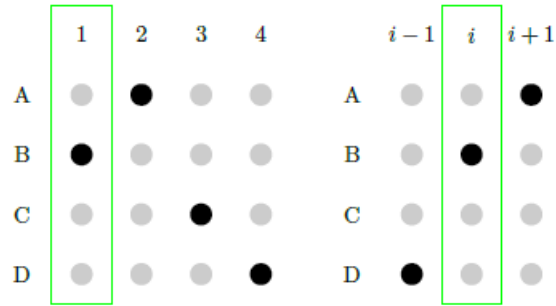


Figura 36: Il calcolo delle distanze per il problema del commesso viaggiatore.

Il calcolo della distanza visto finora è molto simile alla funzione d'energia descritta precedentemente, ma presenta **una sommatoria in meno**; per questioni d'omogeneità dobbiamo allora riscrivere la seconda formula nei termini della prima. Per prima cosa impostiamo le 4 sommatorie:

$$\frac{1}{2} \sum_x \sum_y \sum_i \sum_j$$

Ci serviamo poi della **notazione Kroniker delta** secondo cui, dati due indici, se sono uguali abbiamo 1, altrimenti 0:

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

La funzione per il calcolo della distanza può allora essere mappata evidenziando, tramite il *kroniker delta*, quando gli indici sono uguali o diversi tra loro:

$$\frac{1}{2} \sum_i \sum_x \sum_y \sum_j d_{xy} (\delta_{j,i+1} + \delta_{j,i-1}) V_{xi} V_{yj} \quad (1)$$

In questo modo solo se  $j \neq i+1$  o  $j \neq i-1$  i valori tra parentesi non contribuiscono alla sommatoria. I **pesi della rete**, di conseguenza, sono i seguenti:

$$w_{xi,yj} = d_{xy} (\delta_{j,i+1} + \delta_{j,i-1})$$

Di conseguenza se costruiamo una rete con  $n^2$  neuroni e impostiamo i pesi della rete secondo la formula precedente, assumendo che i pesi siano simmetrici il sistema evolverà nel tempo fino a minimizzare la funzione obiettivo al punto (1).

## 7.2 PENALIZZAZIONE DELLA FUNZIONE-OBIETTIVO

Siamo sicuri che basti settare i pesi come visto precedentemente per minimizzare (1) e ottenere la matrice di permutazione che cerchiamo? In principio dovremmo rispondere di sì ma, in realtà, abbiamo **bisogno di ulteriori vincoli** – primo fra tutti quelli relativi alla matrice di permutazione. Siamo infatti partiti dall'assunzione che la rete produca in output la matrice di permutazione che minimizza il percorso del commesso viaggiatore e che presenti un solo 1 per riga e colonna; tale matrice, però, può potenzialmente **non soddisfare questi vincoli**: dobbiamo quindi impostarli in modo da far convergere la rete ad una corretta matrice di permutazione. Per fare ciò trasformiamo il problema del commesso viaggiatore in un **problema di ottimizzazione unconstrained**:

- **Constrained.** Consideriamo una funzione  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  e un insieme  $D \subset \mathbb{R}^n$ ; un problema di ottimizzazione *constrained* consiste nel voler minimizzare  $f(x)$  imponendo che  $x \in D$ ;
- **Unconstrained.** In questo caso vogliamo minimizzare invece  $f(x) + \lambda P(x)$  senza imporre nulla. Abbiamo però un *penalty term*,  $P(x)$ : è una funzione aggiuntiva che è pari a 0 se e solo se  $x \in D$ , positiva altrimenti.  $\lambda$  è invece uno scalare. In questo modo se

scegliamo un punto appartenente a  $D$  il problema di ottimizzazione diventa equivalente al problema d'ottimizzazione *constrained*; se il punto scelto è esterno a  $D$ , invece, la funzione  $f(x)$  è penalizzata da  $P(x)$ , che sarà un valore positivo. Intuitivamente, più mi allontano da  $D$  più la penalizzazione (il valore di  $P$ ) aumenta.

Hopfield ha quindi semplicemente **aggiunto dei termini di penalizzazione alla funzione-obiettivo** vista precedentemente in modo da obbligare la rete a convergere ad una matrice di permutazione valida. Vediamo quali sono questi termini di penalizzazione.

### 7.2.1 Penalizzazione sulle righe

Ogni riga della matrice di permutazione deve contenere esattamente un 1. In formule:

$$E_{\text{row}} = \frac{1}{2} \sum_x \sum_i \sum_{j \neq i} V_{xi} V_{xj}$$

Per esempio, se abbiamo la riga  $x$  di una matrice, quando moltiplico ciascun elemento della riga con i rimanenti ottengo:

0	0	1	0	1	1
---	---	---	---	---	---

$$0 * 0 + 0 * 1 + 0 * 0 + 0 * 1 + 0 * 1 = 0 \rightarrow 0;$$

$$0 * 0 + 0 * 1 + 0 * 0 + 0 * 1 + 0 * 1 = 0 \rightarrow 0;$$

$$1 * 0 + 1 * 0 + 1 * 0 + 1 * 1 + 1 * 1 = 2 \rightarrow 1; \quad \dots$$

In altre parole, per ogni riga si moltiplica ciascun elemento per i rimanenti; se esiste esattamente un solo 1 l'output del vincolo è 0 (e il vincolo sparisce).

### 7.2.2 Penalizzazione sulle colonne

Si tratta di un vincolo sostanzialmente identico al precedente:

$$E_{\text{col}} = \frac{1}{2} \sum_x \sum_i \sum_{y \neq x} V_{xi} V_{yi}$$

Come prima, per ogni colonna si moltiplica ciascun elemento per i rimanenti; se esiste esattamente un solo 1 l'output del vincolo è 0.

### 7.2.3 Vincolo globale input

Il vincolo globale impone che la **matrice di permutazione contenga esattamente**  $n$  1:

$$E_{\text{glob}} = \frac{1}{2} \left( \sum_x \sum_i V_{xi} - n \right)^2$$

In altre parole, sommo tutto gli elementi della matrice e vi sottraggo  $n$ : se ottengo 0 allora la matrice è correttamente impostata.

### 7.2.4 Vincolo sulle distanze

Ricordiamoci infine dell'ulteriore vincolo che ora chiamiamo  $E_{\text{data}}$ :

$$E_{\text{data}} = \frac{1}{2} \sum_x \sum_i \sum_{y \neq x} d_{xy} (V_{y,i-1} + V_{y,i+1}) V_{xi}$$

Allora la funzione da minimizzare con l'aggiunta dei suoi penalty terms è:

$$E_{\text{TSP}} = E_{\text{data}} + E_{\text{row}} + E_{\text{col}} + E_{\text{glob}}$$

...dove il primo termine rappresenta i dati su cui lavoriamo (le distanze), i rimanenti sono invece i *penalty terms*. Può anche essere utile avere dei parametri per determinare il peso di ciascun termine della funzione obiettivo. Abbiamo quindi quattro parametri,  $A, B, C, D$ , che sostituiamo così:

$$E_{\text{row}} = \frac{A}{2} \sum_x \sum_i \sum_{j \neq i} V_{xi} V_{xj}$$

$$E_{\text{col}} = \frac{B}{2} \sum_x \sum_i \sum_{y \neq x} V_{xi} V_{yi}$$

$$E_{\text{glob}} = \frac{C}{2} \left( \sum_x \sum_i V_{xi} - n \right)^2$$

$$E_{\text{data}} = \frac{D}{2} \sum_x \sum_i \sum_{y \neq x} d_{xy} (V_{y,i-1} + V_{y,i+1}) V_{xi}$$

C'è un ultimo problema da risolvere: la funzione d'energia presenta quattro sommatorie, questi vincoli solo tre. Abbiamo già sistemato



precedentemente  $E_{data}$ ; ci mancano gli altri tre vincoli. Ci serviamo di nuovo del kroniker delta e otteniamo:

$$E_{row} = \frac{A}{2} \sum_x \sum_y \sum_i \sum_j V_{xi} V_{yj} \delta_{xy} (1 - \delta_{ij})$$

Il kroniker delta serve ad evidenziare che  $x$  dev'essere uguale a  $y$  (altrimenti la sommatoria diventa 0) e che  $j$  dev'essere diverso da  $i$ . Le stesse considerazioni valgono per il vincolo sulle colonne:

$$E_{col} = \frac{B}{2} \sum_x \sum_y \sum_i \sum_j V_{xi} V_{yj} \delta_{ij} (1 - \delta_{xy})$$

$$E_{glob} = \frac{C}{2} \left( \left( \sum_x \sum_y V_{xi} \right)^2 - 2n \sum_x \sum_i V_{xi} + n^2 \right)$$

dove

$$\begin{aligned} \left( \sum_x \sum_y V_{xi} \right)^2 &= \left( \sum_x \sum_y V_{xi} \right) \left( \sum_x \sum_y V_{xi} \right) \\ &= \sum_x \sum_y \sum_i \sum_j V_{xi} V_{yj} \end{aligned}$$

Mettendo tutto insieme otteniamo ciò che dobbiamo minimizzare. I pesi della rete sono quindi:

$$w_{xi,yj} = -D(\delta_{j,i-1} + \delta_{j,i+1}) - A\delta_{xy}(1 - \delta_{ij}) - B\delta_{xy}(1 - \delta_{ij}) - C$$

### 7.2.5 Un esempio

Hopfield ha proposto diversi esperimenti sul problema. Uno dei primi prevedeva lo studio di  $n = 10$  città e, perciò, una rete di  $10^2$  neuroni; la rete iniziava con una configurazione casuale delle 10 città, con i parametri settati  $A = B = D = 500, C = 200$ ; progressivamente la rete riusciva a convergere al percorso DHIFGEAJCB. Notare che ciascun punto nella rappresentazione sottostante indica l'**attività di un neurone**: un piccolo quadrato rappresenta un'attività vicina a 0; un quadrato grande rappresenta invece un'attività vicina 1. Negli stati intermedi possiamo vedere che alcune righe o colonne presentano più quadrati di medie dimensioni; progressivamente la matrice si stabilizza diventando una corretta matrice di permutazione.

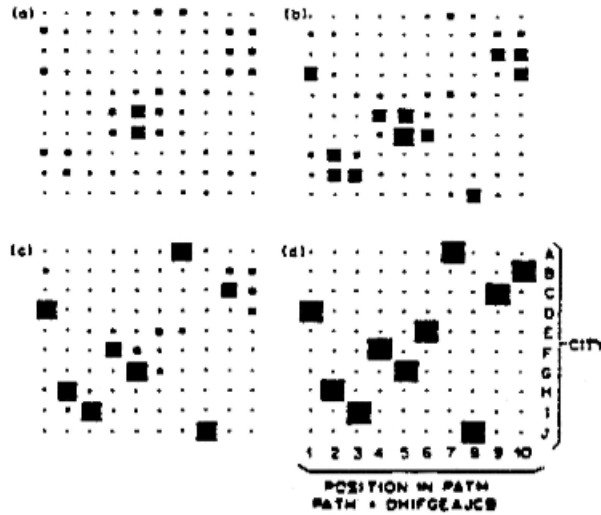


Figura 37: Passaggi intermedi verso la convergenza ad un corretto percorso.

### 7.3 UN'ALTRA FORMULAZIONE DEL TSP

Determinare i parametri A, B, C, D è particolarmente difficile. Possiamo quindi sfruttare un altro modo per esprimere i vincoli del TSP che richiede di **determinare un parametro in meno**:

$$E_{\text{row}} = \frac{A}{2} \sum_x \left( \sum_i V_{xi} - 1 \right)^2$$

$$E_{\text{col}} = \frac{B}{2} \sum_i \left( \sum_x V_{xi} - 1 \right)^2$$

La prima formula andrà a 0 solo se ciascuna riga conterrà esattamente un solo 1; la seconda formula andrà a 0 solo se ciascuna colonna conterrà esattamente un solo 1. Possiamo quindi eliminare il terzo vincolo  $E_{\text{glob}}$ . La funzione d'energia diventa quindi:

$$E = \frac{D}{2} \sum_x \sum_{y \neq x} \sum_i d_{xy} V_{xi} (V_{y,i+1} + V_{y,i-1}) + E_{\text{row}} + E_{\text{col}}$$

### 7.4 IL PROBLEMA DELLE $n$ REGINE

Le reti di Hopfield sono state applicate anche per risolvere altri problemi, ad esempio di **natura combinatoria**; uno di questo è il problema delle  $n$  regine, una variante del problema del commesso viaggiatore: data una scacchiera  $n \times n$ , si vogliono posizionare  $n$  regine in

modo che nessuna di esse possa attaccare un'altra<sup>2</sup>. Per risolvere il problema costruiamo una rete di Hopfield con  $n \times n$  in cui il neurone  $(i, j)$  è attivo solo se una regina occupa la posizione  $(i, j)$ . I vincoli da tener presente sono:

- Solo una regina in ciascuna riga;
- Solo una regina in ciascuna colonna;
- Solo una regina in ciascuna diagonale;
- Solo  $n$  regine sulla scacchiera.

Il vincolo sulle distanze, tipico del TSP, è **sostituito da un vincolo sulle diagonali**. La matrice dei pesi è quindi la seguente:

$$-w_{ij,kl} = A\delta_{ik}(1 - \delta_{jl}) + B\delta_{jl}(1 - \delta_{ik}) + \\ C + D(\delta_{i+j,k-l} + \delta_{i-j,k-l})(1 - \delta_{ik})$$

...dove l'ultimo elemento è il vincolo sulla diagonale.

#### 7.4.1 Conclusioni

L'ottimizzazione tramite le reti di Hopfield presenta alcuni svantaggi:

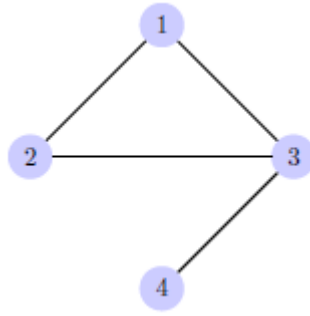
1. sono necessari  $n^2$  neuroni;
2. il numero di connessioni è  $O(n^4)$ ;
3. i parametri  $A, B, C, D$  sono difficili da determinare;
4. non esiste la garanzia che i risultati ottenuti siano soluzioni ammissibili, ovvero matrici binarie con i vincoli rispettati;
5. è difficile evitare i minimi locali della funzione di energia.

### 7.5 IL PROBLEMA DELLA CLIQUE MASSIMA (MCP)

Sia dato un grafo  $G$  non orientato, con  $V = 1 \dots n$  vertici e  $E = 1 \dots m$  archi non orientati; la **clique** (o cricca) è un sottoinsieme di vertici  $C \subset V$  mutualmente adiacenti ovvero tali per cui **esiste un arco tra ogni coppia di vertici di  $C$** . Per esempio, dato il seguente grafo,  $C_1 = \{1, 2, 3\}$  è una clique; lo stesso non si può dire per  $C_2 = \{2, 3, 4\}$  dato che non esiste un arco tra i vertici 2 e 4.

Diamo poi due definizioni:

<sup>2</sup> Le regine possono muoversi verticalmente, orizzontalmente o in diagonale.



- **Clique massimale:** è una clique di  $G$  non contenuta in nessuna altra clique più grande.  $\{1, 2, 3\}$  è una clique massimale,  $\{2, 3\}$  è una clique ma non è massimale.
- **Clique massima:** è una clique massimale di  $G$  con **cardinalità massima**.  $\{1, 2, 3\}$  è una clique massima.

Trovare una clique massimale è un problema facile mentre trovare la clique massima è NP-completo, così come lo è trovare la dimensione di tale clique. Prima di dare una formulazione continua del problema della clique massima sono necessarie alcune definizioni.

#### 7.5.1 Definizione: vettore caratteristico

Dato un sottoinsieme di vertici  $C \subset V$ , indichiamo con  $x^C$  il **vettore caratteristico** di  $C$ :

$$x_i^C = \begin{cases} \frac{1}{|C|} & \text{se } i \in C \\ 0 & \text{altrimenti} \end{cases}$$

Si tratta di un vettore continuo formato da tanti elementi quanti sono i vertici del grafo: se il vertice appartiene a  $C$  allora l'elemento è  $\frac{1}{|C|}$ , altrimenti l'elemento è 0. Per esempio, se  $|V| = 4$  e  $C = \{1, 2, 4\}$ , allora  $x^C = [\frac{1}{3}, \frac{1}{3}, 0, \frac{1}{3}]$ .

#### 7.5.2 Definizione: simpleso standard

Il simpleso standard di  $\mathbb{R}^n$ , chiamato  $S$ , è l'insieme di tutti i componenti  $x \in \mathbb{R}^n$  tali per cui i componenti sono non negativi e la loro somma è 1.

$$S = \left[ x \in \mathbb{R}^n : \sum_{i=1}^n x_i = 1, x_i \geq 0 \forall i \right]$$

Quando  $n = 3$ , il semplice standard è così rappresentabile:

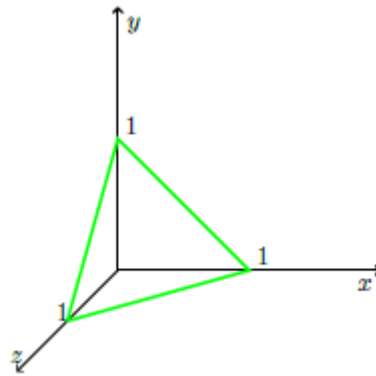


Figura 38: Semplesso standard di tre dimensioni.

Notare che i vettori caratteristici **appartengono al simplesso standard**, dato che i suoi componenti sono tutti non-negativi e la loro somma ammonta sempre a 1.

#### 7.5.3 Definizione: matrice di adiacenza

Dato un grafo  $G$  di  $n$  vertici, è possibile costruire una **matrice di adiacenza**  $A_{n \times n}$  in cui l'elemento di coordinate  $a_{ij}$  è 1 se esiste un arco tra i vertici  $i$  e  $j$ , 0 altrimenti. Tipicamente, dato che stiamo parlando di un grafo non orientato, la **diagonale** è 0.

	1	2	3	4
1	0	1	0	0
2	1	0	1	0
3	1	1	0	0
4	0	0	1	0

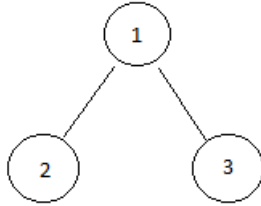
Tabella 1: Esempio di matrice d'adiacenza

#### 7.5.4 Una prima soluzione al problema

Per risolvere il problema della clique massima dobbiamo usare il **lagrangiano del grafo**, una funzione quadratica continua – in modo tale che possa essere trattabile tramite reti neurali – e *unica* per ciascun grafo. Sia dato un grafo non orientato  $G = (V, E)$  con  $|V| = n$  vertici: se associamo a ciascun vertice una variabile  $x$  il lagrangiano si calcola come segue:

$$f_G^{(x)} = x^T A x = \sum_i \sum_j a_{ij} x_i x_j = \sum_{(i,j) \in E} x_i x_j$$

$a_{ij}$  contribuisce al prodotto solo quando esiste un arco tra  $i$  e  $j$ ; il Lagrangiano si può quindi **riscrivere in termini di archi**, ottenendo un polinomio con tanti termini quanti sono gli archi presenti nel grafo. Per esempio, consideriamo il seguente grafo formato da 3 vertici:



Associamo a ogni vertice una variabile,  $x_1, x_2, x_3$ ; Dato che non esiste un arco tra i vertici 2,3 abbiamo:

$$f(x_1, x_2, x_3) = 0 * x_1 x_3 + 1 * x_1 x_2 + 1 * x_1 x_3 = x_1 x_2 + x_1 x_3$$

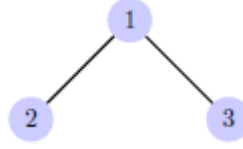
Servendosi di questa nozione, nel 1965 **Motzkin e Straus** individuano un modo per **calcolare la cardinalità della clique massima** e per **trovare la clique massima**.

- **Cardinalità.** Sia  $x^*$  un massimo globale di  $f_G$  nel simpleso standard  $S$ ; allora la cardinalità della clique massima di  $G$  è legata a  $f_G$  in questo modo:

$$w(G) = \frac{1}{1 - f_G(x^*)}$$

- **Clique.** Un sottoinsieme di vertici,  $C$ , è una clique massima se e solo se il suo vettore caratteristico  $x^C \in S$  (tutti i componenti non-zero sono uguali) è un massimo globale per  $f_G$  in  $S$ .

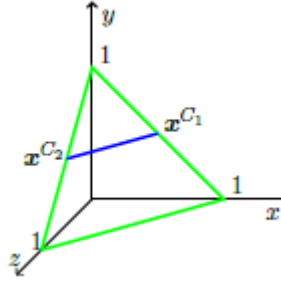
Possiamo quindi risolvere il problema della clique massima grazie a questo teorema: prima calcoliamo la cardinalità della clique massima, poi vediamo quali sono i suoi vertici. Purtroppo, però, **non tutti i massimizzatori di  $f_G$  sono nella forma di vettori caratteristici** (componenti non-zero diversi tra loro), perciò possono essere usati solo per ottenere la cardinalità della clique massima ma non per ottenere i vertici che la compongono; questi massimizzatori sono detti **soluzioni spurie**. Per esempio, consideriamo il seguente grafo:



Abbiamo due clique massime,  $C_1 = \{1, 2\}$  e  $C_2 = \{1, 3\}$ . Per il teorema di Motzkin e Straus sappiamo che i vettori caratteristici corrispondenti a queste clique sono massimi globali per  $f_G$  nel semplice  $S$ :

$$x^{C_1} = \left(\frac{1}{2}, \frac{1}{2}, 0\right) \quad x^{C_2} = \left(\frac{1}{2}, 0, \frac{1}{2}\right)$$

Da un punto di vista geometrico abbiamo quindi **due punti nel semplice** che rappresentano i suoi massimi globali:



Tuttavia possiamo vedere che sono massimi globali anche tutti i punti appartenenti al segmento che congiunge  $x^{C_1}$  a  $x^{C_2}$ ; dato però che non sono nella forma di vettori caratteristici **non possono essere usati** per estrarre informazioni sui vertici delle clique massime e sono quindi catalogati come soluzioni spurie. Il problema è stato risolto da **Bomze** (1997) quando è stata proposta una **versione regolarizzata** di  $f_G(x)$  in cui la matrice  $A'$ , al posto della diagonale nulla, presenta valori compresi tra 0 e 1, per esempio  $\frac{1}{2}$ :

$$A' = A + \frac{1}{2}I_{|V|}$$

...dove  $I$  è la matrice d'identità. Il lagrangiano regolarizzato diventa quindi:

$$\hat{f}_G(x) = x^T A' x$$

Consideriamo allora un sottoinsieme di vertici  $C \subset V$  con vettore caratteristico  $x^C$ ; il teorema rivisto da Bomze diventa:

- $C$  è una **clique massima** di  $G$  se e solo se  $x^C$  è un massimo globale di  $\hat{f}_G(x)$  in  $S$ ;
- $C$  è una **clique massimale** di  $G$  se e solo se  $x^C$  è un massimo locale di  $\hat{f}_G(x)$  in  $S$ ;
- ogni massimo locale è un vettore caratteristico (il che significa che non esistono soluzioni spurie - esattamente come volevamo).

In altre parole, in questo modo abbiamo una soluzione del problema discreto ch'è anche soluzione del problema continuo e viceversa. Ora che abbiamo descritto i termini del problema, dobbiamo risolvere il seguente sistema:

$$\max x^T A_G x \text{ con } x \in S_n$$

Dobbiamo cioè massimizzare la funzione del teorema di Bomze con  $x$  appartenente al simpleso standard. Esistono molti modi per compiere quest'ottimizzazione; quello che forse può risultare più interessante, e che ritroveremo poi nella *Teoria dei giochi*, sfrutta le cosiddette *dinamiche di replicazione*.

#### 7.5.5 Dinamiche di replicazione

Introdotte nell'ambito della *teoria dei giochi*, le dinamiche di replicazione possono essere **discrete o continue nel tempo**.

Nella versione discreta, la dinamica di replicazione è avviata da un certo punto  $x(0) = (x_1(0), \dots, x_n(0))$  appartenente al simpleso standard  $S$ ; le componenti del punto successivo (al tempo  $t + 1$ ) sono calcolate secondo questa regola:

$$x_i(t+1) = \frac{x_i(t)\pi_i(t)}{\sum_j x_j(t)\pi_j(t)}$$

...dove:

- $\sum_j x_j(t)\pi_j(t)$  è un *fattore di normalizzazione* usato per far sì che ogni nuovo punto appartenga ancora al simpleso standard  $S$  (componenti non-negative, somma pari a 1);
- la funzione  $\pi_i(t)$ , una volta implementata la rete che sfrutti la dinamica di replicazione, altro non è che il *net input* dell' $i$ -esimo neurone.



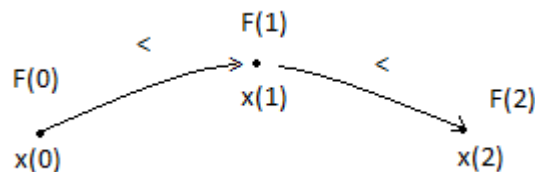
Possiamo costruire una rete neurale che sfrutti questa dinamica di replicazione. Avrà:

- $n$  neuroni, tanti quanti sono i componenti di  $x$ ; il loro **aggiornamento è sincrono**: in ciascun istante  $t$  ogni neurone aggiorna il proprio stato;  $x(t)$  raccoglie quindi gli stati di tutti i neuroni all'istante  $t$ .
- nessuna funzione di attivazione: ogni neurone prima calcola il proprio net input -  $\pi_i(t)$  - quindi lo moltiplica per  $x_i(t)$  (prima non-linearità) e lo divide per il fattore di normalizzazione, che **dipende dall'output dei rimanenti neuroni** (seconda non-linearità).
- tutte le componenti di ciascun punto  $x(t)$  saranno non-negative se la matrice dei pesi è non-negativa (si moltiplicano e si dividono quantità non-negative tra loro, ottenendo un risultato a sua volta non-negativo). Ne deriva che ogni punto della dinamica di replicazione e ogni traiettoria congiungente un punto con un altro saranno sempre **confinati all'interno del simpleso standard  $S$** .

Per capire il comportamento a lungo termine di questa rete possiamo appoggiarci al **teorema fondamentale della selezione naturale** secondo cui, se la matrice dei pesi è simmetrica ( $W = W^T$ ), allora è possibile associare al sistema una funzione  $F$ :

$$F(t) = x(t)^T W x(t) = \sum_i \sum_j w_{ij} x_i(t) x_j(t)$$

— $F$  risulta essere una funzione di Lyapunov, dato che **cresce in modo monotono** lungo ogni traiettoria della dinamica di replicazione fino a convergere ad un massimo locale, anch'esso contenuto nel simpleso standard  $S$ .



Ci troviamo in una **situazione estremamente simile** a quella vista precedentemente: la funzione  $F$  è identica alla funzione vista per il

teorema di Bomze! Se vogliamo quindi trovare la clique massimale è sufficiente costruire una rete neurale con **tanti neuroni quanti sono i vertici del grafo**<sup>3</sup> e la seguente matrice dei pesi:

$$W = A_G + \frac{1}{2}I_{|V|}$$

Per via del *teorema fondamentale della selezione naturale*, allora, questa rete neurale convergerà necessariamente ad un massimo locale contenuto nel simpleso standard  $S$  – ch'è esattamente quello che volevamo ottenere. Notare che la rete convergerà sicuramente ad un massimo *locale*, permettendoci di ottenere una clique *massimale*, non *massima*; la cardinalità di questa clique massimale, però, potrebbe essere molto vicina a quella della clique massima, se non equivalente nel caso in cui il massimo trovato sia non solo locale ma anche globale - un risultato comunque buono.

#### 7.5.6 Punti di sella

Come sappiamo, tra i punti stazionari non abbiamo solo massimi e minimi locali ma anche punti di sella; si tratta di punti in cui le derivate sono tutte 0 ma che non sono **né massimi né minimi**.

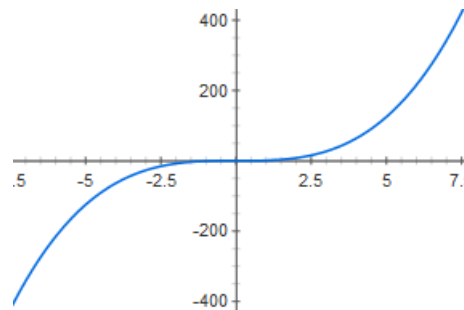


Figura 39: Esempio di punto di sella.

L'algoritmo precedente potrebbe erroneamente convergere ad un punto di sella; per evitare questo problema è sufficiente **controllare la forma del vettore associato a tale punto**: se non è nella forma di vettore caratteristico dev'essere scartato (non può esistere un punto con vettore caratteristico che non sia un massimo o un minimo).

<sup>3</sup> Otteniamo una rete che ha la **stessa topologia** del nostro grafo; l'unica differenza sta nel fatto che i neuroni avranno dei *self-loop* causati dal  $\frac{1}{2}$  presente necessariamente nella tabella dei pesi.



## PROBLEMI DI ISOMORFISMO

---

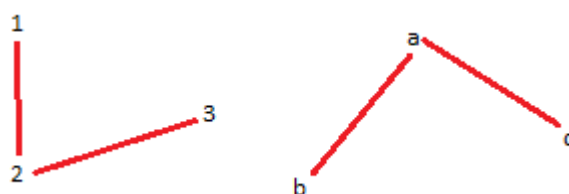
Vediamo ora come la dinamica di replicazione discreta vista precedentemente possa risolvere problemi di isomorfismo tra grafi e alberi.

### 8.1 DEFINIZIONE DI ISOMORFISMO TRA GRAFI

Supponiamo d'avere due grafi non orientati,  $G_1 = (V_1, E_1)$  e  $G_2 = (V_2, E_2)$ : per stabilire se si tratti di grafi identici ci serviamo della nozione di isomorfismo. Un isomorfismo è una qualunque **funzione biunivoca**  $\phi : V_1 \rightarrow V_2$  tale che  $(h, k) \in E_1 \Leftrightarrow (\phi(h), \phi(k)) \in E_2$ . In altre parole, esiste un isomorfismo tra grafi se:

- Hanno lo **stesso numero di vertici**;
- Ad archi del primo grafo corrispondono archi del secondo, e viceversa.

Un esempio di isomorfismo è il seguente: esiste infatti una funzione  $\phi$  che mappa vertici del primo grafo nei vertici del secondo ( $a \Leftrightarrow 2, b \Leftrightarrow 1, c \Leftrightarrow 3$ ); le corrispondenze tra gli archi sono invece  $(1, 2) \Leftrightarrow (a, b), (2, 3) \Leftrightarrow (a, c)$ ; non è presente un arco  $(b, c) \Leftrightarrow (1, 3)$ .



Determinare l'esistenza e le caratteristiche di un isomorfismo tra grafi è una questione la cui complessità è posta nella **zona grigia tra la classe di problemi NP-completi e P**: non ne esiste una soluzione in tempo polinomiale ma non vi sono nemmeno prove certe che si tratti di un problema NP-completo. Nelle prossime pagine vedremo una soluzione che sfrutta il grafo d'associazione tra due grafi.

## 8.2 SOLUZIONE AL PROBLEMA DELL'ISOMORFISMO TRA GRAFI

Siano dati due grafi non orientati,  $G_1 = (V_1, E_1)$  e  $G_2 = (V_2, E_2)$ ; il **grafo d'associazione** tra  $G_1$  e  $G_2$ , chiamato  $G = (V, E)$ , è formato da:

- **Vertici:** il prodotto cartesiano dei vertici  $V_1 \times V_2$ ; ne deriva che i vertici di  $G$  hanno due componenti, un vertice di  $G_1$  e un vertice di  $G_2$ ;
- **Archì:** dati due vertici di  $G_1$ ,  $h$  e  $k$ , e due vertici di  $G_2$ ,  $a$  e  $b$ , allora:
  1. se esiste un arco tra  $h, k$  e  $a, b$  o se *non* esiste un arco tra  $h, k$  e  $a, b$  c'è un arco in  $G$  tra i vertici  $(h, a)$  e  $(k, b)$ ;
  2. nelle situazioni spurie (solo uno dei due archi) non abbiamo nessun arco in  $G$ .

Per esempio, il grafo d'associazione dei due grafi più a sinistra è formato da  $3 \times 4 = 12$  vertici totali e da un insieme di archi la cui presenza segue le regole appena viste.

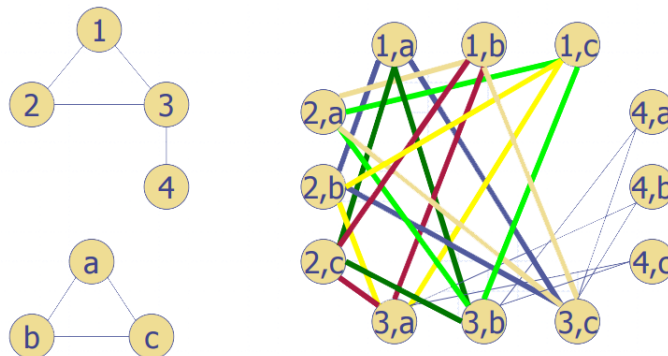


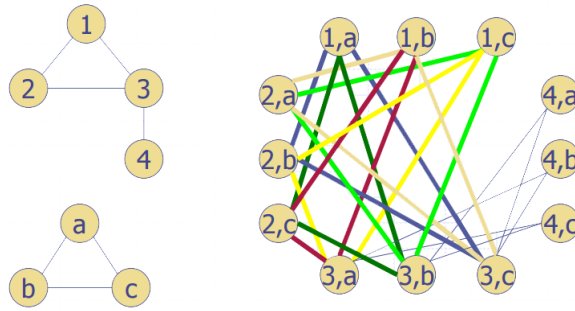
Figura 40: Esempio di grafo d'associazione tra due grafi.

Tramite il grafo d'associazione possiamo capire se due grafi siano isomorfi e quali vertici siano coinvolti nell'isomorfismo. Se supponiamo d'avere due grafi  $G_1$  e  $G_2$  di  $n$  **vertici** ciascuno, allora:

- **C'è isomorfismo?**  $G_1$  e  $G_2$  sono isomorfi se e solo se il corrispondente grafo d'associazione  $G$  presenta una **clique massima** di  $n$  **vertici**.
- **Quali vertici appartengono all'isomorfismo?** Se assumiamo che  $G_1$  e  $G_2$  siano isomorfi, allora i vertici coinvolti nell'isomorfismo sono i vertici della clique massima di  $G$ . Per esempio, se

la clique massima di  $G$  ha il vertice  $(a, 2)$  allora i vertici coinvolti nell'isomorfismo sono  $a \in G_1, 2 \in G_2$ . Esiste quindi una **corrispondenza biunivoca** tra gli isomorfismi dei due grafi  $G_1$  e  $G_2$  e le clique massime del grafo di associazione.

Bisogna sottolineare che in molte situazioni realistiche l'isomorfismo non coinvolge interi grafi ma solamente loro *sottografi*. Determinare isomorfismi tra sottografi o il massimo comun sottografo sono problemi di complessità via via crescente, tanto che è NP-completa anche l'approssimazione del problema del massimo comun sottografo. Il grafo d'associazione risulta comunque utile anche nell'isomorfismo tra sottografi. Per esempio, riprendiamo il caso visto precedentemente in cui due grafi hanno rispettivamente 4 e 3 vertici; i due non possono essere isomorfi vista la disparità di vertici ma può esistere isomorfismo tra sottografi:



Dato che le clique massime del grafo d'associazione hanno cardinalità 3, allora esisteranno isomorfismi tra tre vertici di  $G_1$  e tre vertici di  $G_2$ ; alcuni esempi sono:

$$(1, a), (2, c), (3, b); \quad (1, b), (2, c), (3, a); \quad (2, a), (3, b), (1, c);$$

### 8.3 TROVARE LA CLIQUE MASSIMA: REPLICATION DYNAMICS

Abbiamo visto finora che esiste una relazione biunivoca tra isomorfismi e clique massime del grafo di associazione. Tramite le dinamiche di replicazione possiamo individuare non solo la cardinalità ma perfino i vertici di una clique massima: possono quindi tornare utili in questo frangente. Supponiamo infatti d'avere due grafi,  $G'$  e  $G''$  e il relativo grafo d'associazione  $G$ ; se  $A$  è la **matrice d'adiacenza** di  $G$  allora è possibile trovare la o le clique massime di  $G$  usando una rete che implementi la *replication dynamic*, cioè che abbia:

- tanti nodi quanti sono i vertici del grafo d'associazione  $G$ ;

- i pesi settati a  $W = A' = A + \frac{1}{2}I_{|V|}$ , seguendo cioè la formulazione regolarizzata di Bonze.

Il sistema si avvierà su uno stato iniziale arbitrario – ad esempio il baricentro del simpleso – e convergerà ad un massimo che massimizzerà  $F_G(x) = x'Wx$  nel simpleso standard. Questo massimo corrisponderà alla clique massima nel grafo d'associazione e, di conseguenza, ci farà ottenere l'isomorfismo tra i grafi  $G_1$  e  $G_2$ .

#### 8.4 UN ESEMPIO CONCRETO: RICONOSCIMENTO DI OGGETTI

Un tipico caso in cui questa soluzione trova applicazione è il **riconoscimento d'oggetti in base alla loro forma**: si tratta cioè di determinare il tipo d'oggetto semplicemente studiandone l'ombra proiettata su un piano, scartando ulteriori informazioni di contesto quali la posizione, la scala o la rotazione dell'oggetto. Gli algoritmi di riconoscimento diffusi oggi riescono a compiere un buon lavoro anche senza queste informazioni aggiuntive; possono avere però più difficoltà nel caso vi siano **occlusioni** (un oggetto sopra l'altro) o **deformazioni non-rigide** (deformazioni dell'oggetto in cui la forma non è preservata; si pensi ad un braccio che si piega e alle conseguenti variazioni della sua forma).

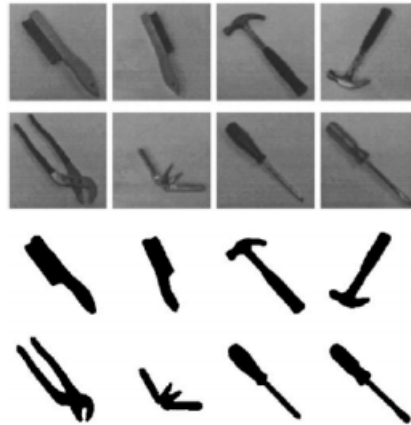


Figura 41: Oggetti e la loro corrispondente forma.

L'algoritmo più diffuso ed in grado di gestire tutte le difficoltà appena viste trasforma ogni forma in un albero; un eventuale **isomorfismo tra alberi** è indice d'appartenenza allo stessa classe di oggetti.

8.4.1 *Trasformare un oggetto in una forma: gli shocks*

Il primo passo per riconoscere un oggetto consiste, come detto, nel trasformarne la forma in un albero. Per fare ciò si usano i cosiddetti *shocks*, cerchi di raggio massimo inscritti e bitangenti i confini degli oggetti. Lo **scheletro** dell'oggetto si ottiene **unendo i centri di questi shocks**.

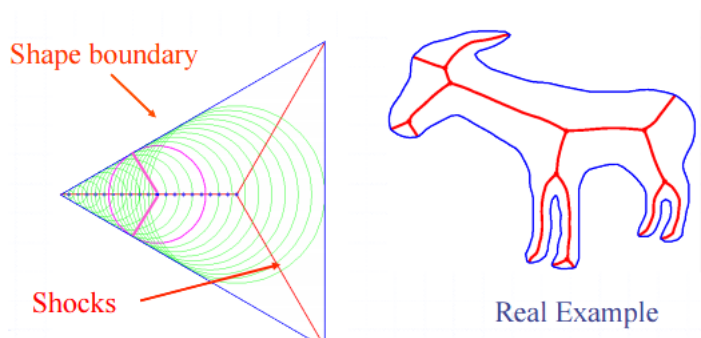


Figura 42: Shocks e scheletri in due esempi concreti.

Notare che questa rappresentazione tramite shocks, per quanto utile, **non è esente da limitazioni**: lo scheletro è infatti estremamente sensibile al rumore, cioè alle **minime perturbazioni nella forma degli oggetti**. Consideriamo ad esempio questo rettangolo e il suo shock:

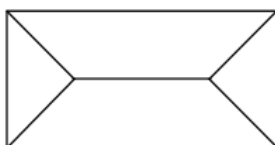


Figura 43: Lo scheletro di un semplice rettangolo.

Se un rettangolo del tutto simile a questo presenta una **piccola perturbazione della forma** otteniamo uno scheletro molto diverso, dotato di un ramo aggiuntivo; se 'dimentichiamo' che stiamo comunque lavorando su un rettangolo, paradossalmente otteniamo scheletri che ci fanno pensare a oggetti molto diversi tra loro.

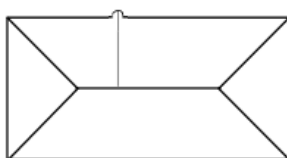
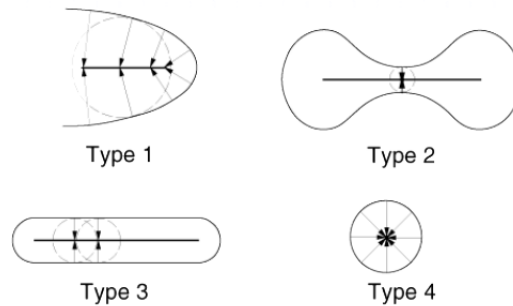


Figura 44: Piccole perturbazioni nella forma del rettangolo provocano scheletri molto diversi.



Ad ogni shock può poi essere **associato un valore numerico** (il raggio o l'istante in cui lo shock si è formato); tramite questo valore possiamo ottenere **shock di 4 diverse classi**:



- Tipo 1: il raggio progressivamente decresce o cresce monotonicamente; visivamente corrisponde a piccole protuberanze.
- Tipo 2: il raggio presenta un minimo locale (nel punto centrale della figura); visivamente corrisponde ad una rientranza.
- Tipo 3: il raggio è costante;
- Tipo 4: il singolo centro è un massimo globale della figura;

#### 8.4.2 Trasformare un oggetto in una forma: dagli shocks agli alberi

Una volta ottenuto lo scheletro di una forma è possibile **suddividerlo in segmenti** etichettati a seconda del tipo di shock e dell'istante di formazione: per esempio la sequenza gialla è etichettata con il tipo 3 e il numero di sequenza 002. L'albero è costruito usando questi segmenti, seguendo queste regole:

- ogni segmento corrisponde ad un vertice;
- segmenti adiacenti corrispondono a vertici adiacenti;
- più tardi è stata creata una sequenza, più alto è il vertice nella gerarchia dell'albero<sup>1</sup>.

Una volta ottenuti gli alberi è possibile applicare la dinamica di replicazione: dati due alberi, ne costruiamo il grafo d'associazione e ne individuiamo la clique massima, quindi verifichiamo la presenza di un isomorfismo e valutiamo quanto i due oggetti siano simili. Se

<sup>1</sup> Questo perché la tarda formazione di uno shock presuppone che esso si occupi di una perturbazione o di un dettaglio significativo dell'oggetto.
















							
	<b>12.35</b>	5.73	6.02	7.35	10.25	11.15	9.12
	4.05	<b>6.88</b>	5.22	3.27	2.84	4.18	5.95
	5.37	3.13	<b>8.40</b>	4.47	7.56	4.21	2.72
	15.18	9.02	5.44	13.19	10.18	<b>15.95</b>	13.22
	<b>21.84</b>	11.01	12.17	15.88	9.21	17.75	16.37
	<b>10.43</b>	3.41	4.19	4.00	7.26	5.69	4.96

Figura 47: Tabella indicante le distanze tra gesti diversi.



Figura 48: Clusters ciascuno contenente oggetti simili tra loro.

lavorare per riconoscere un isomorfismo tra grafi: lavorando con gli alberi, però, alcuni dettagli cambiano necessariamente. Prima di tutto vediamo alcuni concetti di base:

- Un *percorso* è una sequenza di nodi distinti e adiacenti tra loro; se il nodo iniziale coincide con il nodo finale allora abbiamo un *ciclo*.
- Un grafo è *connesso* se ogni coppia di nodi è congiunta da un percorso (posso cioè arrivare ad un nodo da qualunque altro nodo);
- La distanza tra due nodi  $u$  e  $v$  è la lunghezza del percorso più corto che li congiunge.

Per quanto riguarda gli **alberi**, invece:

- Un albero è un grafo connesso privo di cicli; può essere *radicato* se presenta un nodo di particolare rilevanza, la radice.
- Il livello di un nodo  $u$  in un albero radicato ( $\text{lev}(u)$ ) è la distanza tra  $u$  e la radice;
- Se  $u$  e  $v$  sono adiacenti e la differenza dei loro livelli è 1 ( $\text{lev}(v) - \text{lev}(u) = 1$ ) allora  $u$  è il nodo genitore di  $v$  mentre  $v$  è il figlio di  $u$ .
- Nota: in un albero, qualsiasi coppia di nodi è **connessa da un percorso univoco**.

Dobbiamo poi **ridefinire l'isomorfismo tra sottoalberi**: dati due alberi radicati  $T_1$  e  $T_2$ , un isomorfismo è qualsiasi funzione biunivoca  $\phi : H_1 \rightarrow H_2$ , con  $H_1 \subset V_1$  e  $H_2 \subset V_2$  tale che:

- se esiste un arco tra  $u, v \in H_1$  allora c'è un arco tra  $\phi(u), \phi(v)$ .
- Se  $u$  è il genitore di  $v$ , allora  $\phi(u)$  è il genitore di  $\phi(v)$ ; in questo modo la **struttura gerarchica è mantenuta** tra i due alberi.
- i sottografi indotti sono connessi, cioè sono a loro volta dei *sottoalberi*.

L'isomorfismo può poi essere sia **massimo** che **massimale**:

- Isomorfismo massimale: isomorfismo non contenuto in un altro isomorfismo;
- Isomorfismo massimo: il sottoalbero è massimale e contiene il maggior numero di matching nodes.

Abbiamo infine bisogno di modificare il grafo d'associazione quando parliamo di alberi, visto che tale grafo **non preserva la struttura gerarchica degli alberi**. Per capire perché consideriamo due alberi (fig. 49). Il grafo d'associazione costruito secondo le regole standard permetterà di ottenere una clique massima corrispondente ad un isomorfismo tra sottoalberi; verranno cioè associati  $a$  con  $3$ ,  $b$  con  $4$ ,  $c$  con  $5$ ,  $d$  con  $6$ ,  $f$  con  $7$ ,  $g$  con  $8$ ; in realtà noteremo che anche il vertice  $h$  sarà associato a  $2$  – e questo perché entrambi non sono connessi a nessun altro dei vertici visti precedentemente. La clique massima conterrà questo legame aggiuntivo che, però, **viola i vincoli gerarchici tipici degli alberi**. Dobbiamo quindi modificare il modo in cui si ottiene il grafo d'associazione. La soluzione è appoggiarci al *tree association graph*, che a sua volta si basa sul concetto di *path string*.

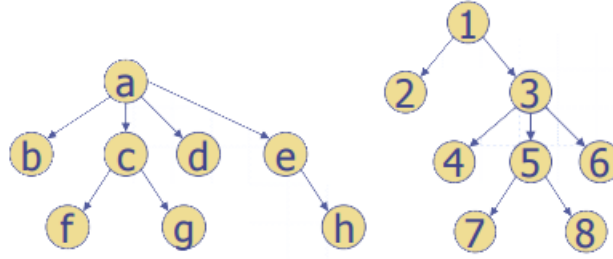


Figura 49: Due alberi: esisterà un isomorfismo?

#### 8.4.4 Path string

Supponiamo d'avere un albero radicato come quello in fig. 49 e due suoi nodi,  $e$  e  $g$ ; come sappiamo esiste un percorso unico che li congiunge,  $e = x_0 x_1 \dots x_n = g$ . Supponiamo di voler percorrere questo percorso: abbiamo un  $+1$  quando saliamo di livello e un  $-1$  quando scendiamo. Otteniamo così la stringa  $\text{str}(e, g) = -1 - 1 + 1 + 1$ . In generale, si otterranno stringhe univoche per ciascuna coppia di vertici dell'albero. Dati due alberi  $T_1$  e  $T_2$ , allora, il *tree association graph* è il grafo non orientato dove:

- $V = V_1 \times V_2$ ;
- esiste un arco tra  $u, w \in V_1$  e  $v, z \in V_2$  solo se il path string tra  $u, w$  è identico al path string tra  $v, z$ .

Una volta ottenuto il nuovo albero d'associazione possiamo procedere ad individuare clique massime o massimali e corrispondenti isomorfismi massimi o massimali. Riprendendo il nostro esempio, otteniamo le corrispondenze in fig. 50. Questi risultati sono stati ottenuti sfruttando solo le **proprietà morfologiche degli shocks**, cioè solo gli eventuali isomorfismi tra alberi, scartando altre informazioni utili: i risultati sono molto incoraggianti ma presentano qualche errore. Possiamo quindi provare a **migliorare l'esperimento** trasformando gli alberi in *attributed trees*, dove cioè ogni nodo dell'albero ha associato un vettore contenente informazioni sul nodo (la lunghezza dello shock, la curvatura dello shock, la velocità con cui si è formato, ...). Indichiamo questi nuovi alberi con  $T = (V, E, \alpha)$ , dove  $\alpha$  è una funzione che associa ad ogni nodo  $u$  un vettore di valori  $\alpha(u)$ . Per associare due *attributed trees*, allora, costruiamo una *tree association matrix pesata*, dove cioè ogni vertice ha associato un indice di similitudine tra le coppie di vertici che lo formano.

Query Shape	Top 8 Topological Matches							
	1	2	3	4	5	6	7	8

Figura 50: Risultati finali dell'esperimento.

La clique ricercata non è più la massima ma quella che **massimizza questi pesi**, tanto che molto spesso si scelgono clique con cardinalità minore ma di peso maggiore. I risultati, con molti meno errori, sono riportati in fig. 51. Notare che può succedere che si vogliano **associare alberi liberi**, cioè senza radice; è possibile farlo costruendo il *tree association graph*, senza però servirsi della nozione di *path string*: si usa invece la nozione di *distanza* (abbiamo un arco tra  $(u, v)$  e  $(1, 2)$  se la distanza tra  $u$  e  $v$  è uguale alla distanza tra  $1$  e  $2$ ). Dato che il percorso è unico per ogni coppia di vertici appartenenti ad un albero, non abbiamo casi spuri. Per il resto possiamo usare come prima una rete che implementi la dinamica di replicazione per risolvere problemi relativi a questi particolari alberi.

Top 8 topological attributed matches								
Query	1	2	3	4	5	6	7	8
	0.000	0.257	0.344	0.455	0.470	0.471	0.481	0.507
	0.000	0.257	0.384	0.416	0.482	0.527	0.529	0.530
	0.000	0.344	0.384	0.447	0.498	0.516	0.520	0.529
	0.000	0.354	0.466	0.466	0.533	0.589	0.590	0.609
	0.000	0.314	0.459	0.466	0.477	0.563	0.567	0.574
	0.000	0.314	0.354	0.459	0.524	0.553	0.557	0.562
	0.000	0.194	0.302	0.354	0.375	0.445	0.537	0.560
	0.000	0.328	0.358	0.359	0.375	0.420	0.562	0.583
	0.000	0.194	0.309	0.310	0.358	0.378	0.507	0.532
	0.000	0.378	0.403	0.404	0.420	0.445	0.481	0.482
	0.000	0.281	0.309	0.354	0.359	0.403	0.536	0.539
	0.000	0.281	0.302	0.310	0.328	0.404	0.544	0.545
	0.000	0.114	0.471	0.471	0.524	0.527	0.558	0.560
	0.000	0.114	0.470	0.476	0.529	0.538	0.564	0.565
	0.000	0.471	0.476	0.480	0.486	0.519	0.539	0.562
	0.000	0.400	0.447	0.459	0.524	0.533	0.596	0.629
	0.000	0.400	0.459	0.466	0.477	0.530	0.550	0.568
	0.000	0.095	0.126	0.555	0.562	0.567	0.572	0.575
	0.000	0.095	0.160	0.544	0.563	0.587	0.588	0.613
	0.000	0.126	0.160	0.592	0.600	0.604	0.605	0.608
	0.000	0.334	0.366	0.416	0.455	0.498	0.518	0.519
	0.000	0.185	0.334	0.486	0.520	0.529	0.529	0.558
	0.000	0.185	0.366	0.480	0.527	0.529	0.531	0.560
	0.000	0.601	0.730	0.730	0.732	0.738	0.742	0.759
	0.000	0.601	0.680	0.681	0.689	0.692	0.692	0.693

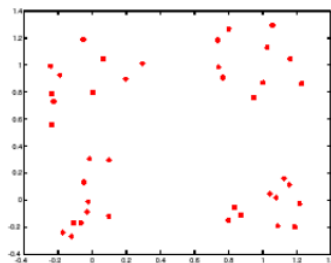
Figura 51: Risultati ottenuti usando gli attributed trees.

## CLUSTERING

Finora abbiamo esaminato problemi d'apprendimento supervisionato in cui un esperto ha il compito di etichettare gli elementi del training set che verranno poi usati nella fase d'apprendimento della rete. L'approccio opposto è l'**apprendimento non supervisionato** o *clustering*: dato un insieme di dati non classificati si vuole raggrupparli in insiemi (*cluster*) omogenei per qualche parametro. Un esempio di clustering è il seguente, in cui siamo portati ad individuare 4 diversi insiemi di punti.

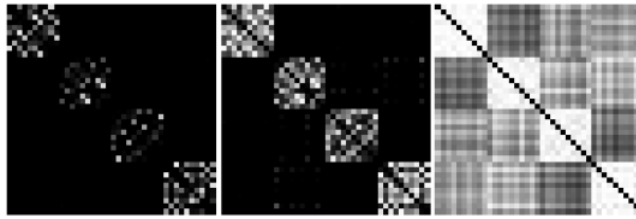
9.0.5 *L'importanza della scala*

In realtà la scala scelta influenza molto il precedente esempio, che può mostrare dai 2 ai 4 cluster. Una scala più o meno fine modifica infatti sensibilmente le affinità tra i punti, cosa che può portare ad individuare un numero molto diverso di cluster. Per esempio i seguenti punti...



...a seconda della scala scelta mostrano diverse affinità:





#### 9.0.6 Una definizione di cluster

Non esiste una definizione universale di cluster ma in genere tutti concordano nel dire che un cluster debba **soddisfare due criteri**:

- **criterio interno**: gli oggetti all'*interno* di un cluster devono essere il più possibile **simili tra loro**;
- **criterio esterno**: gli oggetti all'*esterno* di un cluster devono essere il più possibile **diversi da quelli all'interno**.

#### 9.0.7 Classificazione in base all'input

A seconda dell'**input fornito all'algoritmo di clustering** abbiamo:

- **Clustering *feature-based* o *central clustering***. Gli oggetti sono rappresentati tramite un **vettore di  $n$  features** che, a sua volta, può essere visto come un punto in uno spazio  $n$ -dimensionale. Se da un lato questo approccio permette di sfruttare gli strumenti offerti dall'analisi per lavorare su questi punti, dall'altro limita gli oggetti studiabili, dato che **non tutti sono rappresentabili** tramite un vettore di features (esempio: collezione di documenti);
- **Clustering *pairwise* o *graph-based clustering***. L'algoritmo usa una **matrice di affinità** tra gli oggetti, quadrata e (generalmente) simmetrica. Gli oggetti sono rappresentati tramite un **grafo non orientato ma pesato**, con tanti vertici quanti oggetti e archi tra oggetti simili. Non lavorando necessariamente su vettori di features il clustering pairwise è un metodo **più generale e flessibile** del precedente.

#### 9.0.8 Classificazione in base alla scala

Un'altra distinzione importante si basa sull'uso della scala:

- **Flat clustering.** Si tratta dell'approccio meno sofisticato in cui l'algoritmo di clustering restituisce semplicemente la migliore partizione possibile dei punti forniti.
- **Hierarchical clustering.** Si tratta dell'approccio più sofisticato che considera l'**influenza della scala**. Fornisce una **rappresentazione gerarchica** dei vari cluster (se la scala è fine abbiamo tanti singleton quanti punti; se la scala è grossolana abbiamo via via insiemi più numerosi). Per esempio, dati questi punti nel piano...

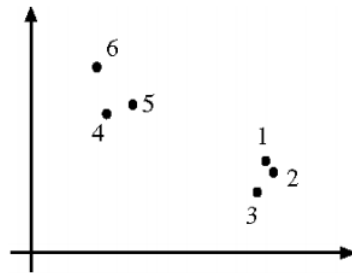


Figura 52: Un generico insieme di dati.

...la corrispondente rappresentazione gerarchica è un albero come il seguente. La sua **costruzione** può essere sia *top-down*, dalla radice alle foglie, o *down-top*.

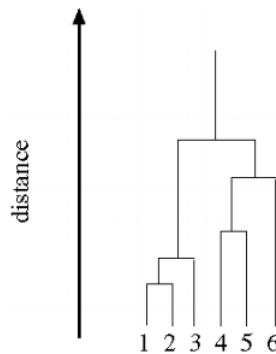


Figura 53: Rappresentazione gerarchica dell'insieme di dati.

#### 9.0.9 Classificazione in base alla modernità

Infine, possiamo distinguere tra gli approcci **classico e moderno** al clustering: il clustering classico prevede semplicemente di partizionare gli oggetti in insiemi massimamente omogenei, il cui numero è tipicamente dato come input all'algoritmo assieme ai dati che vogliamo partizionare; l'approccio più moderno, invece, estrae sequenzialmen-

te i cluster dai dati (non c'è conoscenza pregressa del loro numero) e contempla i casi in cui i cluster sono sovrapposti o alcuni elementi non appartengono a nessun cluster.

#### 9.0.10 Cenni storici

L'idea di clustering nasce nella prima metà del Novecento in ambito psicologico grazie agli studi di tre psicologi tedeschi, Wertheimer, Koehler e Koffka, a capo di un movimento detto *scuola di Gestalt*. I tre studiarono le **leggi della percezione visiva umana**, deducendo che la mente sia spontaneamente portata a raggruppare elementi che presentano proprietà comuni (*gestaltqualitat*) come la prossimità, la similarità, etc.

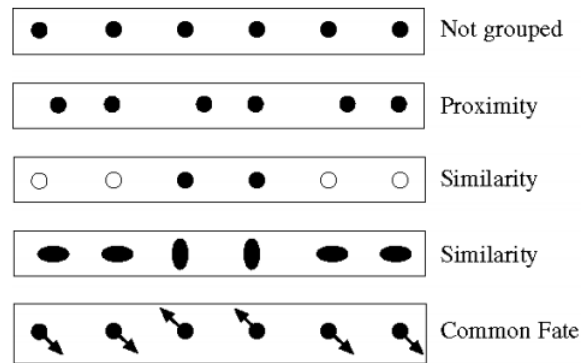


Figura 54: Alcuni esempi relativi alla capacità umana di raggruppare oggetti simili per qualche caratteristica.

Come vediamo, i 6 punti nel piano non hanno nulla che ci permetta di raggrupparli; tuttavia lievi modifiche alla loro posizione, al loro colore o alla loro forma fanno sì che l'occhio naturalmente li organizzi in tre insiemi diversi. Siamo portati a raggruppare anche elementi che presentino simmetrie, parallelismi, aree chiuse/aperte o linee continue/discrete:

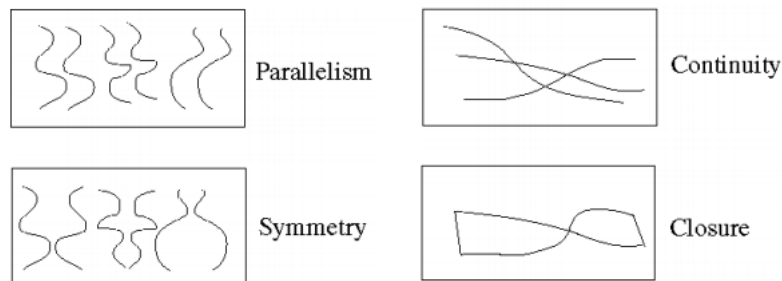


Figura 55: Altre possibili similitudini tra oggetti diversi.

Naturalmente esistono violazioni a queste regole, soprattutto se **conoscenze pregresse** forniscono informazioni che influiscono sui nostri naturali criteri di raggruppamento.

### 9.1 CLUSTERING FEATURE-BASED: K-MEANS

K-means è il più famoso algoritmo di **clustering feature-based**. Sia dato un insieme di vettori di  $n$  features  $\{x_1, \dots, x_n\}$  e il numero desiderato di cluster  $K$ . L'algoritmo di k-means allora è:

1. Scelgo in modo casuale  $K$  punti rappresentativi dei dati, detti **centroidi**;
2. **Fisso i centroidi** e assegno tutti i rimanenti punti al centroide più vicino, sfruttando ad esempio la nozione di *distanza euclidea*;
3. **Fisso gli assegnamenti** e ricalcolo i centroidi sui nuovi cluster appena ottenuti.

I passi 2 e 3 sono ripetuti finché i centroidi non cambiano, non viene prodotta alcuna variazione negli assegnamenti dei punti o dopo un certo numero di iterazioni fissato dall'utente.

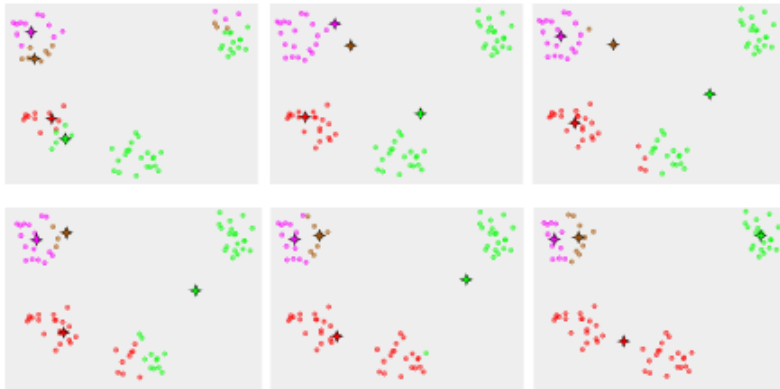


Figura 56: L'andamento dell'algoritmo di clustering.

Formalmente, K-means minimizza la somma, per ciascun cluster, delle distanze euclidee tra i centroidi ( $\mu_i$ ) e i rimanenti punti del data set ( $x_j$ ); più la funzione è bassa, infatti, più la distanza tra il centroide e i punti è minima.

$$F = \sum_{i \in \text{clusters}} \left( \sum_{j \in \text{elements of the } i\text{-th cluster}} \|x_j - \mu_i\|^2 \right)$$

K-means presenta **tre limitazioni**:

- è **spiccatamente feature-based**, cioè possiamo applicarlo solo se abbiamo a disposizione dei vettori di features su cui calcolare i centroidi;
- necessita di conoscere **in anticipo il numero di cluster**, che quasi contraddice l'idea di raggruppare i dati in base alla loro natura;
- la scelta iniziale dei centroidi potrebbe essere la **meno adatta**: potremmo cioè non trovare mai i centroidi che minimizzano la funzione  $F$  (minimi locali ma non globali).

Un esempio di clustering su un'immagine è il seguente: gli oggetti da raggruppare sono i **singoli pixel**, a cui è associato un vettore di 3 features (R, G, B).

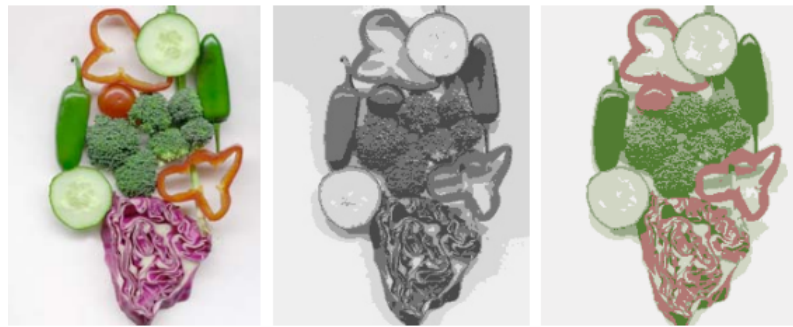


Figura 57: Esempi di clustering: nel primo caso raggruppiamo in base all'intensità, nel secondo in base al colore.

## 9.2 CLUSTERING PAIRWISE: NORMALIZED CUT

*Normalized Cut* è una forma di clustering pairwise che si serve della matrice d'affinità  $A = (a_{ij})$  per rappresentare gli oggetti da raggruppare; tali oggetti sono organizzati in un **grafo pesato non orientato** in cui:

- ogni oggetto è un vertice;
- due vertici  $i, j$  sono connessi se  $a_{ij} \neq 0$ ;
- il peso dell'arco corrisponde all'affinità tra i due nodi.

Lo scopo di normalized cut è **tagliare il grafo in due o più porzioni fortemente coese**, preservando le affinità più forti e eliminando invece quelle più deboli.

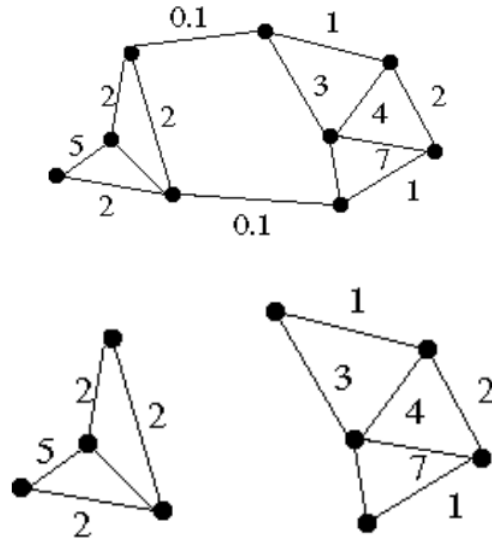


Figura 58: Tagliare un grafo in due componenti fortemente connesse.

### 9.2.1 Un primo approccio: *min-cut*

*Min-cut* è un primo algoritmo di taglio che minimizza la seguente funzione:

$$\text{cut}(A, B) = \sum_{u \in A, v \in B} w(u, v)$$

In poche parole, l'algoritmo individua due aree del grafo meno affini tra loro (A e B) cercando di **minimizzare la somma** dei pesi che congiungono tali aree. L'approccio ha il **problema** di **compiere tagli banali**, separando cioè un singolo nodo dal resto del grafo. Si ottengono così **cluster non bilanciati**, formati o da molti elementi poco coerenti tra loro o da un singolo elemento.

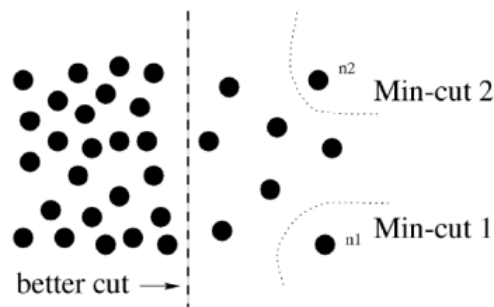


Figura 59: Due esempi di tagli banali effettuati da min-cut.

## 9.2.2 La soluzione di Normalized cut

Normalized cut propone un taglio che tiene conto **non solo della coesione interna** del cluster ( $\text{cut}(A, B)$ ) ma anche di quella esterna ( $\text{aff}(A, V), \text{aff}(B, V)$  indicano quanto l'area A o B sia legata al resto del grafo). Il taglio tra le aree A e B è quindi definito come:

$$N\text{cut}(A, B) = \frac{\text{cut}(A, B)}{\text{aff}(A, V)} + \frac{\text{cut}(A, B)}{\text{aff}(B, V)}$$

$$\text{aff}(A, V) = \sum_{u \in A, t \in V} w(u, t)$$

Anche in questo caso lo scopo è **minimizzare**  $N\text{cut}(A, B)$ . In realtà possiamo **trasformare questo problema in un problema di massimo** se ci accorgiamo che  $\text{cut}(A, B)$  è uguale alla differenza tra l'affinità di A e V ( $\text{aff}(A, V)$ ) e l'affinità interna di A ( $\text{aff}(A, A)$ ). Ne deriva:

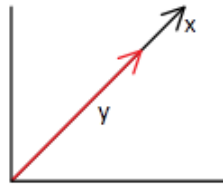
$$\begin{aligned} &= \frac{\text{aff}(A, V) - \text{aff}(A, A)}{\text{aff}(A, V)} + \frac{\text{aff}(B, V) - \text{aff}(B, B)}{\text{aff}(B, V)} \\ &= 2 - \left( \frac{\text{aff}(A, A)}{\text{aff}(A, V)} + \frac{\text{aff}(B, B)}{\text{aff}(B, V)} \right) \end{aligned}$$

Dobbiamo allora massimizzare  $\left( \frac{\text{aff}(A, A)}{\text{aff}(A, V)} + \frac{\text{aff}(B, B)}{\text{aff}(B, V)} \right)$ . Per capire come farlo dobbiamo prima introdurre il clustering attraverso autovalori e autovettori.

## 9.3 CLUSTERING PAIRWISE: APPROCCIO CON AUTOVETTORI

Si tratta di un'altra tecnica di clustering pairwise in cui si sfruttano le proprietà degli autovalori e autovettori della matrice di affinità A (**simmetrica**). Per prima cosa definiamo autovalori e autovettori.

Data la matrice quadrata  $A_n$  si definiscono autovalori e autovettori di A lo scalare  $\lambda$  e il vettore colonna  $x$  tali per cui  $Ax = \lambda x$ .



Geometricamente,  $Ax$  è un vettore che, sotto particolari circostanze, ha la stessa direzione di  $x$  e una lunghezza quantificata dallo scalare  $\lambda$ . Ci interessa sottolineare che, **quando  $A$  è simmetrica, essa ha  $n$  autovalori reali** e non complessi; abbiamo quindi la possibilità di **ordinarli**.

Supponiamo ora d'avere la matrice d'affinità  $A$ , quadrata e simmetrica; vogliamo rappresentare ogni cluster come un **vettore binario**  $x$  di  $n$  componenti (l'elemento  $x_i$  è 1 se l' $i$ -esimo elemento del data set appartiene al cluster, 0 altrimenti). La **coerenza interna del cluster** è misurabile usando una formula che già ben conosciamo:

$$x^T Ax = \sum_i \sum_j a_{ij} x_i x_j = \sum_{x \in \text{Cluster}} \sum_{y \in \text{Cluster}} a_{ij}$$

Più compatto sarà il cluster, più  $x^T Ax$  avrà un valore alto: per massimizzare  $x^T Ax$  dobbiamo quindi trovare il **cluster più compatto** tra tutti quelli ottenibili dal data set – un'operazione combinatoria su un enorme numero di possibili soluzioni. Possiamo però approssimare la soluzione al problema imponendo una **limitazione sul vettore binario**  $x$ , la cui lunghezza dev'essere necessariamente pari a 1. Il **teorema di Rayleigh-Ritz** allora ci dice che:

1. se abbiamo una matrice simmetrica  $A_{n \times n}$  a cui sono associati  $n$  autovalori reali  $\lambda_1, \dots, \lambda_n$ ;
2. se  $\lambda_{\max}$  è il maggior autovalore di  $A$ ;
3. se la lunghezza degli autovettori è fissata a 1;

...allora  $\lambda_{\max}$  massimizza  $x^T Ax$ .

$$\lambda_{\max} = \max x^T Ax \text{ con } \|x\| = 1$$

Per trovare il cluster che massimizza  $x^T Ax$  allora basta trovare il maggior autovalore della matrice  $A$  e prendere il corrispondente autovettore, il cui supporto indicherà gli elementi che fanno parte del cluster. Supponiamo, ad esempio, d'avere i seguenti punti e la matrice d'affinità  $A$  (fig. 60): la struttura sembra ben ordinata dato che i primi 10 componenti formano il primo cluster, i secondi 10 formano il secondo, .... Calcoliamo il più grande autovalore e prendiamo il corrispondente autovettore: otteniamo il cluster contenente gli ultimi



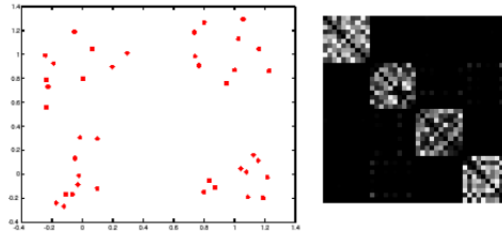


Figura 60: Punti e relativa matrice d'affinità.

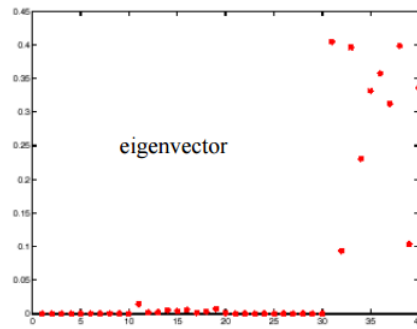


Figura 61: Risultato considerando l'autovettore corrispondente al più grande autovalore.

10 elementi della matrice d'affinità (il vettore presenta infatti 30 zeri e 10 uni).

Per estrarre poi i rimanenti cluster abbiamo due alternative:

- **Opzione banale:** scartiamo i punti associati agli elementi positivi dell'autovettore appena trovato e lavoriamo solo sui punti rimanenti ricalcolando l'autovalore massimo di  $A$  (che risulta ora contenere meno elementi), prendendo il corrispondente autovettore.
- **Opzione usata:** invece di calcolare solo il massimo autovalore di  $A$ , calcolo *tutti* gli autovalori di  $A$ ; uso il primo più grande per il primo cluster, il secondo più grande per il secondo, etc. L'unico vincolo da rispettare è che l'autovettore corrispondente al secondo, terzo, ... autovalore dev'essere **ortogonale a tutti i precedenti**.

L'**algoritmo** di clustering tramite autovettori è quindi il seguente:

1. Costruisco (o ottengo in input) la matrice d'affinità  $A$ ;
2. Calcolo tutti i suoi autovalori e autovettori;
3. Prendo l'autovettore corrispondente al più grande autovalore;

4. Pongo a 0 tutti gli elementi corrispondenti ad elementi già raggruppati, a 1 tutti gli elementi non raggruppati;
5. Se tutti gli elementi sono stati raggruppati, abbiamo sufficienti cluster.

L'algoritmo si ripete finché esiste un numero sufficiente di cluster. Quest'approccio, comunque, ha una limitazione: l'autovettore usato può avere delle **componenti negative**.

#### 9.4 LEGAME TRA NORMALIZED CUT E AUTOVALORI

I vettori binari usati fin qui per rappresentare i cluster possono essere usati anche per rappresentare le partizioni di un grafo: un vettore avrà cioè tanti valori quanti sono i vertici del grafo e, per componenti,

$$y_i = \begin{cases} +1 & \text{se } i \in A \\ -1 & \text{altrimenti} \end{cases}$$

Possiamo quindi chiederci quale sia il **vettore binario che massimizza *normalized cut***:

$$\left( \frac{\text{aff}(A, A)}{\text{aff}(A, V)} \right) + \frac{\text{aff}(B, B)}{\text{aff}(B, V)}$$

Per trovarlo, supponiamo d'avere una matrice d'affinità  $A_{P \times P}$ ; costruiamo una matrice diagonale  $D_{P \times P}$ , dove gli unici elementi diversi da 0 sono posti sulla diagonale. Tali valori sono la somma di tutti i pesi connessi a ciascun vertice:

$$d(i) = \sum_j w_{ij}$$

$$\mathbf{D} = \begin{bmatrix} d_1 & 0 & \dots & 0 \\ 0 & d_2 & \dots & 0 \\ & & \dots & \\ 0 & 0 & \dots & d_p \end{bmatrix}$$

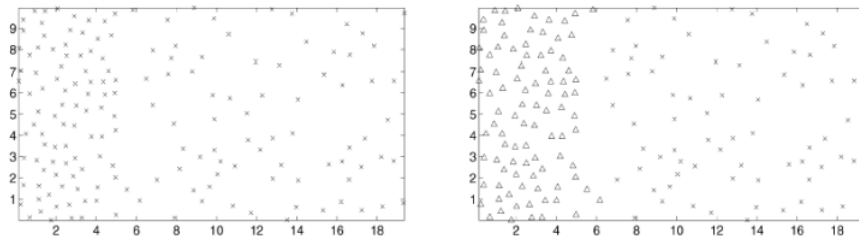
Allora è dimostrabile che trovare il vettore binario  $y$  che massimizza la *normalized cut* equivale a risolvere questo problema:

$$y = \operatorname{argmin}_x \text{Ncut}(x) = \operatorname{argmin}_y \frac{y^T (D - A) y}{y^T D y}$$

...con  $y^T D 1 = 0$ . Se **rilassiamo il vincolo su  $y$**  in modo da fargli assumere valori reali, possiamo approssimare la soluzione risolvendo l'equazione  $(D - A)y = \lambda D y$ , che è un autovettore di  $(D - A)$ . Dato ch'è dimostrabile che il più piccolo autovalore di  $(D - A)$  è sempre 0 - che corrisponde ad una partizione banale (una parte comprende tutti i vertici, un'altra l'insieme vuoto) - allora il trucco è cercare il **secondo minor autovettore**, che darà una ottima partizione del grafo. L'algoritmo per il normalized cut è allora: dato un grafo pesato  $G = (V, E)$ , dove il peso rappresenta l'affinità tra due nodi,

- Risolvo l'equazione  $(D - A)x = \lambda D x$ ;
- Uso l'autovettore il cui autovalore è il secondo più piccolo per partizionare in due parti il grafo;
- Vedo se la partizione corrente necessita di ulteriori divisioni: in caso affermativo, ripeto l'algoritmo daccapo scartando una parte dei vertici oppure scegliendo l'autovettore corrispondente al terzo più piccolo autovalore.

Per esempio, consideriamo questo grafo in cui abbiamo una regione ad alta densità e un'altra a bassa densità: l'algoritmo divide nettamente le due parti.



Un esempio più complesso ma che sostanzialmente ha la stessa struttura è il seguente:



Normalized cut è particolarmente usato nell'area dell'*image recognition* e si presta a risolvere il problema della *segmentazione d'immagine* (data un'immagine, segmentarla in porzioni i cui pixel sono il più possibile simili tra loro). Per esempio, per partizionare la seguente immagine ricca di elementi rumorosi otteniamo un risultato buono

ma non ottimo in quanto non c'è una distinzione corretta tra le parti (l'erba non è distinta dal giocatore di baseball, il giocatore a terra è suddiviso in più parti, ...). Si tratta tuttavia di errori comprensibili visto che l'algoritmo non è in grado di distinguere tra *primo piano* e *sfondo* dell'immagine. In generale normalized cut ha la tendenza a produrre **segmenti in eccesso**.



Figura 62: Esempio di segmentazione d'immagine.

## 9.5 INSIEMI DOMINANTI

Finora abbiamo visto delle tecniche di clustering in cui abbiamo fatto due forti assunzioni:

1. Il problema di clustering è un problema di miglior partizione dei dati a disposizione (cosa che esclude la possibilità di clustering sovrapposti);
2. La matrice d'affinità è simmetrica: solo in questo modo è possibile ordinare gli autovalori e far funzionare l'algoritmo di *normalized cut*.

L'approccio tramite *dominant sets* si domanda invece cosa sia formalmente un cluster: come definirne i criteri di similarità interna e dissimilarità esterna?

### 9.5.1 Esempio: cluster come clique massimale

Consideriamo uno scenario semplice in cui la matrice d'affinità  $A$  sia binaria (contenente 1 se due oggetti sono simili e 0 altrimenti)<sup>1</sup>. Otteniamo quindi un grafo non orientato e non pesato in cui abbiamo tanti vertici quanti punti e un arco solo quando l'affinità tra due vertici è 1. In questa situazione un cluster equivale alla **clique massimale** del grafo. Il concetto di clique massimale ha infatti un criterio esterno e interno:

- Interno: la clique è un sottoinsieme di vertici mutualmente adiacenti tra loro (ogni coppia di vertici è connessa da un arco) → significa che tutti i vertici sono simili tra di loro.
- Esterno: la clique non è contenuta in nessun'altra clique → significa che i vertici esterni sono dissimili dai vertici interni.

In questo caso binario, trovare i cluster di un insieme di punti significa **trovare le clique massimali del grafo associato** a tali punti. In questo modo non abbiamo bisogno di conoscere in anticipo il numero di cluster (o di clique massimali) nel grafo, come avviene invece con l'algoritmo di k-means.

### 9.5.2 Definizione formale di Dominant Set

Per dare la definizione di insieme dominante abbiamo prima bisogno di un paio di concetti preparatori. Consideriamo un grafo  $G = (V, E)$  non orientato e pesato, dove i pesi dipendono dalla matrice d'affinità  $A$ . Prendiamo un sottoinsieme di  $S \subset V$  tale per cui la somma dei pesi di ogni sottoinsieme di  $S$  è sempre positiva ( $W(T) > 0 \forall T \subset S$ ). Possiamo allora:

- vedere quanto un elemento  $i \in S$  sia **legato a S** sommando le affinità tra il vertice  $i$  e il resto dei vertici di  $S$  (normalizzando per la cardinalità di  $S$ ):

$$AvDeg_S(i) = \frac{1}{|S|} \sum_{j \in S} a_{ij}$$

- vedere se un **elemento**  $i \in S$  sia **legato più a S o più ad un generico elemento j esterno** ad  $S$ :

<sup>1</sup> Una matrice così brutale può essere ottenuta settando un valore-soglia.

$$\phi_S(i, j) = a_{ij} - \text{AvDeg}_S(i)$$

Se il risultato è positivo allora l'affinità tra  $i$  e  $j$  è maggiore, se è negativo l'affinità tra  $i$  e  $S$  è maggiore.

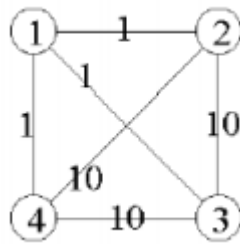
Definiamo poi un altro indice, molto utile per capire il peso (o l'**impatto**) di un generico vertice  $i$  all'interno di un insieme di vertici  $S$ . Se  $S$  è formato da un solo elemento allora il peso sarà 1; altrimenti, il peso è calcolato ricorsivamente come la somma dell'affinità tra  $i$  e  $S$  (non contenente  $i$ ) per il peso di  $j$  in  $S$  (sempre non contenente  $i$ ).

$$w_S(i) = \begin{cases} 1 & \text{se la cardinalità di } S \text{ è } 1 \\ \sum_{j \in S \setminus \{i\}} \phi_{S \setminus \{i\}}(i, j) w_{S \setminus \{i\}}(j) & \text{altrimenti} \end{cases}$$

La definizione di peso ci permette di dire che  $S$  è un insieme dominante se:

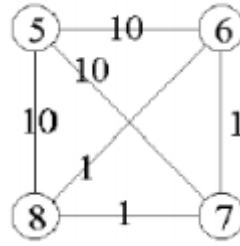
- $w_S(i) \geq 0 \forall i \in S$  (omogeneità interna: per tutti i nodi interni il peso è positivo)
- $w_{S \cup \{i\}}(i) < 0 \forall i \notin S$  (disomogeneità esterna: per tutti i nodi esterni, aggiungerli all'insieme è una pessima idea perché diminuisce la coerenza interna dell'insieme)

In altre parole: se  $i$  è estraneo all'insieme  $S$  il peso è negativo, altrimenti è positivo. Per interpretare il significato del peso consideriamo questo esempio:



Il nodo 2 è simile ai nodi 3 e 4, mentre il nodo 1 sembra essere estraneo ai tre. Infatti, se calcoliamo il peso di 1 rispetto all'insieme  $\{1, 2, 3, 4\}$  otteniamo un valore negativo, che segnala che l'aggiungere 1 **diminuisce la coerenza interna** dell'insieme. In questo secondo esempio, invece, il vertice 5 è altamente simile ai vertici 6, 7, 8: il

peso  $w_{\{5,6,7,8\}}(5)$  è infatti positivo (5 **aumenta la coerenza interna** dell'insieme).  $w$  è una misura che descrive cosa comporta, in termini di similarità, l'aggiunta o la rimozione di un nodo.



Un insieme dominante è quindi un **insieme di vertici massimamente coesi tra loro**.

### 9.5.3 Insieme dominante come problema di ottimizzazione

La **nozione di insieme dominante è equivalente a quella di cluster** e generalizza l'esempio della clique massimale visto precedentemente. Per capire perché, ricordiamo che la coesione di un cluster è misurabile come  $x^T A x$ , dove  $A$  è la funzione d'affinità. Si può dimostrare che se  $S$  è un insieme dominante, allora il suo vettore caratteristico  $x^S$  appartenente al semplice standard e  $n$ -dimensionale definito come:

$$x_i^S = \begin{cases} \frac{w_S(i)}{W(S)} & i \in S \\ 0 & \text{altrimenti} \end{cases}$$

è massimo locale stretto di  $x^T A x$  nel semplice standard. Viceversa, se  $x^*$  è massimo locale di  $x^T A x$  nel semplice standard, il suo supporto  $\sigma$  è un insieme dominante, a condizione che  $w_{\sigma \cup \{i\}} \neq 0$  per ogni  $i$  non appartenente a  $\sigma$ . Si tratta di una **generalizzazione del teorema di Motzin-Straus** visto precedentemente.

Riassumendo, per trovare un insieme dominante **basta massimizzare**  $x^T A x$ ; per farlo, ci serviamo di una rete neurale che implementi la dinamica di replicazione discreta; una volta trovato un insieme dominante abbiamo anche il corrispondente cluster. Per trovare i cluster rimanenti basta cercare iterativamente tutti gli insiemi dominanti e rimuoverli via via dal grafo, finché tutti i vertici non sono stati raggruppati.

9.5.4 Esempio: *image segmentation*

Il problema dell'*image segmentation* consiste nel decomporre una certa immagine in regioni di pixel massimamente coerenti - per esempio rappresentanti lo stesso oggetto. Possiamo considerare le regioni come insiemi dominanti: allora, trasformando l'immagine in un grafo non orientato e pesato, trovare i segmenti significa trovare un insieme dominante dopo l'altro usando le dinamiche di replicazione. Confrontiamo quest'approccio con *normalized cut*:

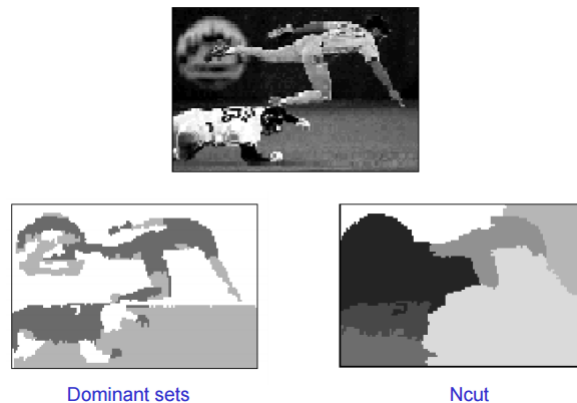


Figura 63: Vari risultati con *dominant set* e *Ncut*.

*Ncut* non riesce a distinguere il primo piano dallo sfondo mentre *dominant sets* non ha problemi a farlo; gli stessi risultati si hanno anche considerando colori o texture:

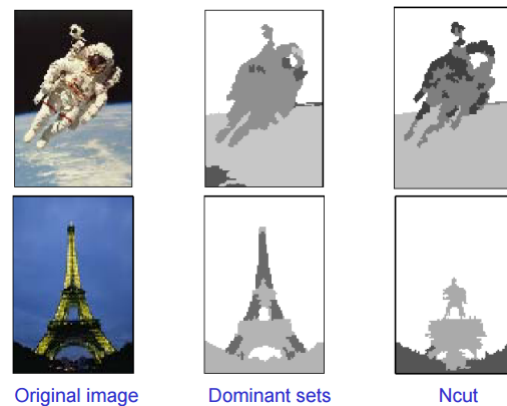


Figura 64: Anche con immagini a colori i risultati cambiano a seconda dell'approccio scelto.





## TEORIA DEI GIOCHI

---

La teoria dei giochi nasce negli anni '20 ad opera di J. Von Neumann e E. Borel. L'originale formulazione del problema prevede  $n$  giocatori o **agenti** partecipanti ad un dato gioco di cui tutti conoscono le regole (e in cui la possibilità di risultati casuali è minima); come deve giocare l' $i$ -esimo giocatore per **massimizzare il proprio beneficio**? Negli anni seguenti questa metafora venne utilizzata in **aree molto diverse** tra loro:

- 1944, 1947: pubblicazione di *Theory of Games and Economic Behavior* (J. von Neumann, O. Morgenstern);
- 1950 - 1953: contributi alla teoria dei giochi non cooperativi, ampliando l'insieme di situazioni in cui la teoria dei giochi trova applicazione (J. Nash);
- 1972 - 1982: applicazione della teoria dei giochi a problemi biologici ed evolutivisti, abbandonando di conseguenza l'idea che il giocatore sia necessariamente *razionale* (J. M. Smith);
- anni '90: sviluppo di una teoria dei giochi algoritmica.

### 10.1 GIOCHI IN FORMA NORMALE

Un gioco in forma normale è formato da:

- un numero finito di **giocatori**  $I = \{1, 2, \dots, n, n \geq 2\}$ . I giocatori devono essere almeno due;
- ciascun giocatore è dotato di un numero finito di azioni dette **strategie pure**,  $S_i = \{1, 2, \dots, m_i, m_i \geq 2\}$ . Ogni giocatore deve avere almeno due strategie pure a disposizione;
- il prodotto cartesiano delle strategie pure forma il **profilo strategico puro** del gioco,  $S = S_1 \times S_2 \times \dots \times S_n$ .
- una **funzione di payoff**,  $\pi$ , che mappa profili strategici puri a valori reali, uno per ciascun giocatore.

$$\pi : S \rightarrow \mathbb{R}^n \qquad \pi(S) = (\pi_1(S), \dots, \pi_n(S))$$

Se supponiamo d'avere 3 giocatori con le seguenti strategie pure...

$$S_1 = \{a, b, c\}; \qquad S_2 = \{0, 1\}; \qquad S_3 = \{x, y, z\}$$

...la funzione di payoff è definita come  $\pi : S_1 \times S_2 \times S_3 \rightarrow \mathbb{R}^3$  e produce tre valori,  $\pi_1, \pi_2, \pi_3$ , uno per ciascun giocatore. Notare che la funzione dipende dalle azioni di *tutti* i giocatori, non solo dall'azione del singolo agente; chiaramente ogni giocatore vuole **massimizzare il proprio beneficio**.

## 10.2 GIOCHI A DUE GIOCATORI

Nel caso speciale in cui i giocatori siano solo 2 i payoff possono essere rappresentati tramite **due matrici quadrate**, A e B:

- A contiene i payoff del primo giocatore:  $a_{ij} = \pi_1(i, j)$  con  $i \in S_1, j \in S_2$ ;
- B contiene i payoff del secondo giocatore:  $b_{ij} = \pi_2(i, j)$  con  $i \in S_1, j \in S_2$ .

Esistono diversi esempi storici di giochi a due giocatori:

- **Giochi a somma-zero**, studiati in particolare da Von Neumann. Sono giochi in cui, dato un certo profilo strategico puro, ciò che guadagna il giocatore 1 è uguale a ciò che perde il giocatore 2 (i payoff del primo giocatore sono sempre opposti al payoff del secondo giocatore). Questo significa che sommando le rispettive matrici dei payoff otteniamo 0:

$$A + B = 0$$

- **Giochi simmetrici**, in cui il ruolo di ciascun giocatore può essere scambiato senza modificare il gioco. Un classico esempio è *sasso-carta-forbice*, un gioco sia simmetrico che a somma zero. In formule questo significa che la **trasposta di B è uguale ad A**:

$$A = B^T$$

- **Giochi doppiamente simmetrici**, dove cioè la matrice di payoff di un giocatore è simmetrica a se stessa. Formalmente:

$$A = A^T = B^T$$

### 10.2.1 Celebri esempi di giochi normali

Consideriamo un **gioco simmetrico** molto famoso, il **dilemma del prigioniero**:

Supponiamo che due amici vengano arrestati per omicidio e non possano comunicare tra di loro. La polizia dice ad entrambi la stessa cosa, separatamente: non ci sono prove per l'omicidio ma possono scegliere se confessare o non confessare. Se solo uno dei due confessa, chi ha confessato evita la pena; l'altro viene però condannato a 7 anni di carcere. Se entrambi confessano, vengono entrambi condannati a 6 anni. Se nessuno dei due confessa, entrambi vengono condannati a 1 anno, perché comunque già colpevoli di porto abusivo di armi.

Il gioco può essere descritto attraverso questa matrice:

	confessa	non confessa
confessa	(6, 6)	(0, 7)
non confessa	(7, 0)	(1, 1)

Razionalmente, com'è meglio comportarsi? Mettiamoci nei panni del primo prigioniero: se l'altro confessa mi conviene confessare (6 anni di prigione invece di 7); se l'altro non confessa mi conviene comunque non confessare (nessun anno in carcere). Lo stesso vale per il secondo prigioniero: qualunque cosa decida l'amico, la strategia migliore è sempre confessare. Ma se entrambi i prigionieri seguono questi ragionamenti entrambi finiscono per passare 6 anni di carcere: la scelta razionale sembra essere la più sbagliata!

Un secondo esempio interessante è la **battaglia dei sessi**.

Immaginiamo d'avere una coppia: il marito sarebbe massimamente felice se potesse andare a una partita di football.

La moglie vorrebbe invece andare all'opera. Inoltre entrambi preferirebbero andare nello stesso luogo piuttosto che in posti diversi. Se essi non possono comunicare, in quale luogo dovrebbero andare?

Il gioco può essere rappresentato con questa matrice:

	Opera	Calcio
Opera	(2, 1)	(0, 0)
Calcio	(0, 0)	(1, 2)

Ciò che importa, in questo caso, è che i due giocatori scelgano la stessa strategia per non essere separati. Un ultimo esempio è quello del gioco **sasso-carta-forbice**, di cui conosciamo bene le regole (carta > sasso > forbice > carta). La matrice in questo caso è:

	Sasso	Forbice	Carta
Sasso	(0, 0)	(1, -1)	(-1, 1)
Forbice	(-1, 1)	(0, 0)	(1, -1)
Carta	(1, -1)	(-1, 1)	(0, 0)

### 10.3 STRATEGIE MISTE

Immaginiamo che l' $i$ -esimo giocatore giochi molte volte a *sasso-carta-forbice*: se nella maggior parte dei casi sceglie *sasso* l'altro giocatore può capire la sua strategia e usare conseguentemente *carta*, vincendo. In generale è quindi preferibile scegliere tra le strategie pure  $S_i$  a disposizione basandosi su una **distribuzione di probabilità**.

Una strategia mista è una distribuzione di probabilità sull'insieme delle strategie pure, indicata dal vettore  $x_i$ ; tale vettore è formato da tante componenti quante sono le strategie del giocatore  $i$ , ciascuna indicante la probabilità che la corrispondente strategia pura venga usata dal giocatore. Tale probabilità è sempre non-negativa: l'insieme delle strategie pure a cui è assegnata una probabilità diversa da 0 forma il **supporto** di  $x_i$ :

$$\sigma(x_i) = \{h \in S_i : x_{ih} > 0\}$$

È interessante notare come ogni strategia, sia essa pura o mista, **appartenga sempre al simpleso standard**:

- Trattandosi di una distribuzione di probabilità, ogni strategia mista è necessariamente non-negativa; la somma delle probabilità ammonta inoltre a 1;
- Una volta scelta una strategia pura, essa ha probabilità 1 mentre le rimanenti 0: sono quindi non-negative e la loro somma è 1.

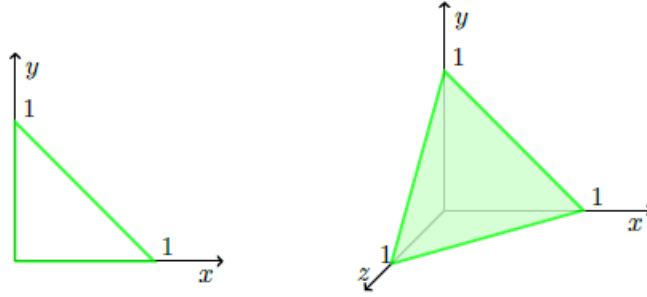


Figura 65: A sinistra il semplice a due dimensioni, a destra quello a tre dimensioni. Le strategie pure corrispondono ai vertici del semplice, le strategie miste sono punti interni al semplice.

Ogni giocatore ha una propria strategia mista: il vettore contenente le strategie miste di tutti gli  $n$  giocatori è detto **profilo strategico misto**,  $x = (x_1, \dots, x_n)$ . Per quanto riguarda il **payoff atteso**<sup>1</sup>, esso si basa sulla probabilità che un certo profilo strategico puro  $s$  (contenente la strategia pura scelta da ciascun giocatore) venga scelto quando viene giocato un profilo strategico misto  $x$ :

$$x(s) = \prod_{i=1}^n x_{is_i}$$

Il **payoff atteso** del giocatore  $i$  è quindi la somma, per tutte le possibili strategie pure, del prodotto tra la probabilità di usare la strategia  $s$  per il payoff ottenuto usando la strategia  $s$ :

$$u_i(x) = \sum_{s \in S} x(s) \pi_i(s)$$

#### 10.4 BEST REPLIES

Nella classica formulazione di gioco, l' $i$ -esimo giocatore si aspetta che gli avversari facciano la loro mossa, ma non sa di quale mossa si

<sup>1</sup> Stiamo ragionando su possibili strategie, perciò il payoff non è certo!

tratti visto che la scelta dipende da una distribuzione di probabilità. Supponiamo però che  $i$  conosca le mosse dei giocatori opposti: qual è la **miglior risposta che può dare alle strategie scelte dagli avversari**? Sia dato il vettore  $z$  contenente le strategie miste  $z_i$  dei vari giocatori:

$$z = (z_1, z_2, \dots, z_n)$$

Indichiamo con  $(x_1, z_{-1})$  la sostituzione della strategia mista  $z_1$  con  $x_1$ . Ora, supponiamo che il giocatore  $i$  sappia quali sono le strategie usate dai rimanenti giocatori,  $x_{-i}$ ; allora  $x_i^*$  è la miglior risposta a  $x_{-i}$  se comunque io scelga il giocatore  $i$ , il payoff medio che il giocatore  $i$  ottiene giocando  $x_i^*$  è **maggiore del payoff medio** ottenuto usando qualsiasi altra possibile strategia.

$$u_i(x_i^*, x_{-i}) \geq u_i(x_{ij}, x_{-i})$$

Se ad esempio scopro che il mio singolo avversario sceglie la strategia *sasso*, allora la migliore strategia sarà *carta*. Notare che la risposta migliore non è necessariamente univoca – anzi, possono esservi un numero infinito di best replies. In particolare:

- quando il supporto di una best reply include due o più strategie pure, una loro combinazione è una best reply;
- se due strategie pure sono individualmente best replies, una loro combinazione è una best reply.

## 10.5 EQUILIBRI DI NASH

Un equilibrio di Nash è una **configurazione di strategie miste** – una per giocatore – tale per cui nessun giocatore è incentivato a cambiare la propria strategia (supponendo che nemmeno gli altri giocatori la cambino). Formalmente, il vettore  $x = (x_1, \dots, x_n)$  è un equilibrio di Nash se è la **miglior risposta a se stesso**, ossia se per ogni giocatore  $i$ :

$$u_i(x_i, x_{-i}) \geq u_i(z_i, x_{-i})$$

...dove:

- $u_i(x_i, x_{-i})$  è il payoff atteso da  $i$  quando  $i$  gioca la strategia  $x_i$  appartenente all'equilibrio e i rimanenti giocatori giocano strategie sempre appartenenti all'equilibrio;

- $u_i(z_i, x_{-i})$  è il payoff atteso da  $i$  quando  $i$  gioca una strategia non appartenente all'equilibrio e  $i$  rimanenti giocatori giocano strategie appartenenti all'equilibrio.

Per trovare un equilibrio di Nash consideriamo il seguente esempio in cui due giocatori hanno 4 e 3 possibili strategie rispettivamente. Vediamo quali sono le *best replies* per ciascuna possibile strategia:

- **Giocatore 2** sceglie *left*, giocatore 1 sceglie *bottom*; giocatore 2 sceglie *middle*, giocatore 1 sceglie *low*; giocatore 2 sceglie *right*, ...;
- **Giocatore 1** gioca *top*, giocatore 2 sceglie *middle*; giocatore 1 gioca *high*, giocatore 2 gioca *left*, ...

		Player 2		
		Left	Middle	Right
Player 1	Top	3, 1	2, 3	10, 2
	High	4, 5	3, 0	7, 4
	Low	2, 2	5, 4	12, 3
	Bottom	5, 6	4, 5	9, 7

Nash equilibrium!

Figura 66: Un equilibrio di Nash in un gioco a due giocatori.

L'equilibrio di Nash è invece (5,4), indicato dalla freccia. Infatti:

- Se il giocatore 2 sceglie *middle*, la miglior scelta per il giocatore 1 è *low* (con altre scelte diminuirei il mio beneficio);
- Se giocatore 1 gioca *low*, la miglior scelta per il giocatore 2 è *middle* (con altre scelte, di nuovo, diminuirei il mio beneficio).

Naturalmente esistono situazioni in cui si hanno più di un equilibrio di Nash, come ad esempio nel gioco della *battaglia dei sessi*, o nessun equilibrio di Nash, come nel caso di *carta-sasso-forbice*:

L'importanza della teoria di Nash deriva principalmente dal fatto che Nash non si limitò a postulare l'esistenza dei propri equilibri ma



		You		
		Rock	Scissors	Paper
Me	Rock	0, 0	1, -1	-1, 1
	Scissors	-1, 1	0, 0	1, -1
	Paper	1, -1	-1, 1	0, 0

Figura 67: Sasso-carta-forbice: nessun equilibrio di Nash.

dimostrò che se consideriamo lo spazio delle strategie miste, **ogni gioco finito in forma normale ammette sempre almeno un equilibrio di Nash**. In altre parole, se tutti i giocatori giocano secondo una certa distribuzione di probabilità e senza mostrare preferenze allora esiste necessariamente almeno un equilibrio di Nash. La tecnica di dimostrazione applicava alcuni risultati di Topologia, trovando una corrispondenza biunivoca tra l'esistenza degli equilibri di Nash e punti stazionari.

#### 10.6 GIOCHI EVOLUTIVI

La **teoria dei giochi evolutivi** è stata introdotta da J. M. Smith tra gli anni '70-'80 per **modellare l'evoluzione del comportamento animale** utilizzando i principi della teoria dei giochi. Si basa sulle seguenti assunzioni:

- Si analizza una **popolazione grande** di individui appartenenti alla stessa specie, che competono per una **risorsa limitata**;
- Il conflitto è modellato come un **gioco simmetrico a due giocatori**, scelti casualmente dalla popolazione;
- Gli individui **non giocano razionalmente** ma seguono dei pattern pre-programmati; dato che la **riproduzione è asessuata**, ogni nascituro è pre-programmato esattamente come il progenitore.

- il **payoff** esprime la **fitness darwiniana** o il **successo riproduttivo** dei giocatori.

In questo framework, una *strategia mista* non è più semplicemente una distribuzione di probabilità sulle strategie a disposizione dei giocatori, ma può essere interpretata in due modi **equivalenti tra loro**:

1. ogni individuo gioca una strategia pura, ma parte della popolazione segue una strategia, il resto ne segue un'altra;
2. ogni individuo gioca una particolare strategia mista (ogni individuo è geneticamente configurato per scegliere da un certo insieme di opzioni con certe probabilità).

#### 10.7 STRATEGIE EVOLUTIVAMENTE STABILI (ESS)

Assumiamo d'avere una serie di giocatori appartenenti ad una popolazione (tutti giocano la stessa strategia mista  $x$ ). Poniamo che una percentuale di invasori, indicata con  $\epsilon \in (0, 1)$ , cominci a giocare una nuova strategia  $y$ . La strategia  $x$  è evolutivamente stabile se **resiste all'invasione della nuova strategia**, cioè se il payoff di un gioco in questa popolazione mista ( $u(y, w)$ ) è equivalente al payoff in un gioco con una sola strategia mista ( $u(x, w)$ ), con  $w = \epsilon y + (1 - \epsilon)x$ . Formalmente, una strategia  $x \in \Delta$  è evolutivamente stabile se per ogni strategia  $y \in \Delta$  diverse da  $x$  esiste un *upper bound*  $\delta \in (0, 1)$  tale per cui, per ogni  $\epsilon \in (0, \delta)$  abbiamo:

$$u[x, \epsilon y + (1 - \epsilon)x] > u[y, \epsilon y + (1 - \epsilon)x]$$

...dove il primo termine è il payoff di un generico giocatore nella popolazione in cui vi è la strategia  $x$ , il secondo è il payoff di un generico giocatore nella popolazione invasa della strategia  $y$ . Possiamo inoltre dimostrare che una strategia  $x$  è evolutivamente stabile se e solo se verificano **entrambe queste condizioni**:

- $u(y, x) \leq u(x, x)$  per ogni strategia  $y$  (equilibrio di Nash);
- $u(y, x) = u(x, x) \Rightarrow u(y, y) < u(x, y)$  per ogni strategia  $y$  diversa dalla strategia  $x$  (condizione di stabilità).

Ciò significa che una strategia è evolutivamente stabile se è un equilibrio di Nash con una verificata condizione di stabilità.

Abbiamo visto precedentemente che ogni gioco in forma normale ha almeno un equilibrio di Nash. Possiamo quindi concludere che abbia anche una strategia evolutivamente stabile? La risposta è no: può mancare infatti la condizione di stabilità. Dal **punto di vista computazionale**, infine, dire se un gioco simmetrico a due giocatori ha un ESS è NP-hard e coNP-hard; dire se  $x$  è ESS di un gioco simmetrico a due giocatori è coNP-hard.

#### 10.8 DINAMICHE DI REPLICAZIONE: CASO CONTINUO

Abbiamo già incontrato le dinamiche di replicazione nella loro versione discreta; concentriamoci ora sulle continue, usate nel contesto della teoria dei giochi evolutivi per applicare il concetto di *fitness* darwiniana. L'idea è semplice: **le strategie il cui payoff è maggiore della media sono destinate a diffondersi nella popolazione**; le strategie il cui payoff è inferiore alla media sono invece destinate a scomparire nel tempo.

Formalmente, sia  $x_i(t)$  la popolazione che condivide la stessa strategia pura  $i$  al tempo  $t$ ; lo stato della popolazione al tempo  $t$  è:

$$x(t) = (x_1(t), \dots, x_n(t))$$

Allora il rapporto seguente<sup>2</sup> è proporzionale a:

$$\frac{dx_i(t)}{x_i(t)} \propto \text{payoff of pure strategy } i - \text{average population payoff} \quad (2)$$

...ed avrà valori positivi se il payoff di  $i$  è maggiore della media, negativo altrimenti. Formalmente, se  $x_i(t)$  la popolazione che condivide la stessa strategia pura  $i$  al tempo  $t$  e se  $A = (a_{ij})$  la dinamica di replicazione continua è:

$$dx_i(t) = x_i[u(e^i, x) - u(x, x)] = x_i[(Ax)_i - x^T Ax]$$

...dove il payoff della strategia pura  $i$  è  $(Ax)_i$  mentre il payoff medio dell'intera popolazione è  $x^T Ax$ . Notare che il **simplexso standard è invariante rispetto a questa dinamica**: ciò significa che se la dinamica descrive una traiettoria che rientrerà sempre nei limiti del simplexso standard.

<sup>2</sup> Notare che la derivata è rispetto al tempo  $t$ .

### 10.8.1 Rapporto tra dinamiche di replicazione e ESS

Secondo il teorema di Nachbar, Taulor e Jonker, se il punto di partenza di una dinamica di replicazione continua appartiene all'interior del semplice standard (tutti i componenti sono strettamente positivi) allora se la dinamica converge ad un punto stazionario allora quel punto è un equilibrio di Nash. Viceversa, se abbiamo un equilibrio di Nash, allora esiste una traiettoria nell'interior del semplice standard che convergerà su di esso. C'è quindi una relazione biunivoca tra equilibri di Nash e punti di convergenza di *interior trajectories*.

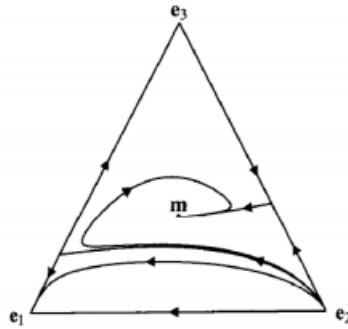


Figura 68: Dinamica appartenente all'interior del semplice.

Inoltre, se  $x \in \Delta$  è una ESS, allora  $x$  è un **punto stabile** d'equilibrio per la dinamica di replicazione - proprio come gli attrattori stabili visti precedentemente. Non vale ovviamente viceversa. Ciò significa che se perturbiamo la traiettoria sul punto stabile essa **tornerà a convergere** su tale punto (come a dire che se aggiungiamo una nuova strategia in una popolazione, nel tempo la strategia evolutivamente stabile **tornerà ad essere la dominante**).

### 10.8.2 Giochi doppiamente simmetrici e dinamiche di replicazione

Abbiamo visto che nei giochi doppiamente simmetrici la matrice dei payoff  $A$  è simmetrica,  $A = A^T$ ; in questo caso il payoff medio della popolazione  $f(x) = x^T A x$  è una funzione di Lyapunov, ossia cresce strettamente lungo la dinamica di replicazione fino a raggiungere un punto stazionario. Formalmente,  $\frac{df(x(t))}{dt} \geq 0$  per ogni  $t \geq 0$  con uguaglianza solo se  $x(t)$  è un punto stazionario. La controparte discreta di questa dinamica di replicazione è, come sappiamo:

$$x_i(t+1) = x_i(t) \frac{A(x(t))_i}{x(t)^T A x(t)}$$

Si può vedere che molte proprietà delle dinamiche di replicazione continue valgono anche per le dinamiche di replicazione discrete – se vogliamo quindi usare le dinamiche di replicazione, quindi, possiamo affidarci ad una delle due versioni, senza senza limitazioni.

#### 10.9 CLUSTERING E STRATEGIE EVOLUTIVAMENTE STABILI

Abbiamo precedentemente visto che esiste una corrispondenza biunivoca tra cluster e *dominant set*; in particolare abbiamo osservato che, in caso di matrice d'affinità binaria, i dominant set si riducono a clique massimali. Queste conclusioni derivano dal fatto che i dominant set e le clique massimali verificano i criteri di similarità interna e dissimilarità esterna tipici dei cluster. Vediamo ora come i dominant set siano in **corrispondenza biunivoca anche con le strategie evolutivamente stabili**, quando però la matrice  $A$  è arbitraria (valori reali ma **non necessariamente simmetrica**).

I dominant set sono insiemi di vertici omogenei e coesi; la nozione di peso,  $w$ , permette di capire se l'aggiunta di un nuovo nodo aumenti o diminuisca la coesione interna dell'insieme di vertici. Dimentichiamoci ora di queste definizioni e consideriamo il seguente gioco di clustering:

- Assumiamo d'avere un insieme d'oggetti,  $O$ , e una matrice d'affinità  $A$  per gli elementi di  $O$ . Tale matrice **non è simmetrica**.
- Due giocatori competono selezionando simultaneamente un elemento di  $O$ ; mostrano quindi la loro scelta e ricevono un payoff proporzionale all'affinità tra gli oggetti scelti dal giocatore (che può essere diversa dato che  $A$  non è simmetrica).
- Ciascun giocatore massimizza il proprio beneficio scegliendo un elemento fortemente affine agli oggetti che l'avversario selezionerà.

Traducendo, in questo *gioco di clustering* gli oggetti corrispondono alle strategie pure, la matrice di affinità coincide con la matrice di similarità; i due giocatori sono invece scelti casualmente da una popolazione. La nozione di **strategie evolutivamente stabili è quindi equivalente alla nozione di dominant set e viceversa**; se la matrice d'affinità è binaria e simmetrica, inoltre, la corrispondenza è con la clique massimale. Per riassumere, valgono le seguenti corrispondenze:

- A binaria, simmetrica  $\Leftrightarrow$  insieme dominante = clique massimale;
- A simmetrica, reale  $\Leftrightarrow$  insieme dominante =  $\max_{x \in \Delta} f(x) = x^T A x$ ,  $x \in \Delta$
- A arbitraria, reale  $\Leftrightarrow$  insieme dominante = ESS. questo significa che il supporto (valori strettamente positivi) di una ESS restituisce i vertici del dominant set.

### 10.9.1 Grado d'appartenenza al cluster

Supponiamo di avviare una dinamica di replicazione da un certo punto  $x(0)$  - in genere il baricentro del simpleso standard  $\Delta$ :

$$x(0) = \left( \frac{1}{n}, \dots, \frac{1}{n} \right) \in \Delta$$

Supponiamo che la dinamica di replicazione converga nel punto  $x(1) \in \Delta$ . Per estrarre il corrispondente dominant set basta mappare le corrispondenze; ad esempio, se...

$$x(1) = (0, 0.1, 0.2, 0, 0.6, 0, 0.1)$$

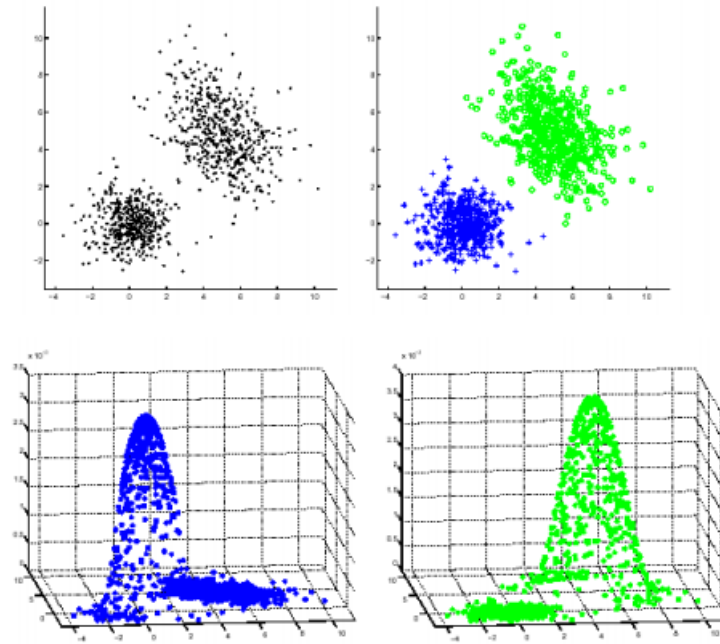
...allora ne prendiamo il supporto<sup>3</sup> e otteniamo un dominant set contenente i seguenti vertici:

$$\text{DominantSet} = \{2, 3, 5, 7\}$$

Potremmo però chiederci: perché alcuni valori del supporto sono **maggiori di altri**? Le osservazioni empiriche mostrano che non tutti gli elementi appartenenti ad un cluster sono ugualmente coesi ad esso: più è alto il valore nel supporto più l'elemento è coeso al cluster. In alcune applicazioni può allora aver senso controllare non solo il supporto, ma anche la distanza di ciascun punto dal centroide del cluster. Per fare un esempio, quando l'algoritmo individua il cluster blu gli elementi più lontani dal centroide assumono via via un valore minore. Lo stesso dicasi per il cluster verde.

Questa dinamica, molto interessante, è mostrata da pochi algoritmi di clustering; uno tra questi è naturalmente *dominant set*.

<sup>3</sup> In realtà molto spesso non si lavora con valori realmente pari a 0; si sceglie quindi un valore-soglia e si costruisce il supporto a partire da esso.



Un'altra osservazione interessante è che la tecnica del *dominant set* **non funziona su strutture allungate**, che vengono in genere segmentate in più dominant set separati; il motivo è semplice: la distanza degli elementi più a sinistra dagli elementi più a destra è tale da indurre l'algoritmo a segmentare, anche quando i punti appartengono in realtà allo stesso segmento. Esistono diversi **modi per risolvere il problema**:

- Applicazione della dinamica di replicazione sul **grafo di chiusura transitiva**. Dato  $G = (V, E)$ , il grafo di chiusura transitiva  $G' = (V, E')$  contiene gli stessi vertici di  $G$ ; ha invece un arco tra  $u, v$  in  $G$  solo se esiste un percorso tra  $u$  e  $v$  in  $G$ . Il grafo di chiusura transitiva è anche ottenibile calcolando tutte le potenze della matrice d'affinità  $A$  del grafo originale: la matrice  $A_{\text{closed}}$  ottenuta sommando tutte le potenze è esattamente la chiusura transitiva di  $G^4$ .
- Uso dei **pesi sui percorsi tra vertici**. Supponiamo d'avere due vertici,  $i$  e  $j$ , connessi da diversi percorsi (i vari pesi indicano la dissimilarità tra i vertici). Per ogni percorso tra  $i$  e  $j$  consideriamo il peso maggiore; quindi prendiamo il minimo tra tutti questi pesi. Per esempio, se abbiamo 3 percorsi, i pesi massimi sono 3, 6, 8; scelgo quindi il minimo, 3, che rappresenta il gap

<sup>4</sup> Questo deriva dal fatto che le potenze  $2, \dots, k$  della matrice d'adiacenza  $A$  sono equivalenti ai percorsi di lunghezza  $2, \dots, k$  tra due dati nodi  $i, j$ .

tra i due vertici. Possiamo quindi costruire una nuova matrice d'adiacenza, su cui poi avviare la dinamica di replicazione.

#### 10.10 CLUSTER SOVRAPPOSTI

I cluster sovrapposti sono oggi materia di studio - si pensi ad esempio allo studio dei *social networks*; non sono tuttavia individuabili tramite *normalized cut* o *k-means* per trovarli. Dominant set è invece in grado di trovarli: se ad esempio consideriamo di nuovo il grafo non orientato con matrice d'affinità binaria, più clique massimali possono avere vertici in comune, producendo quindi cluster con elementi in comune. Le strategie di *peel-off*, dove si scartano i nodi già raggruppati, non sono invece utilizzabili per chiari motivi. Notare ch'è possibile modificare il grafo di partenza in modo da far sì che la dinamica di replicazione discreta **converga ogni volta su una clique massimale diversa**: è sufficiente avviarla su un nuovo grafo, identico al precedente se non per un **nuovo nodo aggiuntivo**, che viene collegato ai vertici della clique massimale trovata precedentemente. Questo nuovo grafo non avrà più la precedente clique massimale come attrattore della dinamica; le altre clique massimali, invece, **resteranno potenziali punti di convergenza** per la dinamica di replicazione. Per trovare quindi tutti i cluster sovrapposti basta ripetere via via la dinamica aggiungendo di volta in volta un vertice.

#### 10.11 CLUSTERING GERARCHICO

Possiamo ottenere un cluster gerarchico con *dominant set*? La risposta è sì: basta considerare l'originale matrice d'affinità  $A$  con diagonale nulla e aggiungere sulla diagonale il parametro  $-\alpha$ , che funziona da **parametro di scala**: se il valore è più grande del più grande autovalore della matrice allora tutti i dati appartengono allo stesso cluster. Cambiando  $\alpha$  si cambia la scala. Concludendo, finora abbiamo considerato solamente relazioni pairwise tra oggetti; in molte applicazioni si va però oltre questo concetto: la similarità è spesso tra tre o più oggetti, sconfinando quindi verso gli ipergrafi.





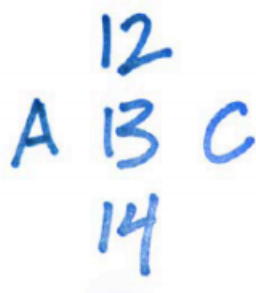
## APPLICAZIONI DELLA TEORIA DEI GIOCHI

---

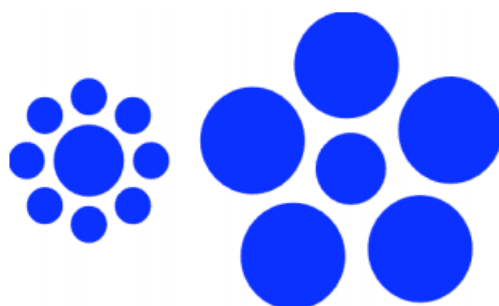
La teoria dei giochi trova applicazione anche in contesti diversi dal clustering, come nel caso del **problema dell'etichettatura consistente**. Vedremo anche alcuni concetti sull'**apprendimento semi-supervisionato**, occupandoci della **trasduzione di grafi**.

### 11.1 IL RUOLO DEL CONTESTO

In molte situazioni le *informazioni di contesto* risultano fondamentali per poter interpretare correttamente ciò che abbiamo davanti; un classico esempio è il seguente, in cui l'interpretazione dell'elemento centrale cambia a seconda del contesto in cui si trova.

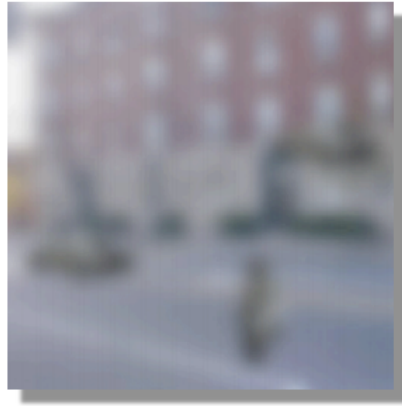


Le informazioni di contesto possono però anche **confondere**, come avviene nella seguente illusione ottica (il cerchio centrale è della stessa dimensione in entrambi gli esempi).



Anche le nostre **conoscenze pregresse** funzionano da informazioni di contesto: siamo infatti portati a pensare che la forma verticale della

fotografia posta più sotto rappresenti un pedone quando si tratta, in realtà, di un'automobile ruotata di 90 gradi.



Le neuroscienze si sono interessate delle informazioni di contesto, scoprendo che aree specifiche della corteccia cerebrale si occupano della loro analisi. Possiamo però trattare il problema delle informazioni di contesto matematicamente? Finora abbiamo assunto di poter estrarre quante più informazioni da ogni singolo oggetto, ottenendo un vettore utile per la classificazione; quest'approccio, però, può risultare non solo complesso (non sempre è semplice estrarre informazioni) ma anche molto limitato, in quanto **non considera mai le informazioni di contesto**. Dobbiamo quindi assumere un nuovo punto di vista in cui l'oggetto è più del proprio vettore d'informazioni, ma è anche **influenzato dal contesto**.

#### 11.2 PROBLEMA DELL'ETICHETTATURA

Assumiamo d'avere un insieme  $B$  contenente  $n$  oggetti e un insieme  $A$  di  $m$  etichette. Scopo del problema è etichettare ogni elemento di  $B$  con una etichetta di  $A$ . Per farlo, abbiamo **due fonti d'informazione**:

- **informazioni locali** estratte dagli oggetti, equivalenti al classico vettore di informazioni associato a ciascun oggetto;
- **informazioni di contesto**, espresse attraverso una matrice  $R_{n^2 \times m^2}$  contenente i cosiddetti **coefficienti di compatibilità**. Tali coefficienti indicano la **compatibilità tra due ipotesi**: se ad esempio il coefficiente  $r_{ij}(\lambda, \mu)$  contiene la compatibilità tra le ipotesi l'oggetto  $b_i$  è etichettato  $\lambda$  e l'oggetto  $b_j$  è etichettato  $\mu$ .

Supponiamo che inizialmente esista un sistema che estrae automaticamente le features di ogni oggetto e, sulla loro base, assegna a cia-

scun oggetto un vettore di  $m$  probabilità (quant'è probabile la prima etichetta, quant'è probabile la seconda, ..., quant'è probabile la  $m$ -esima etichetta). Otteniamo per ciascun oggetto il seguente vettore: trattandosi di probabilità ogni suo componente è non negativo e la somma dei componenti è 1.

$$p_i^{(0)} = (p_i^{(0)}(1), \dots, p_i^{(0)}(m))^T$$

Ripetiamo l'operazione per ognuno degli  $n$  oggetti, ottenendo altrettanti vettori; concateniamoli assieme ottenendo **un unico vettore**,  $p^{(0)}$ : tale vettore è un primo iniziale assegnamento di etichette. A partire da esso possiamo compiere un **processo di rilassamento delle etichette** che aggiorna iterativamente  $p^{(0)}$  basandosi sulla matrice di compatibilità  $R$ , cioè **servendosi delle informazioni di contesto**. La regola d'aggiornamento scelta nel 1976 da Rosenfeld, Hummel e Zucker è la seguente:

$$p_i^{(t+1)}(\lambda) = \frac{p_i^{(t)}(\lambda) q_i^{(t)}(\lambda)}{\sum_{\mu} p_i^{(t)}(\mu) q_i^{(t)}(\mu)}$$

dove  $q_i^{(t)}(\lambda)$  è interpretabile come il supporto fornito al tempo  $t$  dalle informazioni di contesto all'ipotesi che  $b_i$  sia etichettato  $\lambda$ . Più in dettaglio, il supporto è il prodotto tra la compatibilità che  $b_i$  ha etichetta  $\lambda$  e qualsiasi altra ipotesi sull'etichettatura di  $b_i$  (si scorre su  $\mu$ ), moltiplicata per la probabilità che  $b_i$  sia etichettato con  $\mu$ .

$$q_i^{(t)}(\lambda) = \sum_j \sum_{\mu} r_{ij}(\lambda, \mu) p_i^{(t)}(\mu)$$

Tornando alla prima formula, è interessante notare che se  $p_i^{(t)}(\lambda)$  e  $q_i^{(t)}(\lambda)$  sono entrambi alti, la probabilità che l'etichetta al tempo  $t+1$  sia  $\lambda$  aumenta, altrimenti diminuisce. Il denominatore è invece un *fattore di normalizzazione* usato affinché le varie probabilità rientrino sempre nell'intervallo  $(0, 1)$ . Fin dalla loro introduzione, gli algoritmi di rilassamento delle etichette hanno trovato applicazione in *computer vision* e *pattern recognition*.

### 11.2.1 Etichettatura consistente

Nell'83 Hummel e Zucker svilupparono un'ulteriore teoria in merito al problema della *consistenza* dell'etichettatura. In generale, un'assegnamento si dice consistente se l'etichetta assegnata a ciascun oggetto

riceve il **più alto supporto dal contesto**; geometricamente, supponiamo d'avere un punto  $p \in \Delta$ ; un'etichetta è consistente se il vettore di supporto  $q$  punta nella direzione opposta rispetto a tutti i vettori tangenti in  $p$ , cioè se punta al di fuori del semplice standard. Se invece  $q$  punta verso il semplice standard, possiamo proiettarlo sul semplice ottenendo un assegnamento migliore del precedente  $p$ . Formalmente,

$$\sum_{\lambda} p_i(\lambda) q_i(\lambda) \geq \sum_{\lambda} v_i(\lambda) q_i(\lambda) \quad i = 1, \dots, n$$

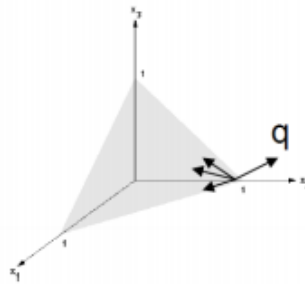


Figura 69: Interpretazione geometrica dell'etichettatura consistente.

Se la disuguaglianza stretta è valida per ogni  $v \neq p$  allora abbiamo una **consistenza stretta**. Riassumendo:

- Se in un certo momento dell'evoluzione del sistema il vettore  $q$  punta nella direzione opposta del semplice standard allora è possibile fermare la dinamica (significa che  $p$  prende in considerazione in modo consistente tutte le informazioni di contesto);
- Se invece riusciamo a proiettare il vettore  $q$  sul semplice, questo restituisce un nuovo punto migliore del precedente in termini di consistenza delle informazioni di contesto.

Possiamo inoltre dire se un certo assegnamento d'etichette  $p$  è consistente: verifica infatti queste proprietà:

1.  $q_i(\lambda) = c_i$  se  $p_i(\lambda) > 0$
2.  $q_i(\lambda) \leq c_i$  se  $p_i(\lambda) = 0$

per  $c_i$  una qualche costante non negativa.

## 11.2.2 Rilassamento d'etichettatura e polymatrix games

Il problema d'etichettatura consistente può essere visto come un gioco in cui:

- I giocatori sono gli oggetti da etichettare;
- Le strategie pure sono le etichette;
- Le strategie miste sono gli assegnamenti pesati;
- La matrice di payoff corrisponde alla matrice di compatibilità.

Esiste quindi una **corrispondenza biunivoca tra equilibri di Nash e etichettatura consistente**.

## 11.2.3 Analogie con le dinamiche di replicazione

Torniamo indietro all'algoritmo standard di rilassamento delle etichette: possiamo notare delle similitudini con le dinamiche di replicazione. La *replicated dynamics* vista precedentemente è infatti...

$$p_i^{(t+1)} = \frac{p_i^{(t)} q_i^{(t)}}{\sum_j p_j^{(t)} q_i^{(t)}}$$

...mentre il *label relaxation* è:

$$p_i^{(t+1)} = \frac{p_i^{(t)} q_i^{(t)}(\lambda)}{\sum_{\mu} p_i^{(t)}(\mu) q_i^{(t)}(\mu)}$$

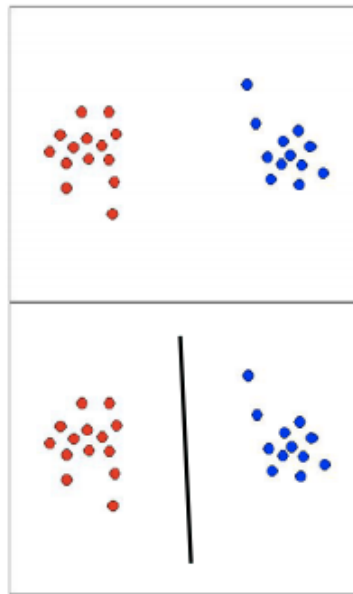
La differenza è cruciale: le dinamiche funzionano all'interno del semplice standard mentre le probabilità funzionano all'interno del **prodotto di diversi semplici standard** (lavoriamo infatti su una matrice dove la somma di *ogni* riga è pari a 1; se abbiamo  $n$  oggetti, quindi, abbiamo  $n$  semplici standard). Ne deriva ch'è chiaro perché questo particolare tipo di dinamiche viene chiamato nelle teorie evolutive *multi-population replicated dynamics*: invece di avere una sola popolazione ne abbiamo molteplici ( $n$ ) e le associazioni avvengono tra rappresentanti di popolazioni diverse.

## 11.3    APPRENDIMENTO SEMI-SUPERVISIONATO

I problemi di apprendimento semi-supervisionato sono strategie intermedie in cui solo una ridotta porzione dei dati a disposizione è etichettata dall'esperto. I problemi di apprendimento supervisionato possono quindi essere espressi in questi termini: *come posso usare l'informazione che ho a disposizione per etichettare anche i dati rimanenti?* Formalmente, sia dato un insieme di dati così suddiviso:

- Dati etichettati:  $\{(x_1, l_1), \dots, (x_m, l_m)\}$ ;
- Dati non etichettati:  $\{x_{m+1}, \dots, x_n\}$ .

Ciascun dato è rappresentato come un vertice di un grafo i cui archi rappresentano l'affinità tra i corrispondenti punti. Lo scopo è allora propagare le informazioni disponibili dai nodi etichettati ai nodi non etichettati (**trasduzione del grafo**). Notare che il tutto si basa sulla *cluster assumption*, che possiamo capire con il seguente esempio. Supponiamo d'avere due cluster, separati da una retta:



Supponiamo ora che questi siano gli unici punti etichettati a fronte di un insieme di dati molto più esteso: modifichiamo la nostra partizione come segue. Siamo comunque portati a dire che i punti appartenenti a ciascun cluster abbiano la stessa etichetta.

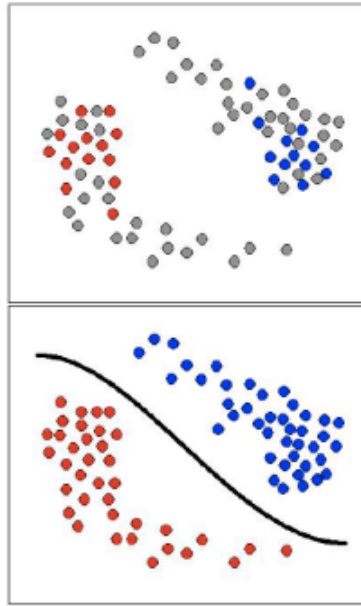


Figura 70: La cluster assumption.

#### 11.3.1 *Image segmentation interattiva*

La trasduzione del grafo trova applicazione nel campo dell'immagine segmentation, in cui una data immagine fornita è segmentata in più parti. Molte spesso *Ncut* o *dominant set* producono un numero troppo elevato di segmentazioni, soprattutto se lo stesso oggetto presenta aree con diversi colori. Un algoritmo basato sull'apprendimento semi-supervisionato - in cui l'esperto etichetta pixel di colori diversi allo stesso modo - ha molte possibilità in più di funzionare.



Figura 71: Un esempio di buona segmentazione.

#### 11.3.2 *Trasduzione di grafi non orientati e non pesati*

Un semplice esempio di trasduzione di grafo è quello che coinvolge **un grafo  $G$  non orientato e non pesato** la cui matrice d'adiacenza è



binaria (0 o 1). In tale grafo, allora, la presenza di un arco denota la **perfetta similarità tra due vertici**. Supponiamo che un esperto etichetti alcuni nodi rossi e altri blu: l'algoritmo propaga questa etichettatura ai rimanenti vertici. Visto che la presenza di un arco denota l'affinità massima, ne deriva che il grafo è partizionato in componenti connesse che condividono la stessa etichetta (come segue dalla *cluster assumption*).

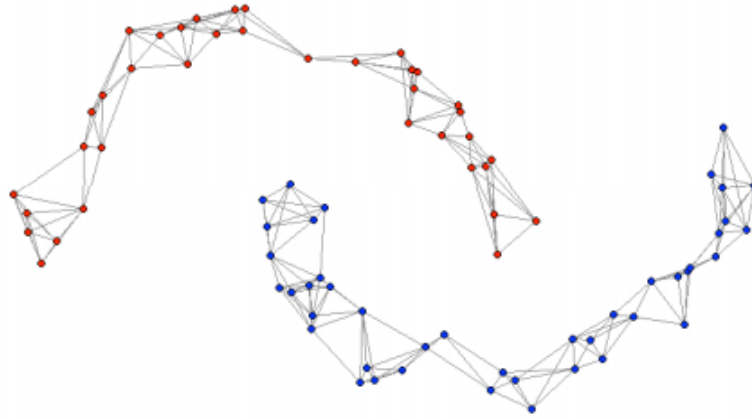


Figura 72: Due componenti connesse aventi le stesse etichette.

Per generalizzare quest'esempio al caso reale possiamo tradurre il problema in un gioco e individuarne l'equilibrio di Nash che, come sappiamo, è in corrispondenza biunivoca con l'etichettatura consistente. In questo gioco abbiamo:

- Giocatori che corrispondono ai vertici del grafo; gli *effettivi giocatori* del gioco sono quelli rappresentanti i **vertici non ancora etichettati**.
- Strategie pure corrispondenti, per ciascun vertice/giocatore, alla possibile etichettatura.

In questo gioco le etichette dei giocatori già etichettati e la relativa affinità con essi funzionano da *bias* sulle scelte dei giocatori non etichettati. Assumendo che siano possibili solo interazioni tra coppie di vertici otteniamo un **polymatrix game** che può essere risolto usando l'algoritmo standard di rilassamento dell'etichetta. Se poi **troviamo un Equilibrio di Nash** in questo gioco, troviamo anche un'etichettatura consistente dei vertici.

## Parte II

### APPENDICE



## STATISTICAL LEARNING THEORY

---

La *statistical learning theory* (SLT) è stata ideata negli anni '70 da due studiosi russi, Vapnik e Chervonenkis, e utilizzata poi negli anni '90 in *machine learning*. Incidentalmente, negli anni '60 avvenne una rivoluzione in campo statistico: se inizialmente era la *statistica parametrica* di Fischer a dominare il campo (dato un fenomeno, si assume l'esistenza di una certa distribuzione di probabilità su di esso di cui è necessario scoprire i parametri come la varianza, la media, etc), a partire dagli anni '60 si sviluppò una **statistica non-parametrica** in cui le inferenze sono fatte senza compiere assunzioni sul fenomeno. Statistical learning theory fa appunto parte di questo nuovo capitolo della statistica.

### A.0.3 Il setup standard

SLT risolve problemi di **apprendimento supervisionato** e si interessa sia di problemi di classificazione che di regressione. Dati:

- $X$ , lo spazio delle features d'input;
- $Y$ , lo spazio delle etichette d'output. Sia assume per semplicità che tale spazio sia binario,  $Y = \{+1, -1\}$ :  $+1$  indica che un oggetto appartiene ad una certa classe,  $-1$  che non vi appartiene.

allora lo scopo di SLT è scoprire le **dipendenze funzionali tra le features di  $X$  e le etichette di  $Y$** , individuando cioè la funzione  $f$  detta **classificatore**:

$$f : X \rightarrow Y$$

Per individuare questa funzione abbiamo accesso ad un *training set* contenente dati già etichettati:

$$(X_1, Y_1), \dots, (X_n, Y_n) \in X \times Y \quad (3)$$

Un algoritmo di classificazione è una procedura che prende in input il training set e restituisce esattamente  $f$ . SLT assume che esista una distribuzione di probabilità congiunta  $P$  su  $X \times Y$ ; inoltre, gli

esempi del training set sono estratti indipendentemente da  $P$ , con una procedura casuale. Non sono infine fatte assunzioni su  $P$  (la sua distribuzione è sconosciuta nella fase d'apprendimento; è solo assunto che non cambi nel tempo).

#### A.0.4 Perdite e rischi

Per misurare la **qualità di un classificatore** definiamo la *funzione di perdita* che misura il costo del classificare un'istanza di  $x \in X$  con l'etichetta  $y \in Y$ . La più semplice funzione di perdita è la **0-1 loss**:

$$\ell(x, y, f(x)) = \begin{cases} 1 & \text{se la predizione è errata} \\ 0 & \text{altrimenti} \end{cases}$$

Il **rischio** di una funzione è la perdita media, in accordo con la distribuzione di probabilità  $P$ :

$$R(f) = E(\ell(x, y, f(x)))$$

Idealmente, il miglior classificatore è quello che **minimizza il rischio**  $R(f)$ . Il classificatore che ha la migliore performance è il **classificatore di Bayes**:

$$f_{\text{Bayes}}(x) = \begin{cases} +1 & \text{if } P(y = 1|X = x) \geq 0.5 \\ -1 & \text{altrimenti} \end{cases}$$

Restituisce cioè 1 se la probabilità che l'etichetta sia 1 quando l'oggetto è  $x$  è maggiore o uguale a 0.5, -1 altrimenti. In pratica, però, è impossibile calcolare il classificatore di Bayes direttamente in quanto la distribuzione di probabilità  $P$  è sconosciuta - e l'idea di stimare  $P$  in genere non funziona (grandi quantità di dati, molte dimensioni, etc).

#### A.1 TEOREMA DI BAYES

Supponiamo d'avere dell'evidenza ( $e$ ) e di voler stimare quanto l'ipotesi  $h$  sia probabile, data quell'evidenza. Possiamo usare il fondamentale **teorema di Bayes**:

$$p(h|e) = \frac{P(e|h)p(h)}{p(e)}$$

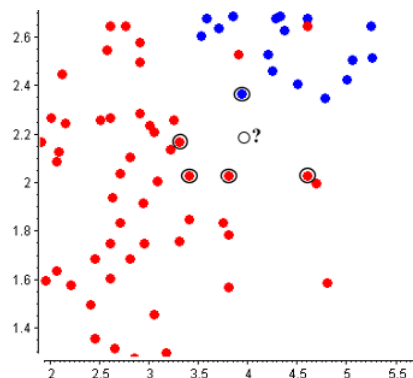
Dove:

- $P(h)$  è la probabilità a priori dell'ipotesi  $h$ ;
- $P(h|e)$  è la probabilità a posteriori dell'ipotesi  $h$  alla luce dell'evidenza di  $e$ ; è in genere facile da calcolare.
- $P(e|h)$  è quanto l'evidenza  $e$  supporti l'evidenza  $h$ ; è in genere complicata da calcolare, da cui l'importanza del teorema.

Ora, ritornando al problema della classificazione, supponiamo d'avere un training set di punti, estratti secondo le regole viste precedentemente, da una distribuzione  $P$  sconosciuta, e una funzione di perdita. Come determinare il classificatore  $f$  senza conoscere  $P$ ? Sembreremmo bloccati, ma c'è invece un modo per aggirare il problema.

#### A.1.1 Nearest neighbour (NN)

Si tratta di un metodo ideato negli anni '60<sup>1</sup>. Assumiamo d'avere un training set così rappresentato nello spazio euclideo:



Come classificare il punto centrale, non ancora etichettato? Secondo la regola del *nearest neighbour* la classe del punto ancora non etichettato è quella del **punto ad esso più vicino**. Il risultato è **tassellare l'intero spazio** euclideo ottenendo quello che viene chiamato **spazio di Voronoi**. Sorprendentemente, NN ha un'ottima performance: Cover e Thomas riuscirono a dimostrare che il rischio di NN, quando il training set contiene un numero di elementi tendente a infinito, è:

<sup>1</sup> Già citato, naturalmente in termini diversi, dallo studioso medievale Alhazen.

$$R(f_{\text{Bayes}}) \leq R_{\infty} \leq 2R(f_{\text{Bayes}})$$

In altre parole quando il training set è molto grande l'errore commesso da NN non può essere più grande di due volte l'errore del Bayes classifier, cioè del migliore classificatore disponibile. Ovviamente, a seconda della distribuzione latente  $P$ , entrambi i valori-limite possono essere raggiunti. Esistono poi delle **variazioni di NN**:

- $k$ -NN rule: usiamo i  $k$  vicini del punto da classificare e scegliamo l'etichetta in base ad un voto di maggioranza;
- $k_n$ -NN rule: come prima, solo che  $k_n$  cresce al crescere del numero di elementi del training set,  $n$ .

Infine, se  $n, k \rightarrow \infty$  (o in altre parole  $\frac{k}{n} \rightarrow 0$ ) allora a prescindere dalla probabilità di distribuzione  $P$  il rischio di  $k_n$ -NN tende al rischio del classificatore di Bayes (detto anche *universally bayes consistent*):

$$R(k_n - \text{NN}) \rightarrow R(f_{\text{Bayes}})$$

## A.2 EMPIRICAL RISK MINIMIZATION (ERM)

Si tratta di una teoria per il *pattern recognition* formata da un aspetto qualitativo (contenente le condizioni necessarie e sufficienti affinché il principio funzioni) e quantitativo (contenente le performance numeriche dell'approccio). Il funzionamento è il seguente: invece di cercare una funzione che minimizzi il rischio  $R(f)$  (detto *true risk*, sconosciuto visto che non possiamo conoscere la probabilità  $P$ ), possiamo cercare la **funzione che minimizza l'errore empirico**, cioè l'errore compiuto sul training set. Se il nostro training set contiene  $n$  campioni:

$$R_{\text{emp}}(f) = \frac{1}{n} \sum_i \ell(X_i, Y_i, f(X_i))$$

Formalmente, dato un training set  $(X_1, Y_1), \dots, (X_n, Y_n)$ , lo spazio delle funzioni  $\mathcal{F}$  e una funzione di perdita, definiamo il classificatore  $f_n$  come:

$$f_n = \operatorname{argmin}_{f \in \mathcal{F}} R_{\text{emp}}(f)$$

Questo approccio è chiamato *principio di induzione di ERM* e la sua motivazione viene dalla legge dei grandi numeri. Dobbiamo però porci una domanda: dato che dobbiamo fissare la classe delle funzioni per utilizzare ERM, come possiamo trovare la *miglior* funzione appartenente a quello spazio? Per rispondere a questa domanda dobbiamo introdurre il concetto di **VC dimension**, una misura di complessità di una funzione classificatrice. Il risultato è che  $\mathcal{F}$  non dev'essere troppo ricco, dove la ricchezza è proprio misurata dalla sua *VC dimension*.

### A.2.1 Stima e approssimazione

Idealmente vorremmo minimizzare la differenza  $R(f_n) - R(f_{\text{bayes}})$ , con  $n \rightarrow \infty$ . Se denotiamo con  $f_{\mathcal{F}}$  il **miglior classificatore di  $\mathcal{F}$**  allora possiamo dividere la sottrazione precedente in due parti, l'*estimation error* (detto anche *varianza*) e l'*approximation error* (altrimenti conosciuto come *bias*):

$$R(f_n) - R(f_{\text{bayes}}) = (R(f_n) - R(f_{\mathcal{F}})) + (R(f_{\mathcal{F}}) - R(f_{\text{bayes}}))$$

Dobbiamo quindi distinguere due problemi separati:

- l'*estimation error* è sostanzialmente quanto dista la funzione scelta  $f_n$  dal migliore classificatore di  $\mathcal{F}$ . Dipende dal training set.
- l'*approximation error* misura invece quanto dista la miglior funzione di  $\mathcal{F}$  dal classificatore di Bayes; non c'è nessuna dipendenza dal training set.

Il comportamento di questi due termini varia in base alla complessità del *function space*:

- Se la complessità è bassa – e cioè consideriamo uno spazio povero di funzioni – rischiamo *underfitting* (*estimation error* basso, *approximation error* grande);
- Se invece la complessità è alta abbiamo *overfitting* (grande *estimation error*, basso *approximation error*).



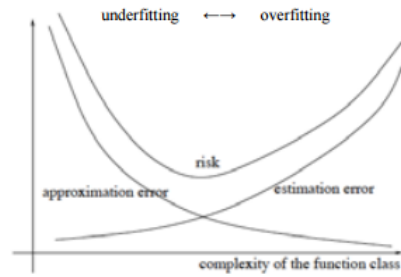


Figura 73: Esempi di underfitting e overfitting.

Dobbiamo quindi trovare un *function space* né troppo complesso né troppo semplice. Ma come possiamo misurare la complessità di tale spazio? Un primo criterio può essere il numero di parametri della funzione; la teoria sviluppata da Vapnik e Chervonenkis va in relata ancora più in profondità, ideando la *VC dimension*. Per capirla dobbiamo prima introdurre la nozione di *shattering*.

### A.3 SHATTERING

Un insieme di istanze  $X_1, \dots, X_n$  dallo spazio d'input  $\mathcal{X}$  si dice *shattered* da una funzione  $\mathcal{F}$  se tutte le etichette possibili di queste istanze (per un totale di  $2^n$ ) possono essere generate usando funzioni appartenenti a  $\mathcal{F}$ . Per esempio, se  $\mathcal{F}$  contiene tutte le funzioni lineari allora possiamo vedere che ogni set di 3 punti non-collineari "shatter"  $\mathcal{F}$ .

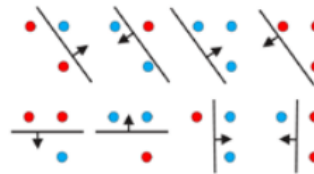


Figura 74: Esempio di shattering di 3 punti non-collineari.

Lo stesso non avviene con 4 punti, dove sono invece necessarie funzioni non-lineari.



Figura 75: 4 punti non possono essere "shattered".

Ora, una VC dimension di una classe di funzioni  $\mathcal{F}$ , denotata con  $VC(\mathcal{F})$ , è l'intero  $h$  più grande tale per cui esiste un campione

di dimensione  $h$  che è "shattered" da  $\mathcal{F}$ . se campioni di dimensioni arbitraria sono shattered, allora  $VC(\mathcal{F}) = \infty$ . Per fare qualche esempio:

- $\mathcal{F}$  = funzioni lineari in  $\mathbb{R}^2$ , allora  $VC(\mathcal{F}) = 3$ ;
- $\mathcal{F}$  = funzioni lineari in  $\mathbb{R}^n$ , allora  $VC(\mathcal{F}) = n + 1$ ;
- $\mathcal{F}$  = perceptron multistrato di pesi  $W$ , allora  $VC(\mathcal{F}) = O(W \log W)$ ;
- $\mathcal{F}$  = classificatori NN, allora  $VC(\mathcal{F}) = \infty$ .

K. Popper era particolarmente interessato dal calcolo della complessità di un modello, e pensava che il numero di parametri fosse una misura di complessità di un modello. La VC dimension non è però correlata al numero di parametri liberi di un modello.

### A.3.1 Un ulteriore risultato fondamentale

Vapnik e Chervonenkis sono responsabili anche di un ulteriore risultato fondamentale: comunque io scelga una funzione  $f \in \mathcal{F}$  con probabilità di almeno  $1 - \delta$ , il suo *true error* è sempre minore o uguale al rischio empirico più la *VC dimension* di  $\mathcal{F}$ . In altre parole il *true error* dipende dalla **complessità della funzione**.

$$R(f) \leq R_{\text{emp}}(f) + \sqrt{\frac{h(\log(2n/h) + 1) - \log(\delta/4)}{n}}$$

Questa formula connette l'errore effettivamente prodotto da un classificatore nello spazio delle funzioni  $r(f)$  che non conosceremo mai, non conoscendo  $P$ ; Vapnik e Chervonenkis riuscirono a provare che esiste un *upper bound* a questo valore. Inoltre è stato dimostrato che, con probabilità vicina a 1, qualunque sia la distribuzione di probabilità sconosciuta se  $n \rightarrow \infty$  l'*expected error* della funzione individuata da ERM approssima il valore minimo dell'*expected error* delle funzioni in  $\mathcal{F}$  se e solo se  $\mathcal{F}$  ha una *VC dimension* finita. Ciò significa che se la *VC dimension* della classe di funzioni è finito è possibile minimizzare l'*expected error* delle funzioni in  $\mathcal{F}$ . Chiaramente ERM s'interessa dell'*estimation error* ma non dell'*approximation error*, cioè dell'unico errore che considera il training set. Il modello ottimale si ottiene quindi cercando un equilibrio tra l'*empirical risk* e la *VC dimension* della classe di funzioni  $\mathcal{F}$ . L'idea di base della *structural risk minimization* è:

- Costruire una struttura innestata per una famiglia di classi di funzione di VC dimension via via crescente:  $VC(\mathcal{F}_1) \subset VC(\mathcal{F}_2) \subset \dots$ ;
- Trovare, per ciascuna classe, la funzione che minimizza l'*empirical risk*.
- Scegliere la classe (e la corrispondente funzione) che minimizza il *risk bound* visto precedentemente, formato cioè da *empirical risk* e *VC dimension*.



Figura 76: VC dimension, empirical risk e casi di overfitting/underfitting.

Si riesce quindi a trovare una **funzione-compromesso** che evita sia i casi di overfitting che i casi di underfitting.

BIBLIOGRAFIA

---

1. A. Casini, M. Tempesta, *Dispensa di Intelligenza Artificiale*, 2014.
2. M. Pelillo, *Slides del corso intelligenza artificiale*, 2014. <http://www.dsi.unive.it/pelillo/Didattica/>.
3. P. Norvig, S. Russel, *Artificial Intelligence, A Modern Approach*. Pearson, third edition, 2001.
4. Appunti e registrazioni delle lezioni del corso.