

LEZIONE 1

Cos'è l'intelligenza artificiale?

Qualcosa di complesso e articolato che ha un svolgimento lungo e approfondito.

È un progetto con un punto di inizio e un obiettivo: il punto di inizio è coincidente con l'inizio

dell'informatica, infatti si considera l'iniziatore dell'informatica e dell'AI Alan Touring.

L'obiettivo del progetto è realizzare, quindi di ingegneria.

Ma realizzare cosa? Macchine intelligenti inteso in senso ampio.

Nonostante all'inizio l'intelligenza artificiale si occupasse di realizzare macchine intelligenti sia dal punto di vista dal loro comportamento della loro interazione con il mondo che dal punto di vista degli strumenti per interagire con il mondo (hw per realizzarlo), poi le cose si sono specializzate.

Ad oggi si tende un po di più a parlare del comportamento del sw —> macchine intelligenti inteso programmi da computer in grado di eseguire un qualche comportamento che per noi sarà ritenuto intelligente.

Chi si occupa invece di realizzare la parte hw (la parte di interazione con il mondo) non si dice più intelligenza artificiale ma di robotica.

Siccome l'AI è evoluta nel tempo (è un progetto che nasce negli anni 50) e ad oggi si occupa di:

- sw in grado di risolvere problemi complessi;
- Sw che espongono comportamenti razionali —> macchine che sanno cosa fare sulla base di obiettivi;
- Sw in grado di interagire con l'uomo in un linguaggio naturale;
- Conseguentemente macchine in grado di generare contenuti creativi (sintetizzare immagini sulla base di descrizioni testuali o costruire poesie, musica ecc...);
- Sw in grado di interagire con mondi complessi e dinamici (bot che cerca informazioni sul web, bot che suggerisce cosa fare riguardo al nostro profilo di utenza, ecc...).

Come tutti i progetti l'AI ha un punto di inizio: una lettera che John McCarthy ha mandato ai suoi colleghi a metà degli anni 50, in quel periodo abbastanza per caso si ritrova a lavorare sul primo computer interattivo.

Alla domanda "cos'è l'intelligenza artificiale", McCarthy risponde: è una scienza e un'ingegneria della

creazioni di macchine intelligenti. (...) È simile al capire l'intelligenza umana, ma AI non ha un confine ai metodi che sono biologicamente osservabili.

Per altri AI significa risolvere problemi complessi che umanamente possono essere risolti in maniera semplice, ma che dove non ci si può associare un algoritmo accanto.

Ad oggi i fenomeni che vengono considerati complessi sono ancora molto umani, per esempio camminare: umanamente è semplice, dal punto di vista algoritmico invece è difficile da descrivere.

L'AI la si può affrontare con due prospettive:

- FORTE: che dice che il progetto AI non finirà fintanto che non riusciremo a realizzare delle macchine intelligenti, qualsiasi cosa vuol dire intelligente;
- DEBOLE: AI che si concentra su problemi specifici, che cerca di risolvere questi problemi nel modo migliore possibile, scegliendo i problemi da risolvere sempre nell'ottica di realizzare un qualcosa che possiamo considerare intelligente, ma senza porsi l'obiettivo di arrivare a ricostruire le persone sintetiche. Quindi passi piccolo, che sempre più e involontariamente portano ad avvicinarsi all'AI forte.

Ad oggi si considera che il prossimo passo possa essere la Artificial general Intelligence, un tipo di intelligenza che non è più debole perché non è più legata a compiti specifici ma è generica, come quella che hanno le persone.

L'intelligenza artificiale ha avuto dei fenomeni ad ondate: sempre nel tentativo di raggiungere l'obiettivo, un po' grossolanamente sono state percorse 4 strade (Thinking humanly, thinking rationally, acting humanly, acting rationally).

Oggi si parla di più di reti neurali.

Il primo modo di fare AI è stato acting humanly: l'intelligenza artificiale la si può fare comportandosi come si comportano le persone. Questo modo viene da Touring: inventando un test (The Imitation game) si pone il problema serio di capire che cosa voglia dire realizzare una macchina intelligente e per farlo, parte dal presupposto che una macchina la si definire intelligente se si comporta come una persona.

Test di Touring:

Prendiamo due persone C e B che giocano a scacchi, ma c'è un muro che separa i due giocatori. I

due iniziano a giocare a scacchi, nessuno dei due è un campionissimo ma sono entrambi in grado di esibire un'intelligenza sensata. Dopo un po' senza nessun avvertimento, il giocatore B viene sostituito da un programma per computer. L'obiettivo del test è misurare quanto sia intelligente nel comportamento del gioco degli scacchi il programma A, per farlo però ci serve una misura, un'unità, un numero: numero di mosse che A è in grado di giocare senza che C si renda conto dello scambio. Se il numero di mosse è basso, non va bene; al contrario con il numero di mosse elevato allora A si comporta bene nel task.

Siccome contare vuol dire generare numeri, siamo in grado di mettere in scala l'intelligenza di A: se le mosse sono poche, l'intelligenza di A sarà bassa, al contrario se le mosse sono tante, l'intelligenza è A è elevata.

Il problema però sta in C, che ha un'intelligenza a sè, nel cambiare il giocatore C, probabilmente cambierà la scala perchè cambia la capacità valutativa di C.

Queste macchine che *imitano* l'azione umana, erano dette macchine da imitation game perchè appunto il loro obiettivo non è capire non è fare, ma imitare.

Nota: le persone no agiscono così perché non sono in grado di fare meglio ma perchè hanno un pensiero che le spinge ad agire (il pensiero genera l'azione, non viceversa); quindi se vogliamo fare un passo avanti rispetto ad agire come persone, possiamo porci il problema di pensar come persone, fare quindi sistemi sw in grado di pensare come le persone = simulare l'organo fisico in cui il pensiero si forma -> il cervello.

I computer sono in grado di simulare qualsiasi sistema fisico, nell'ipotesi che esistano note delle leggi in grado di descrivere il sistema fisico, nell'ipotesi di avere abbastanza potenza di calcolo per poter fare una simulazione in un tempo ragionevole, però note queste due cose conosciamo abbastanza nel dettaglio il sistema che stiamo simulando e abbiamo abbastanza potenza di calcolo per simularne un comportamento interessante, il sistema fisico è simulabile e in maniera semplice. (Es. previsioni del tempo).

Il cervello non è nient'altro che una rete formata da neuroni, simuliamo il comportamento simulando una rete di struttura di calcolo che chiameremo neuroni. I neuroni ricevuto un input tramite segnale elettrico attraverso delle propagazioni tramite la rete, che prendono il nome di dendridi, quindi

collegheremo l'input non con dei segnali elettrici, ma con dei numeri che indicheranno l'intensità.

Ci viene detto che il neurone prende i segnali elettrici e li propaga verso l'esterno attraverso delle propagazioni che prendono il nome di assone e il tutto finisce in collegamenti che prendono il nome di sinapsi.

In astratto è una formula: entrano numeri, escono numeri, quello che succede nel mezzo dipende dalla somma dei numeri che entra, sapendo che qualche numero conta più di altri, ma in modo non lineare produce questi numeri in uscita.

Quindi quello che facciamo è prendere i numeri in ingresso, attribuire un peso a ciascuno dei numeri in ingresso, sommare tutto e questo ci darebbe una struttura totalmente lineare, ma non basta, passiamo per una funzione non lineare e andiamo a produrre un uscita: per ognuna di queste facciamo un calcolo, per ognuna di queste scegliamo i pesi.

La rete neurale non è nient'altro che un simulatore di cervello, che oggi riusciamo a far funzionare bene (cosa non possibile negli anni 70) perché abbiamo computer sufficientemente potenti e con un hw sufficientemente raffinato da analizzare bene queste cose.

Quello che andiamo a costruire è un rete neurale (in italiano un rete neurale è la simulazione di una rete neuronale = rete composta di neuroni), se noi prendiamo in generale un rete neurale, che cosa fa? Prende i numeri in input, produce numeri in output e di conseguenza ho qualcosa che lega gli input agli output: in matematica è una funzione da x variabili ad y variabili, in informatica è una funzione che calcola l'output sulla base dell'input.

Il calcolare l'output sulla base dell'input però vuol dire andare a scegliere i pesi da mettere sui collegamenti tra neuroni e neuroni, a questo punto ci vuole un programmatore che non scriva un programma ma che vada a definire i pesi e se ne abbiamo 10^{12} , diventa complicato; quindi anziché programmare la rete neurale, la si addestra: si forniscono degli input, si forniscono gli output corrett, si cercano dei pesi in grado di mappare l'input nell'output, poi si fornisce un altro input, un output corretto, si forniscono due pesi in grado di mappare i due output corretti, ecc... si continua così fino a quando tutti gli input che poniamo non vengono mappati in modo abbastanza preciso con tutti gli output corretti, se non ci riesce vuol dire che abbiamo bisogno di più neuroni, se la contrario si riesce subito vuol dire che ce ne sono troppi. Il problema è poi avere un algoritmo di apprendimento, cioè

un algoritmo in grado di calcolare i pesi, che sono numeri reali.

Rete convoluzionale: è un tipo di rete neurale artificiale spesso utilizzato per l'elaborazione delle immagini. Si basa sul concetto di convoluzione, che è un'operazione matematica che consente di estrarre caratteristiche da delle immagini.

Se si forza una rete ad avere una struttura convoluzionale/convolutiva, allora va fatta con più di 2 strati (input, output + intermedi) e allora si parla di rete profonda (deep network) perché ha più di due strati e un addestramento di una rete profonda prende il nome di deep learning.

Una volta che abbiamo capito che agire come una persona ci porta ai giochi di imitazione, pensare come una persona ci porta a simulare il cervello, ci rendiamo poi conto che le persone non sono interessanti. In relata dell'esperienza umana, quello che ci piace di più è la parte razionale: agire razionale e pensare razionale.

Il pensiero razionale vuol dire la logica: ne conosciamo due logiche tra le più semplici: logica delle proposizioni e la sua estensione temporale LTL, ovviamente non sono sufficienti per poter descrivere la logica razionale, ci servirà altro più raffinato (logica del primo ordine).

Il pensiero razionale è descritto bene dalla logica, quindi nessuno ci vieta di utilizzarla per descrivere il pensiero che vogliamo abbiamo i nostri sistemi sw.

Seguire la logica passa per un'idea che è quella di descrivere lo stato del nostro programma tramite un insieme di fatti che ci dicono cosa riteniamo vero e tramite un insieme di regole di deduzione (regole di inferenza), ci dicono cosa sarà vero una volta che noi abbiamo certi fatti veri

– Facts:

mother_of(ann, bob), sister_of(claire, ann)

– Inference rules:

mother_of(X, Y) ∧ sister_of(Z, X) → aunt_of(Z, Y)

Questo è un sistema esperto perchè conosce come un esperto, un insieme di fatti, un insieme di regole, ha una descrizione del mondo e in base a quella sa produrre nuove verità logiche incontrovertibili.

Pensare non basta, i sistemi di AI agiscono: nel momento in cui noi abbiamo un essere razionale

vogliamo immergerlo in un mondo e permettergli di muoversi nel tentativo di raggiungere i propri obiettivi. Un sistema sw immerso in un mondo in grado di ascoltare lo stato del mondo, in grado di compiere azioni sul mondo, che in modo razionale cerca di portare il mondo verso uno dei suoi stati di goal prende il nome di agente intelligente.

Un agente vuole il mondo (agente e ambiente) in certi stati, e agisce autonomamente per portarceli.

L'agente è caratterizzato dal suo comportamento ma non da cosa lo ha spinto a comportarsi in quel modo. Gli agenti sono spesso basati su linguaggi logici e, di conseguenza, usano la deduzione per decidere quale azione eseguire sull'ambiente e dato che sono razionali, non gli è richiesto imitare il comportamento umano.

Agenti intelligenti che sanno cosa fare in modo eventualmente stocastico e sono in grado di motivare le decisioni.

Se prima si considerava un singolo agente, ad oggi si considerano sempre più spesso sistemi multi-agente.

GPT (Generative Pre-Trained Transformers) sono delle reti neurali convoluzionali profonde ricorrenti che sono state addestrate per prevedere il prossimo testo a fronte di un testo parziale. (CHAT GPT, il completatore di parole sulla keyboard).

Gli LLM (modelli di linguaggio di grosse dimensioni) sono stati addestrati per fare questo compito: completare quello che l'utente ha lasciato incompleto, tranne che l'utente non fornisce qualche lettera o qualche parola, fornisce un testo e il completamento non è di un o qualche parola, ma è potenzialmente anche di un testo anche lungo un centinaio di parole; quindi in realtà ci stiamo solo illudendo che GPT stia rispondendo alla nostra domanda ma in realtà non è così, sta solo completando un dialogo ipotetico tra un personaggio che sta dicendo quello che state digitando voi e un altro personaggio che risponde.

GPT e LLM vengono addestrati con una quantità di materiale enorme che l'effetto sembra che GPT sappia scrivere.

Nei linguaggi naturali c'è anche la pragmatica (in aggiunta a semantica e sintassi), cioè come le cose vengono davvero utilizzate nelle conversazioni.

Tutti gli LLM (GPT incluso) sono esempi di agenti generativi di AI.

Appunti del Corso di Intelligenza Artificiale

Prerequisiti Matematici → **utile per l'algoritmo
di back propagation**

Prof. Federico Bergenti

26 febbraio 2024

Ricordiamo che le reti
neurali manipolano vettori
di numeri

1 Insiemi Numerici

In questi appunti vengono indicati gli insiemi con lettere maiuscole dell'alfabeto latino, mentre gli elementi degli insiemi vengono indicati con lettere minuscole, sempre dell'alfabeto latino. Unica eccezione è la notazione per gli insiemi numerici:

- \mathbb{N} è l'insieme dei **numeri naturali**, che include zero, mentre \mathbb{N}_+ è l'insieme dei numeri naturali positivi;
- \mathbb{Z} è l'insieme dei **numeri interi** e \mathbb{Z}_+ è l'insieme dei numeri interi positivi;
- \mathbb{Q} è l'insieme dei **numeri razionali** e \mathbb{Q}_+ è l'insieme dei numeri razionali positivi;
- \mathbb{R} è l'insieme dei **numeri reali** e \mathbb{R}_+ è l'insieme dei numeri reali positivi; e
- \mathbb{C} è l'insieme dei **numeri complessi**.

In più, come spesso accade quando si parla di funzioni polinomiali in più variabili, si assume la convenzione che $0^0 = 1$.

2 Vettori Reali

→ prodotto cartesiano
n volte su se stesso

Dato $n \in \mathbb{N}_+$, un elemento \mathbf{x} dell'insieme \mathbb{R}^n viene detto **vettore reale** (o di \mathbb{R}^n)

$$\mathbf{x} = (x_1, x_2, \dots, x_n) = (x_i)_{i=1}^n \quad \begin{array}{l} \text{forma alternativa} \\ = \text{tutti gli } x_i \text{ con } i \text{ che va da } 1 \text{ ad } n \end{array} \quad (1)$$

e i numeri reali x_i , con $1 \leq i \leq n$, sono le **componenti** (del vettore) \mathbf{x} . Si noti che n viene detto **dimensione** di \mathbb{R}^n e, quando si lavora anche con matrici, i vettori reali si comportano come **vettori colonna**. Infine, si noti che $\mathbf{0} \in \mathbb{R}^n$ e che i numeri reali vengono spesso chiamati **scalar**i.

I vettori reali possono essere moltiplicati per uno scalare e sommati, quindi \mathbb{R}^n è uno **spazio vettoriale** con le seguenti proprietà:

- scalare vettore
- Se $\alpha \in \mathbb{R}$ e $\mathbf{x} \in \mathbb{R}^n$, allora $\alpha\mathbf{x} \in \mathbb{R}^n$ e $\alpha\mathbf{x} = (\alpha x_1, \alpha x_2, \dots, \alpha x_n)$; e **PRODOTTO PER UNO SCALARE**
 - Se $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{R}^n$, allora $\mathbf{x} + \mathbf{y} \in \mathbb{R}^n$ e $\mathbf{x} + \mathbf{y} = (x_1 + y_1, x_2 + y_2, \dots, x_n + y_n)$. **SOMMA DI VETTORI**

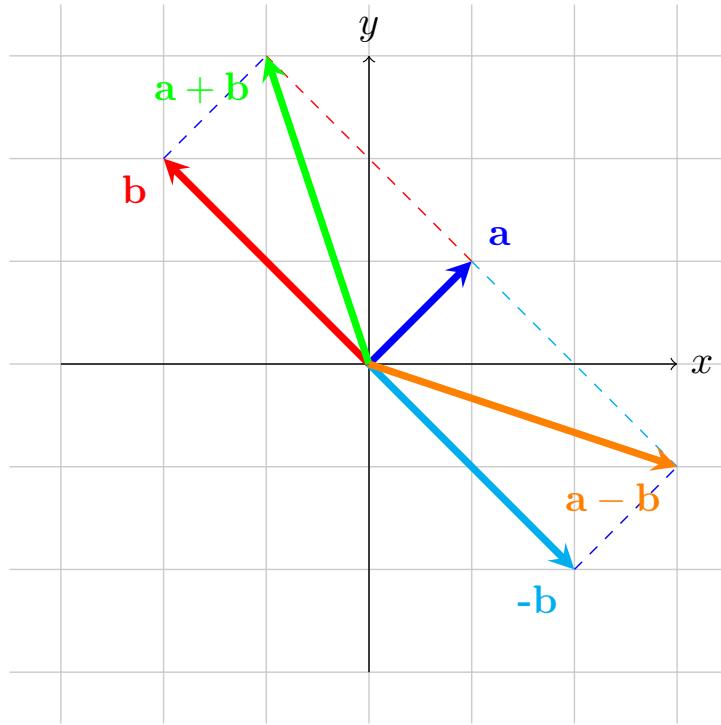


Figura 1: I vettori $\mathbf{a} = (1, 1)$ (blu) e $\mathbf{b} = (-2, 2)$ (rosso) e la loro somma $\mathbf{a} + \mathbf{b} = (-1, 3)$ (verde) e differenza $\mathbf{a} - \mathbf{b} = (3, -1)$ (arancione)

Naturalmente, il prodotto per uno scalare e la somma consentono di esprimere la sottrazione tra vettori reali $\mathbf{x} - \mathbf{y} = \mathbf{x} + (-1)\mathbf{y}$.

Sui vettori reali è possibile definire la norma euclidea (o lunghezza) e, dato un qualsiasi $\mathbf{x} \in \mathbb{R}^n$, vale la seguente definizione

$$\|\mathbf{x}\| = \sqrt{\sum_{i=1}^n x_i^2} \quad \begin{array}{l} \text{Pitagora.} \\ \text{componente i-esima elevata} \\ \text{al quadrato} \end{array} \quad (2)$$

dove per radice quadrata si intende, come di consueto, la radice quadrata reale positiva. Quindi, per ogni $\mathbf{x} \in \mathbb{R}^n$, valgono le due seguenti proprietà

$$\|\mathbf{x}\| \geq 0 \quad \text{e} \quad \|\mathbf{x}\| = 0 \iff \mathbf{x} = \mathbf{0} \quad \begin{array}{l} \text{vettore} \\ \text{nullo} \end{array} \quad (3)$$

Si noti che un vettore reale la cui norma euclidea vale 1 viene detto vettore unitario (o versore). In più, si noti che \mathbb{R}^n è uno spazio normato perché è stata definita opportunamente la norma euclidea.

Proposizione 1. *Data una qualsiasi coppia di vettori reali $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{R}^n$, vale la seguente diseguaglianza*

$$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\| \quad \begin{array}{l} \text{la norma della somma dei due vettori} \\ \text{è minore o uguale alla somma delle} \\ \text{norme dei vettori} \end{array} \quad (4)$$

che viene normalmente chiamata **diseguaglianza triangolare**.

Nel caso dei numeri reali, la norma euclidea si riduce al valore assoluto e, quindi, vale il noto caso particolare della diseguaglianza triangolare, con dove $x \in \mathbb{R}$ e $y \in \mathbb{R}$

$$|x + y| \leq |x| + |y| \quad (5)$$

✓ **Esempio 1.** Si consideri il caso bidimensionale, se

$$\mathbf{x} = (x_1, 0) \quad \text{e} \quad \mathbf{y} = (0, y_2) \quad (6)$$

allora

$$(\|\mathbf{x}\| + \|\mathbf{y}\|)^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 + 2\|\mathbf{x}\|\|\mathbf{y}\| \geq \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 = \|\mathbf{x} + \mathbf{y}\|^2 \quad (7)$$

che conferma la diseguaglianza triangolare sfruttando il teorema di Pitagora nell'ultima uguaglianza ed estraendo la radice quadrata.

La norma euclidea può essere utilizzata per definire la **distanza** tra due vettori reali $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{R}^n$, rendendo quindi \mathbb{R}^n uno **spazio metrico**,

$$d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\| = \|\mathbf{y} - \mathbf{x}\| \quad (8)$$

che, nel caso bidimensionale e tridimensionale, è la consueta distanza tra i punti identificati dai vettori una volta che sia stato scelto un sistema di riferimento.

Dati due vettori reali $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{R}^n$, si definisce **prodotto scalare**

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^n x_i y_i \quad \begin{array}{l} \text{sommatoria del prodotto} \\ \text{delle i-esime componenti} \\ \text{di vettori} \end{array} \quad \begin{array}{l} \text{il risultato è} \\ \text{uno scalare} \end{array} \quad (9)$$

Si noti che, per ogni $\mathbf{x} \in \mathbb{R}^n$, valgono le due seguenti proprietà

$$\|\mathbf{x}\|^2 = \mathbf{x} \cdot \mathbf{x} \quad \text{e} \quad \mathbf{0} \cdot \mathbf{x} = \mathbf{x} \cdot \mathbf{0} = 0 \quad (10)$$

Quindi, \mathbb{R}^n è uno **spazio con prodotto interno**.

Proposizione 2. Dati due vettori reali $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{R}^n$, vale la seguente **diseguaglianza di Cauchy-Schwarz**

$$|\mathbf{x} \cdot \mathbf{y}| \leq \|\mathbf{x}\| \|\mathbf{y}\| \quad (11)$$

e l'uguaglianza vale se e soltanto se esiste un $\lambda \in \mathbb{R}$ tale che $\mathbf{x} = \lambda \mathbf{y}$

$$|\mathbf{x} \cdot \mathbf{y}| = |\mathbf{x}| \cdot |\mathbf{y}|$$

se \mathbf{x} e \mathbf{y} sono paralleli
quindi multipli l'uno dell'altro

La diseguaglianza di Cauchy-Schwarz ha un'interpretazione geometrica interessante nel caso bidimensionale, che si estende facilmente al caso tridimensionale. In particolare, considerati $\mathbf{x} \in \mathbb{R}^2$ e $\mathbf{y} \in \mathbb{R}^2$

$$\mathbf{x} \cdot \mathbf{y} = x_1 y_1 + x_2 y_2 \quad (12)$$

Se entrambi i vettori vengono ruotati di un angolo $\beta \in \mathbb{R}$ in senso antiorario, allora

$$\mathbf{x}' = R_\beta \mathbf{x} \quad \text{e} \quad \mathbf{y}' = R_\beta \mathbf{y} \quad \begin{array}{l} \text{ruotare entrambi} \\ \text{vettori di} \\ \text{tor gradi} \end{array} \quad (13)$$

dove

$$R_\beta = \begin{pmatrix} \cos \beta & -\sin \beta \\ \sin \beta & \cos \beta \end{pmatrix} \quad \begin{array}{l} \text{il cui determinante} \\ (\mathbf{A} \cdot \mathbf{B}) - (\mathbf{C} \cdot \mathbf{D}) \end{array} \quad (14)$$

e quindi, riscrivendo opportunamente \mathbf{x}' e \mathbf{y}' e sfruttando il fatto che $\cos^2 \beta + \sin^2 \beta = 1$, si ottiene

$$\mathbf{x}' \cdot \mathbf{y}' = \mathbf{x} \cdot \mathbf{y} \quad \begin{array}{l} \text{questo conclude che} \\ \text{il prodotto scalare è} \\ \text{indipendente dalla rotazione} \end{array} \quad (15)$$

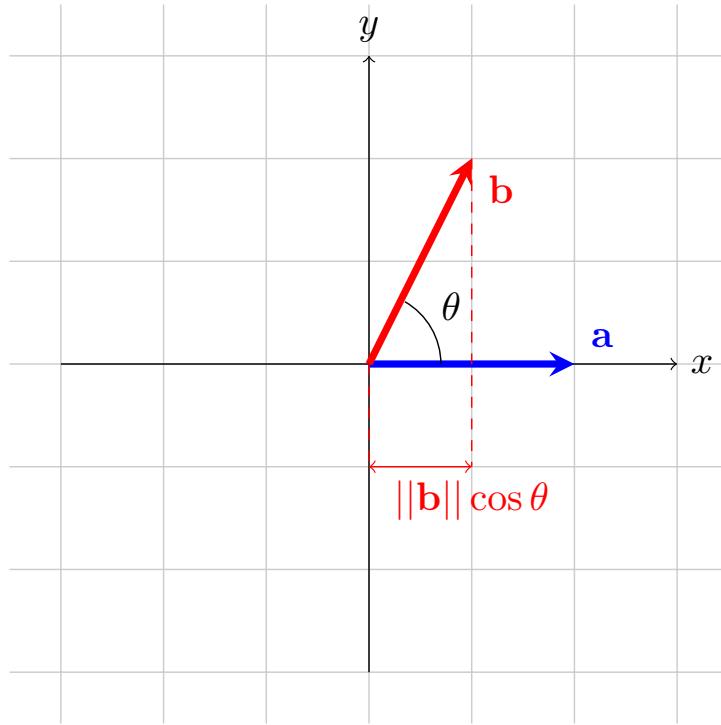


Figura 2: I vettori $\mathbf{a} = (2, 0)$ (blu) e $\mathbf{b} = (1, 2)$ (rosso) e la lunghezza $\|\mathbf{b}\| \cos \theta$ della proiezione di \mathbf{b} su \mathbf{a}

La considerazione precedente assicura che il prodotto scalare in \mathbb{R}^2 sia indipendente dalla rotazione dello stesso angolo di entrambi i vettori. In più, assicura che sia sempre possibile calcolare il prodotto scalare tra due vettori reali allineando uno dei due vettori con l'asse delle ascisse. Ad esempio, se il vettore \mathbf{x} viene allineato con l'asse delle ascisse, allora

$$\mathbf{x} = (\|\mathbf{x}\|, 0) \quad \text{e} \quad \mathbf{y} = (\|\mathbf{y}\| \cos \theta, \|\mathbf{y}\| \sin \theta) \quad (16)$$

dove $\theta \in \mathbb{R}$ è l'angolo tra i due vettori. In questa situazione, $0 < \cos \theta < 1$

$$\xrightarrow{\substack{\text{modulo del} \\ \text{prodotto} \\ \text{scalare}}} |\mathbf{x} \cdot \mathbf{y}| = \|\mathbf{x}\| \|\mathbf{y}\| |\cos \theta| \leq \|\mathbf{x}\| \|\mathbf{y}\| \quad (17)$$

che conferma che l'uguaglianza vale se e soltanto se i due vettori giacciono sulla stessa retta perché, in questo caso, vale $\cos \theta = 1$.

Si noti che le considerazioni precedenti riguardo al calcolo del prodotto scalare possono essere generalizzate. Dato $n \in \mathbb{N}_+$, una matrice $R = (r_{i,j})_{i,j=1}^n$ si dice **ortogonale** se e soltanto se è invertibile e vale

$$R^\top = R^{-1} \quad \substack{\downarrow \text{matrice quadrata } n \times n \\ \text{trasposta} = \text{inversa}} \quad (18)$$

Quindi, una matrice ortogonale R rappresenta un'applicazione lineare che preserva le lunghezze dei vettori reali, e quindi viene chiamata **isometria**^{*} perché per ogni $\mathbf{x} \in \mathbb{R}^n$ vale

$$\|\mathbf{Rx}\|^2 = (\mathbf{Rx}) \cdot (\mathbf{Rx}) = \mathbf{x}^\top [R^\top R] \mathbf{x} = \mathbf{x}^\top \mathbf{Id} \mathbf{x} = \|\mathbf{x}\|^2 \quad (19)$$

Si noti che il determinante di una matrice di ortogonale può valere solo ± 1 , infatti se R è una matrice di ortogonale

$$1 = \det RR^{-1} = \det RR^\top = (\det R)^2 \quad (20)$$

una funzione matematica
che mappa uno spazio
vettoriale, preservando
le operazioni di addizione
vettoriale e moltiplicazione
per uno scalare

applicazioni lineari
isometriche =
partendo da un vettore
con una certa lunghezza,
preservano la lunghezza
del vettore

Una matrice ortogonale R per cui vale $\det R = 1$ si dice **matrice di rotazione** e vale, per ogni $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{R}^n$,

$$R\mathbf{x} \cdot R\mathbf{y} = \mathbf{x}^\top R^\top R\mathbf{y} = \mathbf{x} \cdot \mathbf{y} \quad (21)$$

↓ preserva sia le lunghezze
che il prodotto scalare

Dato $n \in \mathbb{N}_+$, due vettori reali $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{R}^n$, entrambi diversi da $\mathbf{0}$, si dicono **ortogonali** se e soltanto se

$$\text{prodotto scalare } \mathbf{x} \cdot \mathbf{y} = 0 \quad (22)$$

coerentemente con il fatto che, nei casi bidimensionali e tridimensionali, l'angolo tra i due vettori annulla il coseno utilizzato per il calcolo di $\mathbf{x} \cdot \mathbf{y}$.

Un insieme finito di vettori reali non nulli $O \subset \mathbb{R}^n$ si dice formato da vettori **mutuamente ortogonali** se e soltanto se, qualsiasi siano $\mathbf{x} \in O$ e $\mathbf{y} \in O$, vale $\mathbf{x} \cdot \mathbf{y} = 0$. Si noti che gli insiemi di vettori mutuamente ortogonali sono particolarmente interessanti perché vale la seguente proposizione.

Proposizione 3. *Dato $n \in \mathbb{N}_+$, se $O \subset \mathbb{R}^n$ è un insieme di vettori reali mutuamente ortogonali, allora è anche un insieme di vettori linearmente indipendenti.*

* se nessuno di essi può essere espresso come combinazione lineare degli altri.

Un insieme di vettori è lin. ind. se

$c_1v_1 + c_2v_2 + \dots + c_nv_n = 0$
ha soluzione solo se tutti i coefficienti $c_x = 0$.

Spesso si studiano dei sottoinsiemi di \mathbb{R}^n che hanno un interesse dal punto di vista dell'intuizione geometrica. Questi sottoinsiemi generalizzano gli insiemi con gli stessi nomi comunemente descritti nei casi bidimensionali e tridimensionali.

Dati due vettori reali $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{R}^n$, la **retta** (\mathbf{x}, \mathbf{y}) passante per i punti identificati da \mathbf{x} e \mathbf{y} è

$$(\mathbf{x}, \mathbf{y}) = \{\mathbf{v} \in \mathbb{R}^n : \mathbf{v} = \mathbf{x} + \lambda(\mathbf{y} - \mathbf{x}) \wedge \lambda \in \mathbb{R}\} \quad (23)$$

$y = mx + q \text{ in } \mathbb{R}^2$

Dati due vettori reali $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{R}^n$, il **segmento** $[\mathbf{x}, \mathbf{y}]$ che congiunge i punti identificati da \mathbf{x} e \mathbf{y} è

$$[\mathbf{x}, \mathbf{y}] = \{\mathbf{v} \in \mathbb{R}^n : \mathbf{v} = \mathbf{x} + \lambda(\mathbf{y} - \mathbf{x}) \wedge \lambda \in [0, 1]\} \quad (24)$$

Dato il vettore reale $\mathbf{x} \in \mathbb{R}^n$ e il versore $\mathbf{n} \in \mathbb{R}^n$, l'**iperpiano** che contiene \mathbf{x} e ha \mathbf{n} come (versore) **normale** è

$$\mathcal{H}_{\mathbf{n}}(\mathbf{x}) = \{\mathbf{v} \in \mathbb{R}^n : (\mathbf{v} - \mathbf{x}) \cdot \mathbf{n} = 0\} \quad (25)$$

+ assicura che i vettori partano dalla fine di x

Dati due vettori reali $\mathbf{x} \in \mathbb{R}^n$ e $\mathbf{y} \in \mathbb{R}^n$, la **scatola (aperta)** (\mathbf{x}, \mathbf{y}) identificata dagli estremi \mathbf{x} e \mathbf{y} è

$$(\mathbf{x}, \mathbf{y}) = \{\mathbf{v} \in \mathbb{R}^n : \underline{v}_1 < v_1 < \bar{v}_1 \wedge \underline{v}_2 < v_2 < \bar{v}_2 \wedge \dots \wedge \underline{v}_n < v_n < \bar{v}_n\} \quad (26)$$

dove, per ogni $1 \leq i \leq n$,

$$\underline{v}_i = \min\{x_i, y_i\} \quad \text{e} \quad \bar{v}_i = \max\{x_i, y_i\} \quad (27)$$

mentre la **scatola chiusa** $[\mathbf{x}, \mathbf{y}]$ è

$$[\mathbf{x}, \mathbf{y}] = \{\mathbf{v} \in \mathbb{R}^n : \underline{v}_1 \leq v_1 \leq \bar{v}_1 \wedge \underline{v}_2 \leq v_2 \leq \bar{v}_2 \wedge \dots \wedge \underline{v}_n \leq v_n \leq \bar{v}_n\} \quad (28)$$

Si noti che una scatola aperta può degenerare nell'insieme vuoto se viene usato lo stesso vettore per identificare i due estremi della scatola. Al contrario, in questa situazione, una scatola chiusa degenera in un insieme singoletto.

Dato un vettore reale $\mathbf{x} \in \mathbb{R}^n$ e uno scalare $r \in \mathbb{R}_+$, la **palla (aperta)** $\mathcal{B}_r(\mathbf{x})$, centrata nel punto identificato da \mathbf{x} e di raggio r è

$$\mathcal{B}_r(\mathbf{x}) = \{\mathbf{v} \in \mathbb{R}^n : \|\mathbf{v} - \mathbf{x}\| < r\} \quad (29)$$

mentre la **palla chiusa** $\mathcal{B}_r[\mathbf{x}]$ è

$$\mathcal{B}_r[\mathbf{x}] = \{\mathbf{v} \in \mathbb{R}^n : \|\mathbf{v} - \mathbf{x}\| \leq r\} \quad (30)$$

↓ in \mathbb{R}^3 , graficamente, è una sfera privata dello strato esterno

Si noti che una palla non può degenerare nell'insieme vuoto perché $r > 0$.

Le definizioni precedenti consentono di generalizzare alcune nozioni comunemente usate nei casi monodimensionali, bidimensionali e tridimensionali:

- $L \subseteq \mathbb{R}^n$ si dice **limitato** se esistono $r \in \mathbb{R}_+$ e $\mathbf{x} \in \mathbb{R}^n$ tali che $L \subseteq \mathcal{B}_r(\mathbf{x})$;
- $A \subseteq \mathbb{R}^n$ si dice **aperto** se, per ogni $\mathbf{x} \in A$, esiste $r \in \mathbb{R}_+$ tale che $\mathcal{B}_r(\mathbf{x}) \subseteq A$;
- $C \subseteq \mathbb{R}^n$ si dice **chiuso** se esiste un $A \subseteq \mathbb{R}^n$ aperto tale che $C = \mathbb{R}^n \setminus A$;
- $K \subseteq \mathbb{R}^n$ si dice **compatto** se è chiuso e limitato; e
- $V \subseteq \mathbb{R}^n$ si dice **convesso** se per ogni coppia $\mathbf{x} \in V$ e $\mathbf{y} \in V$ vale $[\mathbf{x}, \mathbf{y}] \subseteq V$.

Esempio 2. Si consideri il sottoinsieme S di \mathbb{R}^2 definito come segue:

$$S = \{(x, y) \in \mathbb{R}^2 : 2x^2 + 2y^2 + 2xy - 1 \leq 0\} \quad (31)$$

Visto che la forma quadrica non è in forma canonica (parabola, ellisse, iperbole, coppia di rette), si cerchi una matrice di rotazione M tale che

$$2x^2 + 2y^2 + 2xy = \mathbf{x}^\top Q \mathbf{x} \quad \mathbf{x} = (x, y) \quad Q = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \quad (32)$$

risulti diagonale quando $\mathbf{x} = M\mathbf{x}'$. Gli autovalori di Q si ottengono risolvendo

$$\det(Q - \lambda I_2) = \det \begin{pmatrix} 2 - \lambda & 1 \\ 1 & 2 - \lambda \end{pmatrix} = 0 \quad (33)$$

e valgono $\lambda_1 = 1$ e $\lambda_2 = 3$. Due autovettori corrispondenti ai due autovalori trovati sono

$$\mathbf{v}_1 = (1, -1) \quad \mathbf{v}_2 = (1, 1) \quad (34)$$

da cui si può ottenere la matrice diagonale D cercata

$$D = M^\top Q M \quad M = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \quad D = \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix} \quad (35)$$

Quindi, S è l'insieme dei vettori reali racchiusi da una ellisse, inclinata di $-\frac{\pi}{4}$ radianti, con semiassi di lunghezza 1 e $\frac{1}{\sqrt{3}}$.

3 Funzioni Reali di Più Variabili Reali

Dati $n \in \mathbb{N}_+$ e $m \in \mathbb{N}_+$, si considerino un insieme $D \subseteq \mathbb{R}^n$, detto **dominio**, e un insieme $C \subseteq \mathbb{R}^m$, detto **codominio**. La funzione

$$f : D \rightarrow C \quad (36)$$

funzione che produce vettori

è una **funzione vettoriale** a m componenti reali di n variabili reali. Le funzioni di questo tipo possono essere studiate come m funzioni reali di n variabili reali

$$f(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_m(\mathbf{x})) \quad (37)$$

una funzione vettoriale prende in input uno scalare da un output un vettore

e quindi, normalmente, lo studio delle funzioni vettoriali viene ridotto allo studio delle funzioni reali di più variabili reali.

Si consideri una funzione reale f di più variabili reali definita almeno in un aperto $D \subseteq \mathbb{R}^n$. Si dice che

$$\lim_{\mathbf{x} \rightarrow \mathbf{w}} f(\mathbf{x}) = l \quad (38)$$

vettori di \mathbb{R}^n

se e soltanto se

$$\forall \epsilon \in \mathbb{R}_+, \exists \delta \in \mathbb{R}_+ : \forall \mathbf{x} \in D, \|\mathbf{x} - \mathbf{w}\| < \delta \implies |f(\mathbf{x}) - l| < \epsilon \quad (39)$$

Ad ogni ϵ reale positivo, esiste δ reale positivo tali che per ogni \mathbf{x} del dominio, se la distanza tra \mathbf{x} e \mathbf{w} è minore di δ , allora il modulo della differenza tra la funzione in \mathbf{x} e il limite può essere minore di ϵ . La definizione precedente permette di chiarire quando una funzione reale possa essere definita continua in $\mathbf{w} \in D$

$$\lim_{\mathbf{x} \rightarrow \mathbf{w}} f(\mathbf{x}) = f(\mathbf{w}) \quad (40)$$

la funzione è continua se esiste il limite e vale $f(\mathbf{w})$

Proposizione 4. Sia f una funzione reale di $n \in \mathbb{N}_+$ variabili reali definita almeno in un aperto $S \subseteq \mathbb{R}^n$, se la funzione f è continua in tutto S , allora per ogni compatto $D \subset S$ la funzione ammette almeno un minimo globale in D e almeno un massimo globale in D .

Si consideri una funzione reale f definita almeno in un aperto $D \subseteq \mathbb{R}^n$ e si considerino $\mathbf{x} \in D$ e un versore $\mathbf{v} \in \mathbb{R}^n$. Se esiste finito

$$\partial_{\mathbf{v}} f(\mathbf{x}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h} \quad (41)$$

allora viene chiamato **derivata direzionale** della funzione f nella direzione \mathbf{v} . La derivata direzionale quantifica la velocità di crescita della funzione nella direzione considerata. Ovviamente, nel caso di funzioni reali di una sola variabile reale, la derivata direzionale si riduce alla **derivata (ordinaria)**.

Si consideri una funzione reale f definita almeno in un aperto $D \subseteq \mathbb{R}^n$ e si considerino $\mathbf{x} \in D$ e un versore del tipo $\mathbf{v} = (0, 0, \dots, 1, 0, 0, \dots, 0)$ con il valore 1 in posizione $1 \leq i \leq n$. Se esiste, la derivata nella direzione \mathbf{v} viene chiamata **derivata parziale** in x_i e viene indicata

$$\frac{\partial f}{\partial x_i}(\mathbf{x}) \quad \text{oppure} \quad \partial_i f(\mathbf{x}) \quad (42)$$

non è δ , ma un simbolo apposito

Si noti che il calcolo delle derivate parziali può essere ridotto al calcolo delle derivate ordinarie. Infatti, siccome il limite del rapporto incrementale coinvolge una sola variabile, durante il calcolo di una derivata parziale è sempre possibile trattare le altre variabili come se fossero delle costanti.

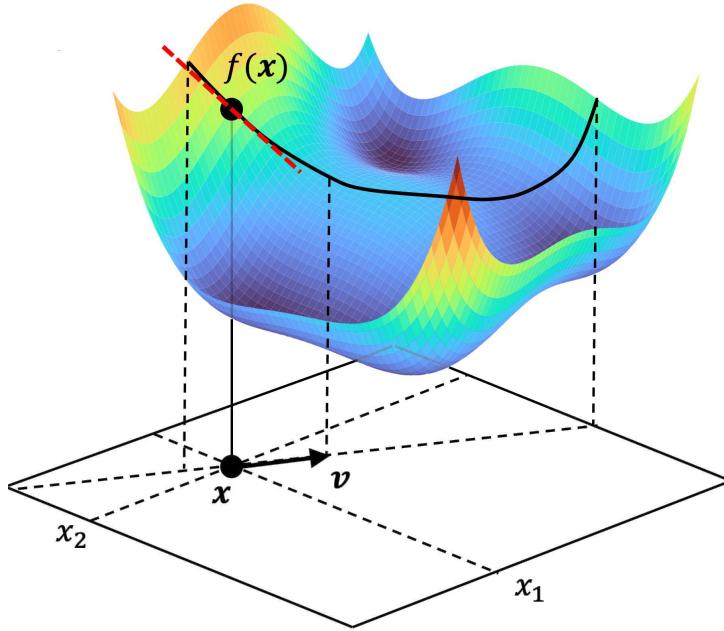


Figura 3: Derivata di $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ in $\mathbf{x} \in \mathbb{R}^2$ nella direzione $\mathbf{v} \in \mathbb{R}^2$

Si noti che, se esiste, il vettore formato dalle $n \in \mathbb{N}_+$ derivate parziali di una funzione reale di n variabili reali viene detto **gradiente** della funzione e si indica con

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}(\mathbf{x}), \frac{\partial f}{\partial x_2}(\mathbf{x}), \dots, \frac{\partial f}{\partial x_n}(\mathbf{x}) \right) \quad (43)$$

Esempio 3. Si consideri

$$f(x, y, z) = 3x^2 + 5xy - 7z^3 \quad (44)$$

allora

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}(x, y, z), \frac{\partial f}{\partial y}(x, y, z), \frac{\partial f}{\partial z}(x, y, z) \right) = (6x + 5y, 5x, -21z^2) \quad (45)$$

Infatti, trattando y e z come costanti, è facile calcolare che

derivo la formula 44

$$\frac{\partial f}{\partial x} = 6x + 5y \quad (46)$$

Allo stesso modo, trattando x e z come costanti, si ottiene

↓ f. 44

$$\frac{\partial f}{\partial y} = 5x \quad (47)$$

Infine, trattando x e y come costanti, si ottiene

↓ f. 44

$$\frac{\partial f}{\partial z} = -21z^2 \quad (48)$$

Se le derivate parziali di una funzione possono essere ulteriormente derivate, allora si parla di **derivate (parziali) di ordine superiore (al primo)** quando le derivate vengono calcolate rispetto ad una sola variabile, oppure di **derivate (parziali) miste** quando le

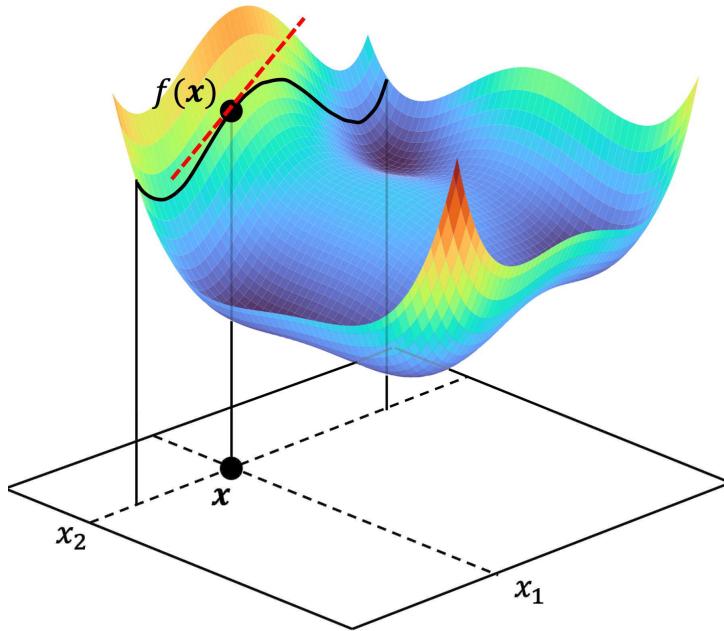


Figura 4: Derivata parziale in x_2 di $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ in $\mathbf{x} \in \mathbb{R}^2$

derivate vengono calcolate rispetto a più variabili. In generale, l'**ordine** di una derivata parziale è pari al numero di derivate calcolate, indipendentemente dalle variabili utilizzate. Per le derivate miste e le derivate di ordine superiore si utilizza una notazione che estende quella delle derivate parziali. Ad esempio, le due seguenti espressioni

$$\frac{\partial^3 f}{\partial x^3} \quad \text{e} \quad \frac{\partial^2 f}{\partial z \partial y} \quad (49)$$

indicano, rispettivamente, la derivata della funzione f nella variabile y compiuta tre volte e la derivata della funzione f nella variabile y e poi nella variabile z .

Sia $D \subseteq \mathbb{R}^n$ un aperto e sia $k \in \mathbb{N}_+$, una funzione $f : S \rightarrow \mathbb{R}$, con $D \subseteq S$, si dice di **classe $C^k(D)$** se è derivabile almeno k volte su D e se le sue derivate sono continue su D . Una funzione si dice di classe $C^0(D)$ se è continua su D e si dice di classe $C^\infty(D)$ se è derivabile un numero arbitrario di volte su D e le sue derivate sono tutte continue su D .

Proposizione 5. *Se una funzione reale di $n \in \mathbb{N}_+$ variabili reali è di classe $C^k(D)$, con $k \in \mathbb{N}$ e $D \subseteq \mathbb{R}^n$ aperto, allora le sue derivate fino all'ordine k incluso sono indipendenti dall'ordine in cui vengono calcolate. Se una funzione è di classe $C^\infty(D)$, con $D \subseteq \mathbb{R}^n$ aperto, allora le sue derivate, di qualsiasi ordine, sono indipendenti dall'ordine in cui vengono calcolate.*

La seguente proposizione mette in relazione la possibilità di calcolare le derivate parziali con la proprietà di continuità delle funzioni.

Proposizione 6. *Se una funzione reale di $n \in \mathbb{N}_+$ variabili reali è di classe $C^k(D)$, con $k \in \mathbb{N}_+$ e $D \subseteq \mathbb{R}^n$ aperto, allora la funzione è di classe $C^{k-1}(D)$. Se una funzione è di classe $C^\infty(D)$, con $D \subseteq \mathbb{R}^n$ aperto, allora per ogni $k \in \mathbb{N}$ è di classe $C^k(D)$.*

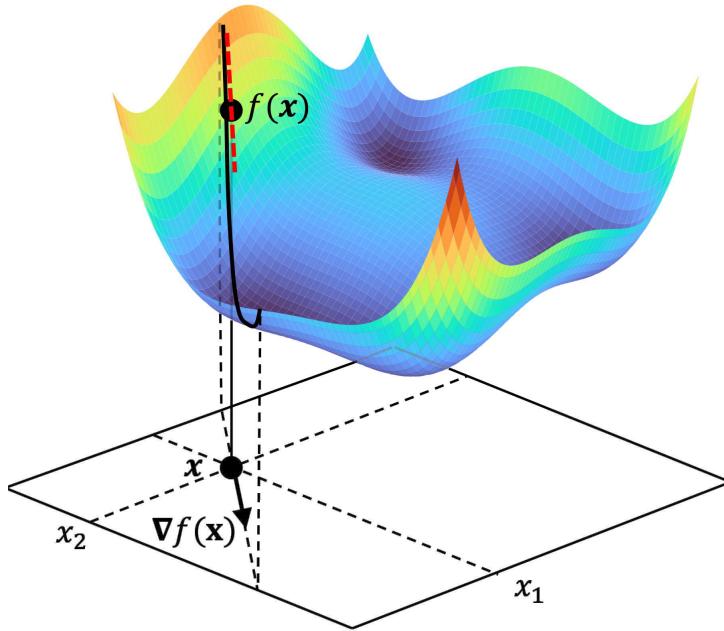


Figura 5: Gradiente di $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ in $\mathbf{x} \in \mathbb{R}^2$

Il seguente esempio mostra come sia possibile sfruttare la classe di una funzione nel calcolo delle sue derivate. In particolare, mostra come sia possibile sfruttare il fatto che una funzione polinomiale di $n \in \mathbb{N}_+$ variabili è sempre di classe $C^\infty(\mathbb{R}^n)$.

✓ **Esempio 4.** Si consideri

$$f(x, y, z) = 3x^2 + 5xy - 7z^3 \quad (50)$$

allora

vuol dire derivata due volte: prima in x e poi in y

$$\frac{\partial^2 f}{\partial x \partial y}(x, y, z) = \frac{\partial^2 f}{\partial y \partial x}(x, y, z) = 5 \quad (51)$$

che conferma come l'ordine in cui vengono calcolate le derivate non sia rilevante visto che, in questo caso, tutte le derivate di qualsiasi ordine esistono e sono continue.

La seguente proposizione mostra la rilevanza del gradiente nel calcolo delle derivate direzionali. In particolare, mostra come la conoscenza del gradiente sia sufficiente per il calcolo della derivata in una direzione qualsiasi.

Proposizione 7. Sia f una funzione reale di $n \in \mathbb{N}_+$ variabili reali definita almeno in un aperto $D \subseteq \mathbb{R}^n$, se f è almeno di classe $C^1(D)$, allora per ogni $\mathbf{x} \in D$ vale

$$\partial_{\mathbf{v}} f(\mathbf{x}) = \mathbf{v} \cdot \nabla f(\mathbf{x}) \quad (52)$$

qualsiasi sia il versore $\mathbf{v} \in \mathbb{R}^n$. derivata direzionale di f nella direzione v prodotto scalare tra versore e gradiente della funzione

Si noti che la diseguaglianza di Cauchy-Schwarz applicata ad una funzione f per cui valgono le ipotesi della proposizione precedente permette di dire che

$$-\|\nabla f(\mathbf{x})\| \leq \partial_{\mathbf{v}} f(\mathbf{x}) \leq \|\nabla f(\mathbf{x})\| \quad (53)$$

l'opposto della norma del gradiente

10 derivata direzionale della funzione nella direzione di v

norma del gradiente

qualsiasi v che sceglieremo avrà derivata direzionale minore della norma del gradiente e maggiore dell'opposto questo fissa un upper e un lower bound della norma del gradiente

qualsiasi sia il versore $\mathbf{v} \in \mathbb{R}^n$ e, in più, permette di dire che le uguaglianze valgono se e soltanto se esiste un $\lambda \in \mathbb{R}$ tale che

$$\mathbf{v} = \lambda \nabla f(\mathbf{x}) \rightarrow \begin{array}{l} \text{il versore è multiplo} \\ \text{del gradiente} \end{array} \rightarrow \begin{array}{l} \text{e ci dice quale è la direzione} \\ \text{in cui muoversi per} \end{array} \quad (54)$$

massimizzare la velocità di crescita della funzione (direzione gradiente) o massimizzare la velocità di decrescita della f. ne (direzione gradiente cambiato di segno)

Quindi, si evince che, fissato un vettore $\mathbf{x} \in \mathbb{R}^n$ nel dominio della funzione, $\nabla f(\mathbf{x})$ indica la direzione di massima velocità di crescita della funzione nel punto identificato da \mathbf{x} e $-\nabla f(\mathbf{x})$ indica la direzione di massima velocità di decrescita della funzione nel punto identificato da \mathbf{x} . Se vale $\nabla f(\mathbf{x}) = \mathbf{0}$, allora la funzione si dice **stazionaria** nel punto identificato da \mathbf{x} e il punto identificato da \mathbf{x} si dice (**punto**) **critico**.

Proposizione 8. Sia f una funzione reale di $n \in \mathbb{N}_+$ variabili reali definita almeno in un aperto $D \subseteq \mathbb{R}^n$, se il gradiente di f è definito su tutto D , allora f può ammettere massimi locali e minimi locali in D solo nei punti in cui è stazionaria.

L'esempio seguente mostra come sia possibile studiare massimi e minimi di una funzione in un aperto studiando il gradiente della funzione.

Esempio 5. Si consideri la funzione $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ definita da

$$f(x, y) = -2x^2 - 3y^2 + 4xy + 10 \quad (55)$$

Il gradiente della funzione vale

$$\nabla f(x, y) = (4y - 4x, 4x - 6y)$$

e quindi la funzione ammette un unico punto critico $(0, 0)$ in cui vale $f(0, 0) = 10$. Siccome è possibile scrivere la funzione come

$$f(x, y) = -2(x - y)^2 - y^2 + 10$$

si evince che l'unico punto critico è un **massimo globale**. **verificare**

- Spiegazione in generale:**
1. Calcolare derivate parziali
 2. Impostazione del sist. di eq. ni e risoluzione
 3. Analisi del punto critico ottenuto
 4. Valutato la matrice hessiana in f : $H(f) = \begin{bmatrix} -4 & 4 \\ 4 & -6 \end{bmatrix}$
- $\det(H(f)) = 24 - 16 = 8$
Il det è positivo, valuto l'elemento in 1,1, è negativo quindi il punto critico è un massimo.
(se fosse stato negativo, il punto critico è un minimo.
se fosse stato uguale a zero, sarebbero serviti ulteriori test.)

L'esempio seguente mostra come sia possibile studiare massimi e minimi di una funzione definita in un insieme ben più complesso di quello utilizzato nell'esempio precedente. In particolare, l'esempio mostra come massimi e minimi globali di una funzione debbano essere cercati anche sul **bordo** (o **frontiera**) del dominio quando il dominio è chiuso.

Esempio 6. Si consideri la funzione $f : D \rightarrow \mathbb{R}$ definita da

$$f(x, y) = 2x^2 + 5y^2 + 4 \quad (58)$$

sul dominio $D = \mathcal{B}_2[1, 0]$. Il gradiente della funzione vale

palla chiusa di
 $r=2$

$$\nabla f(x, y) = (4x, 10y) \quad (59)$$

e quindi la funzione ammette un unico punto critico $(0, 0)$ in cui vale $f(0, 0) = 4$. Questo punto critico non può essere un massimo globale sul dominio D perché, ad esempio,

$f(1, 0) = 6$. Quindi, i massimi globali possono solo trovarsi sul bordo del dominio, che è l'insieme

$$\partial D = \{(x, y) : (x - 1)^2 + y^2 = 4\} \quad (60)$$

Sfruttando il fatto che in ∂D vale $y^2 = 4 - (x - 1)^2$ è possibile cercare i massimi globali di f cercando i massimi globali di $g : \mathbb{R} \rightarrow \mathbb{R}$

$$g(x) = 2x^2 + 5[4 - (x - 1)^2] + 4 = -3x^2 + 10x + 19 \quad (61)$$

Siccome g è una parabola con concavità verso il basso, è semplice calcolarne il valore massimo. Infatti, siccome g ammette massimo globale in $x = \frac{5}{3}$, il massimo globale di f si trova nei due punti $(\frac{5}{3}, \pm \frac{4\sqrt{2}}{3})$.

Lo studio di massimi e minimi locali in un aperto può essere raffinato per funzioni di una o due variabili reali, come mostrato dalle seguenti proposizioni.

Proposizione 9. Sia $D \subseteq \mathbb{R}$ un aperto e sia $f : D \rightarrow \mathbb{R}$ di classe $C^2(D)$. Se $x \in D$ è un punto critico di f , allora:

- Se $f''(x) > 0$, allora x è un minimo locale;
- Se $f''(x) < 0$, allora x è un massimo locale; e
- Se $f''(x) = 0$, allora il test è inconclusivo.

Una proposizione simile può essere espressa per funzioni reali di due variabili reali considerando la seguente definizione di **matrice hessiana**

$$H_f(x, y) = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2}(x, y) & \frac{\partial^2 f}{\partial x \partial y}(x, y) \\ \frac{\partial^2 f}{\partial y \partial x}(x, y) & \frac{\partial^2 f}{\partial y^2}(x, y) \end{pmatrix} \quad (62)$$

per cui viene definito il **(valore) hessiano** $h_f(x, y) = \det H_f(x, y)$. Si noti che se una funzione è definita su un aperto D ed è almeno di classe $C^2(D)$, allora la sua matrice hessiana è simmetrica perché che le due derivate miste sono uguali tra loro. In più, si noti che il calcolo del valore hessiano è sempre semplice perché vengono considerate solo matrici con due righe e due colonne.

Proposizione 10. Sia $D \subseteq \mathbb{R}^2$ un aperto e sia $f : D \rightarrow \mathbb{R}$ di classe $C^2(D)$. Se $x \in D$ è un punto critico di f , allora:

- Se $h_f(x) > 0$, ~~allora x è un minimo locale~~; devo valutare l'elemento in posizione (1,1)
- Se $h_f(x) < 0$, allora x è un massimo locale; e
- Se $h_f(x) = 0$, allora il test è inconclusivo.

se è positivo:
minimo locale
se è negativo:
massimo locale

Il seguente esempio mostra come sia possibile utilizzare la proposizione precedente per studiare massimi e minimi locali di una funzione reale di due variabili reali. In più, l'esempio mostra come sia possibile utilizzare la proposizione precedente per dimostrare che massimo o minimo non esistono sul dominio considerato.

✓ **Esempio 7.** Si consideri la funzione $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ definita da

(vedere anche
spiegazione es.5)

$$f(x, y) = x^2 - 8x + y^2 - 3y + 12 \quad (63)$$

Il gradiente della funzione vale

$$\nabla f(x, y) = (2x - 8, 2y - 3) \quad (64)$$

e quindi la funzione ammette un unico punto critico $(4, \frac{3}{2})$. La matrice hessiana delle funzione vale

$$H_f(x, y) = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \quad (65)$$

e, quindi, l'hessiano vale $h_f = 4$, indipendentemente dal valore di x e di y . Quindi, il punto critico $(4, \frac{3}{2})$ è un minimo locale della funzione. In più, si può constatare come la funzione f non ammetta massimi locali su \mathbb{R}^2 perché non esistono punti critici delle funzione in cui valga $h_f < 0$ oppure $h_f = 0$.

Le precedenti considerazioni sulle relazioni tra minimi, massimi e punti critici permettono di descrivere un algoritmo per la ricerca di un minimo locale di una funzione reale di $n \in \mathbb{N}_+$ variabili reali almeno di classe $C^1(\mathbb{R})$. L'algoritmo, detto di *discesa del gradiente*, è descritto nell'Algoritmo 1 e richiede quattro argomenti: la funzione f di cui si cerca un minimo locale, una prima approssimazione $\mathbf{x} \in \mathbb{R}^n$ di uno dei minimi locali di f , il **passo di discesa** $\alpha \in \mathbb{R}_+$ e un valore $\delta \in \mathbb{R}_+$ da utilizzare per decidere quando la lunghezza del gradiente possa essere considerata abbastanza piccola.

Algoritmo 1 Algoritmo di discesa del gradiente

```
function GRADIENT_DESCENT( $f, \mathbf{x}, \alpha, \delta$ )
     $\mathbf{g} \leftarrow \nabla f(\mathbf{x})$                                  $\triangleright$  inizializza  $\mathbf{g}$  a  $\nabla f(\mathbf{x})$ 
    while  $\|\mathbf{g}\| > \delta$  do
         $\mathbf{x} \leftarrow \mathbf{x} - \alpha \mathbf{g}$                        $\triangleright$  entra se  $\mathbf{x}$  non identifica un punto critico di  $f$ 
         $\mathbf{g} \leftarrow \nabla f(\mathbf{x})$                            $\triangleright$  muovi  $\mathbf{x}$  nella direzione di  $-\nabla f(\mathbf{x})$ 
    end while
    return  $\mathbf{x}$                                           $\triangleright$   $\mathbf{x}$  identifica un punto critico di  $f$ 
end function
```

L'algoritmo parte dall'approssimazione \mathbf{x} fornita e cerca un minimo locale in \mathbb{R}^n spostandosi, ad ogni iterazione, nella direzione identificata dal gradiente della funzione cambiato di segno. Siccome la direzione identificata dal gradiente cambiato di segno corrisponde alla direzione di massima velocità di decrescita della funzione, la ricerca dei minimi locali procede sempre nella direzione più promettente. L'algoritmo termina quando \mathbf{x} identifica un punto critico della funzione, che corrisponde quindi ad un minimo locale. Si noti che il passo di discesa α corrisponde alla velocità utilizzata dall'algoritmo per percorrere la discesa nella direzione identificata dal gradiente cambiato di segno.

Il seguente esempio mostra come sia possibile studiare il comportamento dell'algoritmo di discesa del gradiente in casi sufficientemente semplici. In particolare, l'esempio studia un casso in cui la funzione oggetto di studio ha un unico minimo globale.

✓ **Esempio 8.** Si consideri la seguente funzione $f : \mathbb{R}^2 \rightarrow \mathbb{R}$

$$f(x, y) = x^2 - 8x + y^2 + 10y + 28 \quad (66)$$

di cui si vuole trovare il minimo utilizzando l'algoritmo di discesa del gradiente. L'algoritmo viene applicato partendo da un punto $\mathbf{x}_0 = (x_0, y_0)$ e usando gli argomenti $\alpha \in (0, \frac{1}{2})$ e $\delta = 10^{-6}$. Il gradiente della funzione vale

$$\nabla f(x, y) = (2x - 8, 2y + 10) \quad (67)$$

e quindi, se $\mathbf{x}_i = (x_i, y_i)$ è l'approssimazione del risultato alla i -esima iterazione, allora

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \alpha \nabla f(\mathbf{x}_i) \quad (68)$$

e, quindi

$$\begin{aligned} x_{i+1} &= x_i - \alpha(2x_i - 8) = (1 - 2\alpha)x_i + 8\alpha \\ y_{i+1} &= y_i - \alpha(2y_i + 10) = (1 - 2\alpha)y_i - 10\alpha \end{aligned} \quad (69)$$

l'idea è quella
di applicare il
metodo di Newton
al gradiente

In particolare, è semplice verificare che

$$\begin{aligned} x_1 &= (1 - 2\alpha)x_0 + 8\alpha \\ x_2 &= (1 - 2\alpha)x_1 + 8\alpha = (1 - 2\alpha)^2 x_0 + 8\alpha(1 - 2\alpha) + 8\alpha \\ x_3 &= (1 - 2\alpha)x_2 + 8\alpha = (1 - 2\alpha)^3 x_0 + 8\alpha(1 - 2\alpha)^2 + 8\alpha(1 - 2\alpha) + 8\alpha \\ \dots \longrightarrow &\text{ci si ferma quando il modulo del gradiente si annulla o è più piccolo di } \delta \end{aligned} \quad (70)$$

Quindi, è semplice dimostrare per induzione che

$$x_i = (1 - 2\alpha)^i x_0 + 8\alpha \sum_{j=0}^{i-1} (1 - 2\alpha)^j \quad (71)$$

Questa relazione permette di studiare la convergenza dell'algoritmo. Infatti

$$\bar{x} = \lim_{i \rightarrow \infty} x_i = 8\alpha \sum_{j=0}^{\infty} (1 - 2\alpha)^j \quad \text{Convergenza dell'algoritmo} \quad (72)$$

siccome $0 < 1 - 2\alpha < 1$. L'ultima uguaglianza coinvolge una serie geometrica di ragione $0 < 1 - 2\alpha < 1$ e quindi

$$\bar{x} = \frac{8\alpha}{1 - 1 + 2\alpha} = 4 \quad (73)$$

che permette di dire che l'algoritmo di discesa del gradiente termina producendo un risultato che ha come prima componente un'approssimazione di $\bar{x} = 4$. In modo simile è possibile calcolare la seconda componente del risultato perché è semplice ottenere che

$$y_i = (1 - 2\alpha)^i y_0 - 10\alpha \sum_{j=0}^{i-1} (1 - 2\alpha)^j \quad (74)$$

e quindi

$$\bar{y} = \lim_{i \rightarrow \infty} y_i = -10\alpha \sum_{j=0}^{\infty} (1-2\alpha)^j = -\frac{10\alpha}{1-1+2\alpha} = -5 \quad (77)$$

Naturalmente, utilizzando la matrice hessiana è semplice calcolare che $(4, -5)$ corrisponde al minimo assoluto della funzione.

Si noti che per procedere ad uno studio più dettagliato del comportamento dell'algoritmo di discesa del gradiente in questa situazione è possibile notare come le due seguenti equazioni siano semplici relazioni di ricorrenza lineari e non omogenee

$$\begin{aligned} x_k - (1-2\alpha)x_{k-1} &= 8\alpha \\ y_k - (1-2\alpha)y_{k-1} &= -10\alpha \end{aligned} \quad (78)$$

Entrambe queste relazioni hanno il seguente polinomio caratteristico $\lambda - (1-2\alpha) = 0$ e quindi le soluzioni delle relazioni di ricorrenza omogenee associate diventano

$$\begin{aligned} \bar{x}_k &= c(1-2\alpha)^k \\ \bar{y}_k &= d(1-2\alpha)^k \end{aligned} \quad (79)$$

per opportuni $c \in \mathbb{R}$ e $d \in \mathbb{R}$. Due soluzioni particolari delle relazioni non omogenee si possono ottenere ipotizzando che queste soluzioni siano delle costanti $a \in \mathbb{R}$ e $b \in \mathbb{R}$, da cui si ottiene $a = 4$ e $b = -5$. Quindi, imponendo che i valori per $k = 0$ siano, rispettivamente, x_0 e y_0 , si ottiene

$$\begin{aligned} x_k &= (x_0 - 4)(1-2\alpha)^k + 4 \\ y_k &= (y_0 + 5)(1-2\alpha)^k - 5 \end{aligned} \quad (80)$$

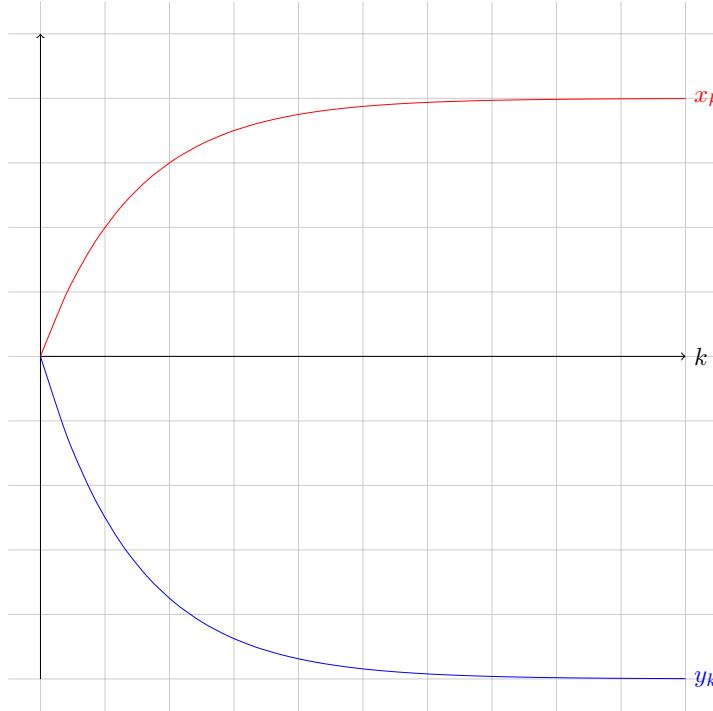


Figura 6: Le funzioni x_k (rossa) e y_k (blu) in (80) estese per $k \in [0, 10]$ partendo da $x_0 = y_0 = 0$ e usando $\alpha = \frac{1}{4}$

Appunti del Corso di Intelligenza Artificiale

Reti Neurali

Prof. Federico Bergenti

27 marzo 2024

1 Reti Neurali e Neuronali

Per la neurofisiologia, un cervello è un organo formato da cellule tra loro connesse che prendono il nome di **neuroni**. Nel suo complesso, quindi, un cervello è una **rete neuronale**. Nel caso specifico del cervello umano, normalmente si ritiene che:

1. Ogni cervello contenga un numero di neuroni dell'ordine di 10^{11} ;
2. Ogni cervello contenga 20 tipi di neuroni o poco più; e
3. Ogni cervello contenga un numero di collegamenti tra neuroni dell'ordine di 10^{14} .

Quindi, la rete neuronale che forma un cervello umano è caratterizzata da un numero enorme di neuroni quasi tutti uguali tra loro collegati mediante relativamente poche connessioni. Le connessioni consentono di trasportare informazioni attraverso la rete e ogni neurone riceve e invia informazioni da e per pochi altri neuroni con un tempo di reazione dell'ordine di 1 ms e, quindi, abbastanza lentamente.

Figura 1 mostra una schematizzazione delle parti principali di un neurone e, in particolare, mostra:

1. Il **soma**, che è la struttura cellulare che racchiude il **nucleo** di un neurone;
2. I **dendridi**, che sono i collegamenti in grado di veicolare informazioni mediante il movimento di ioni; e
3. Le **sinapsi**, che sono i punti di interfaccia tra diversi neuroni e si trovano nelle parti terminali delle **arborizzazioni** dell'**assone** di ogni neurone.

Il trasporto di informazioni tra i neuroni avviene mediante l'interazione tra dendridi e sinapsi di diversi neuroni. Quindi, ogni neurone ha una struttura semplice in grado di elaborare le informazioni veicolate dal trasporto di ioni mediante dendridi e sinapsi.

Una **rete neurale** (o **rete neurale artificiale**, da **Artificial Neural Network**, ANN) è uno strumento hardware/software in grado di simulare una rete di neuroni. Essa funziona con gli stessi principi della rete neuronale del cervello umano anche se, normalmente, una rete neurale è strutturalmente molto più semplice di un cervello umano. In più, le reti neurali più comuni hanno un'architettura dei collegamenti tra i neuroni progettata per uno specifico scopo. Quindi, anche se è possibile dire che una rete neurale sia un simulatore di un semplice cervello umano, le reti neurali sono progettate per specifici scopi e solo marginalmente sono pensate per riprodurre i fenomeni neurofisiologici.

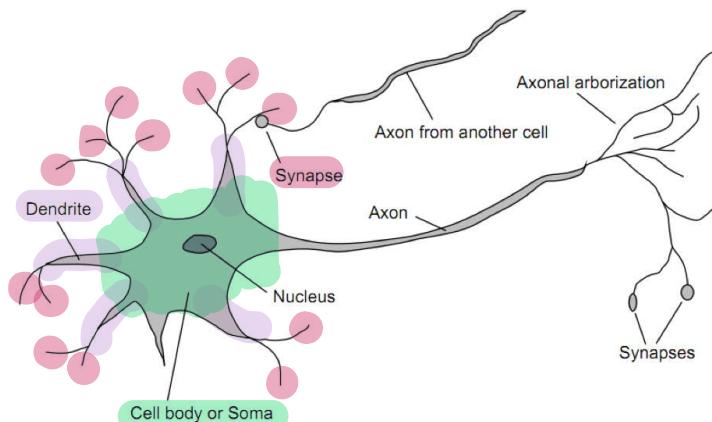


Figura 1: La schematizzazione di un neurone

Una rete neurale simula il comportamento di una rete di neuroni andando a simulare il comportamento dei singoli neuroni. Il simulatore di un neurone viene detto **unità di McCulloch-Pitts**. L'unità di McCulloch-Pitts non è altro che un modulo hardware/software pensato per simulare il comportamento di un neurone dal punto di vista del trasporto e dell'elaborazione delle informazioni. Infatti, un'unità di McCulloch-Pitts riceve informazioni rappresentate mediante numeri reali tramite una quantità prefissata di canali di comunicazione. Poi, l'unità elabora l'informazione ricevuta producendo un valore numerico legato ai valori ricevuti dai canali di comunicazione. Si noti che, spesso, quando si parla di reti neurali, le unità di McCulloch-Pitts sono chiamate semplicemente neuroni.

In sostanza, un'unità di McCulloch-Pitts non è altro che una funzione reale di $n \in \mathbb{N}_+$ variabili reali, dove n è noto e potenzialmente diverso per ogni neurone simulato. La funzione realizzata da un'unità di McCulloch-Pitts con $n \in \mathbb{N}_+$ ingressi è molto semplice e può essere descritta come segue:

*ricorda che ogni neurone ha i suoi canali
→ sono tipici anche dei neuroni

$$y = g \left(\sum_{i=1}^n w_i x_i - w_0 \right)$$

f.n.e di attivazione

questa uscita dal neurone andrà collegata come entrata ad un altro neurone di McCulloch-Pitts

valore prodotto

pesi (tipici del canale)*

peso di bias (non voglio che ad ingresso nullo valga 0, sottraggo un coefficiente tipico del neurone)

→ valore di soglia = dove si trova lo 0

La singola uscita y , la posso calcolare andando a prendere i singoli ingressi x_i , con i che va da 1 a n , moltiplicando i singoli ingressi per dei coefficienti specifici del canale e faccio passare tutto per una f.n.e lineare detta f.n.e di attivazione

dove $y \in \mathbb{R}$ è il valore prodotto dall'unità a fronte degli n ingressi reali $(x_i)_{i=1}^n$ utilizzando n coefficienti reali $(w_i)_{i=1}^n$ detti **pesi**, un coefficiente reale aggiuntivo w_0 detto **peso di bias** e una funzione $g : \mathbb{R} \rightarrow \mathbb{R}$ detta **funzione di attivazione**. Nell'ambito di una rete neurale, il risultato del calcolo di un'unità di McCulloch-Pitts viene spesso utilizzato come ingresso di un'altra unità collegata alla rete. In questo modo, una rete neurale è in grado di compiere elaborazioni complesse essenzialmente legate alla struttura della rete e non alle capacità di calcolo delle unità di McCulloch-Pitts.

Si noti che un'unità di McCulloch-Pitts viene descritta completamente dai pesi, comprensivi del peso di bias, e dalla funzione di attivazione. Normalmente, la funzione di attivazione è fissata per tutta la rete e quindi la descrizione di un'unità di McCulloch-Pitts si riduce all'enumerazione dei pesi. Quindi, spesso, si preferisce estendere gli n ingressi con un ingresso aggiuntivo fissato al valore -1 in modo da poter descrivere un'unità di McCulloch-Pitts solo mediante un vettore dei pesi $\mathbf{w} = (w_i)_{i=1}^{n+1}$, dove w_{n+1} è il peso di bias. Noto \mathbf{w} , l'elaborazione dell'unità è espressa da

$$y = g(\mathbf{x} \cdot \mathbf{w}) \quad (2)$$

dove $\mathbf{x} = (x_i)_{i=1}^{n+1}$ e $x_{n+1} = -1$.

vettore dei pesi + peso di bias

f.n.e di attivazione

prodotto scalare

Normalmente, la funzione di attivazione di un'unità di McCulloch-Pitts non è un'arbitraria funzione reale di variabile reale, ma viene scelta tra una delle seguenti funzioni di uso comune. Si noti che, oltre a quelle discusse, esistono tante altre funzioni di attivazione utilizzate per applicazioni reali, ma quelle elencate nel seguito sono tra quelle più utilizzate.

La prima possibilità, che è la più semplice e la meno usata, è di scegliere come funzione di attivazione la **funzione identità**

$$\text{Id}(x) = x \quad (3)$$

Si noti che questa scelta rende l'unità di McCulloch-Pitts una funzione lineare. La seconda possibilità, spesso utilizzata per la sua semplicità realizzativa, è la **funzione ReLU** (da **Rectifier Linear Unit**)

$$R(x) = \max\{0, x\} \quad \begin{array}{l} \text{per } x < 0, \\ \text{la f.n. ritorna 0} \end{array} \quad (4)$$

La terza possibilità, spesso utilizzata per elaborare valori binari mediante reti neurali, è la **funzione di Heaviside** (con $H(0) = 1$)

$$H(x) = \begin{cases} 1 & \text{se } x \geq 0 \\ 0 & \text{se } x < 0 \end{cases} \quad (5)$$

La quarta possibilità, simile alla precedente, è la **funzione signum** (o segno)

$$\text{sgn}(x) = \begin{cases} 1 & \text{se } x > 0 \\ 0 & \text{se } x = 0 \\ -1 & \text{se } x < 0 \end{cases} \quad (6)$$

La quinta possibilità, che è la più frequente perché è non lineare e derivabile ovunque, è la **funzione logistica** (o **funzione sigmoide**)

$$S(x) = \frac{1}{1 + e^{-x}} \quad (7)$$

Figura 2 mostra due delle cinque funzioni di attivazioni discusse in precedenza, la funzione ReLU e la funzione sigmoide.

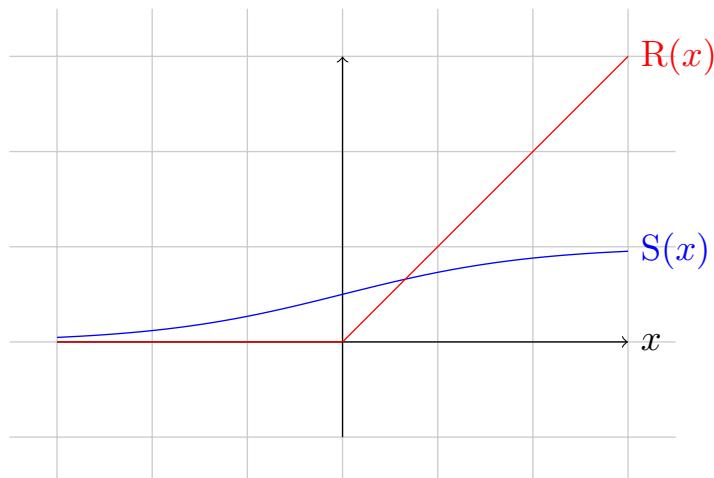


Figura 2: Le funzioni $R(x)$ (rossa) e $S(x)$ (blu)

Una rete neurale è quindi un grafo orientato pesato i cui nodi sono unità di McCulloch-Pitts e, per ogni unità, i pesi sugli archi sono i pesi in ingresso all'unità, comprensivi del peso di bias associato ad un ingresso aggiuntivo fissato a -1 . Normalmente, non si ammettono autoanelli in una rete neurale in modo da non dover tenere conto dei ritardi introdotti dalle singole unità di McCulloch-Pitts. In generale, per una rete neurale non è possibile identificare quale sia l'ingresso e quale sia l'uscita, anche se è sempre possibile identificare quale sia l'ingresso e quale sia l'uscita di una singola unità di McCulloch-Pitts perché il grafo è orientato. Si noti che è sempre possibile ipotizzare che una rete neurale sia un grafo completamente connesso, eventualmente utilizzando dei pesi nulli per indicare l'assenza di archi tra unità di McCulloch-Pitts.

Data una rete neurale con $n \in \mathbb{N}_+$ neuroni, fissato un qualsiasi istante $t \in \mathbb{R}_{>0}$, il vettore $\mathbf{s}(t) \in \mathbb{R}^n$ formato delle uscite correnti dei neuroni viene detto **stato della rete** all'istante t . Lo stato della rete evolve dinamicamente partendo da uno stato iniziale $\mathbf{s}(0)$ che, tipicamente, si assume noto. In generale, però, non è detto che una rete neurale raggiunga uno stato finale.

Siccome, spesso, le unità di McCulloch-Pitts utilizzate in una rete neurale sono tutte caratterizzate dalla stessa funzione di attivazione, la descrizione della rete si riduce all'enumerazione di tutti i pesi associati agli archi del grafo completamente connesso che descrive la rete. Quindi, senza perdere di generalità, pur nell'ipotesi che tutte le unità di McCulloch-Pitts utilizzino la stessa funzione di attivazione, è possibile dire che una rete neurale viene descritta completamente una volta che siano noti i pesi associati agli archi che collegano le unità della rete.

Si noti che, una volta identificati i pesi adeguati, è anche possibile utilizzare una rete neurale per realizzare funzioni dello stato della rete. In questi casi, si ipotizza che esistano delle unità di McCulloch-Pitts che possano ricevere valori da elaborare dall'esterno. Coerentemente, si ipotizza che esistano delle unità di McCulloch-Pitts i cui risultati vengano resi disponibili verso l'esterno.

**spiegazione
file a parte**

Esempio 1. Un'unità di McCulloch-Pitts che utilizza la funzione di Heaviside come funzione di attivazione può essere utilizzata per realizzare le funzioni dell'algebra di Boole sull'insieme $\{0, 1\}$. Come mostrato in Figura 1(a), se la funzione da realizzare ha $n \in \mathbb{N}_+$ ingressi, allora l'unità avrà n ingressi collegati con l'esterno, un ingresso fissato a -1 per il peso di bias, e un'uscita collegata con l'esterno.

Naturalmente, i pesi indicati in Figura 1(a) non sono gli unici possibili. In generale, se la funzione da realizzare deve associare un ingresso $\mathbf{x} = (x_i)_{i=1}^3$, con $x_3 = -1$, al valore 1, allora sarà sufficiente trovare un vettore dei pesi $\mathbf{w} = (w_i)_{i=1}^3$ tale che $\mathbf{w} \cdot \mathbf{x} \geq 0$. In modo simile, se l'uscita deve valere 0, allora dovrà essere $\mathbf{w} \cdot \mathbf{x} < 0$. Però, non è possibile trovare un vettore dei pesi adeguato a realizzare la funzione XOR, come mostrato in Figura 1(b). Infatti, non è possibile trovare un vettore dei pesi per cui valgano contemporaneamente le seguenti disuguaglianze:

$$\begin{array}{ll} \mathbf{w} \cdot (1, 0, -1) \geq 0 & \mathbf{w} \cdot (0, 1, -1) \geq 0 \\ \mathbf{w} \cdot (0, 0, -1) < 0 & \mathbf{w} \cdot (1, 1, -1) < 0 \end{array} \quad \begin{array}{l} \text{l'elemento del vettore} \\ \mathbf{x} \text{ in } 3^{\text{a}} \text{ posizione è} \\ \text{sempre } -1 \end{array} \quad (8)$$

perché queste disuguaglianze equivalgono alle seguenti disuguaglianze

$$\begin{array}{ll} w_1 - w_3 \geq 0 & w_2 - w_3 \geq 0 \\ -w_3 < 0 & w_1 + w_2 - w_3 < 0 \end{array} \quad (9)$$

ma $w_1 \geq w_3 > 0$ e $w_2 \geq w_3 > 0$ sono incompatibili con $w_3 > w_1 + w_2$.

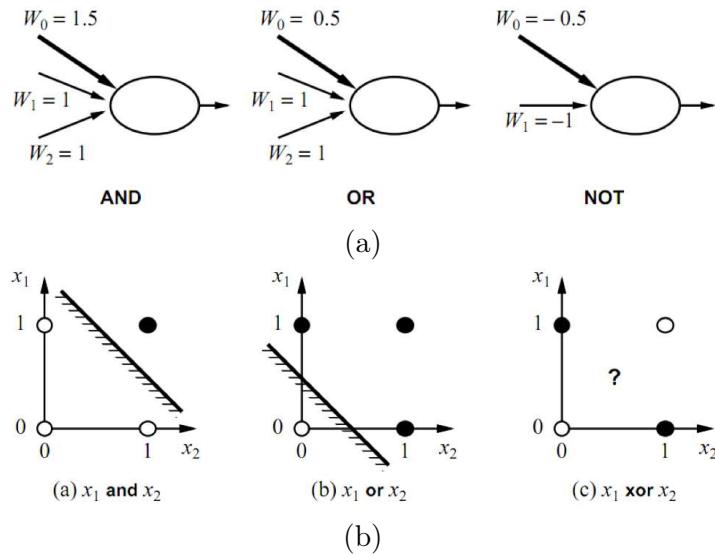


Figura 3: Le funzioni principali dell'algebra di Boole (a) realizzate mediante singole unità di McCulloch-Pitts e (b) realizzate, ad eccezione della funzione XOR, con la stessa tecnica

Da questo punto di vista, è quindi possibile costruire una rete che abbia un comportamento desiderato andando ad associare un peso adeguato ad ogni arco. Questo approccio, però, pur essendo idealmente percorribile, non è praticabile perché il numero di archi con peso non nullo è normalmente molto elevato. Quindi, per la costruzione di una rete, si ricorre spesso all'**apprendimento**, che prevede che i pesi della rete vengano ottenuti partendo dalle caratteristiche di alcuni esempi. Si dice, quindi, che una rete viene **addestrata** fornendole degli esempi che vengono raccolti in un **training set**.

Si noti che, in base alle caratteristiche degli esempi forniti per l'addestramento, si può parlare di due tipologie principali di addestramento:

1. **Addestramento supervisionato:** in cui viene fornito un insieme di esempi e, per ogni esempio, viene anche detto qual è l'uscita attesa per alcune unità di McCulloch-Pitts della rete che rivestono un particolare interesse.
2. **Addestramento non supervisionato:** in cui viene fornito un insieme di esempi e, sfruttandone le caratteristiche, si cerca di ottenere un insieme di pesi adeguato. *

L'addestramento consente di ottenere un insieme adeguato di pesi in grado di garantire che la rete si comporti bene nei casi d'esempio. Però, per valutare quanto bene la rete si comporti in situazioni non ricomprese negli esempi forniti, le prestazioni di una rete vengono sempre valutate utilizzando un secondo insieme di esempi detto **test set**. Training set e test set devono essere disgiunti ed è possibile dire che una rete **generalizza** se ha un comportamento adeguato anche per gli esempi del test set.

Quindi, come spesso si dice, una rete neurale non viene programmata ma viene addestrata ad avere dei comportamenti. Il **machine learning** (o apprendimento automatico) è quella disciplina che si occupa di studiare questo approccio alla sintesi di sistemi di calcolo, sia per reti neurali che per altri tipi di sistemi di calcolo. Quando le informazioni imparate mediante le tecniche del machine learning sono di tipo numerico, o riconducibili a tali, allora si parla di intelligenza artificiale **subsimbolica**.

* esempio che spiega cosa vuol dire che sfrutta le caratteristiche: si prende l'input (vettore di n.r. R dato che sono gli stati di alcuni nodi), si mettono in questo spazio R^n e ci si accorgere che gli input sono tutti raggruppati vicini da una parte e tutti raggruppati vicini dall'altra parte. Quando sono vicini da una parte la rete li deve considerare tutti simili, quando sono vicini tutti dall'altra parte la rete li deve considerare tutti simili. La rete non deve considerare simili quelli da un gruppo o dall'altro. L'algoritmo di addestramento, quello che fa è prendere i valori e cercare di capire quando sono abbastanza simili e trovare dei pesi che fanno in modo che se sono simili, l'uscita ha un certo valore; se sono simili dall'altra parte ha un altro valore, ma non glielo si dice dall'esterno qual è l'uscita, non gli si dice nemmeno i cluster che si studiano, è l'algor. per come è fatto, che prende gli esempi e valuta. Il NON SUP. non prevede l'algoritmo che dice qual è l'uscita corretta.

L'alg. di back propagation non si applica a tutte le reti neurali ma alle reti neurali in avanti.

insieme di alberi disgiunti, in cui l'uscita è la radice di ogni albero. Più uscite ho e più la rete sarà in grado di produrmi valori in R^m

2 Single-Layer Perceptron e Approssimazione di Funzioni

Una rete neurale in avanti (o feed forward) è un grafo orientato aciclico pesato organizzato a strati (o livelli). I nodi del grafo sono unità di McCulloch-Pitts e valgono le seguenti proprietà:

1. Gli strati sono numerati;
2. Ogni nodo appartiene ad un solo strato;
3. I nodi del primo strato non hanno archi entranti e il valore di uscita delle relative unità di McCulloch-Pitts viene fornito dall'esterno;
4. Ogni nodo, tranne quelli del primo strato, ha archi entranti che provengono solo dai nodi dello strato precedente;
5. I nodi dell'ultimo strato non hanno archi uscenti e il valore di uscita delle relative unità di McCulloch-Pitts viene fornito all'esterno;
6. I pesi sugli archi sono i pesi entranti nelle unità di McCulloch-Pitts; e
7. Uno dei valori di ingresso per ogni strato della rete viene fissato al valore di bias -1 e non riceve archi entranti.

Se non viene esplicitamente indicato il contrario, si assume che una rete neurale in avanti abbia il massimo numero di connessioni che le consentono di mantenere la struttura a strati. Si noti che gli strati diversi dal primo e dall'ultimo vengono normalmente chiamati strati nascosti. Infine, si noti che una rete neurale in avanti viene anche chiamata perceptron (o percepitrone) e, quando la struttura della rete non è a strati, spesso si parla di rete neurale ricorrente, anziché semplicemente di rete neurale.

Ad esempio, in Figura 4 viene mostrata una rete neurale in avanti con tre ingressi forniti da tre delle unità di McCulloch-Pitts del primo strato, quattro unità di McCulloch-Pitts nel primo strato nascosto, quattro unità di McCulloch-Pitts nel secondo strato nascosto e un'uscita fornita dall'unica unità di McCulloch-Pitts dell'ultimo strato. Questa rete ha complessivamente nove unità di McCulloch-Pitts che compiono elaborazioni, quattro nel primo strato nascosto, quattro nel secondo strato nascosto e una nello strato di uscita.

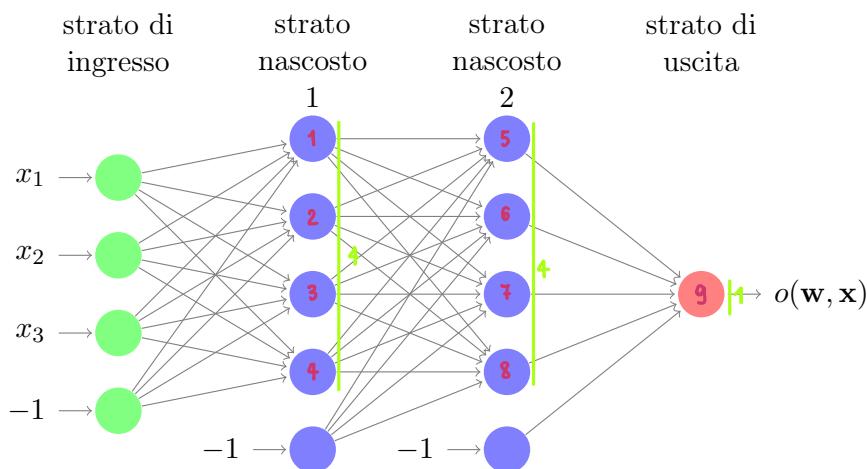


Figura 4: Esempio di una rete neurale in avanti formata da 3 strati (lo strato di input non fa calcoli quindi non si conta)

multi layer perceptron
4-4-1

Un **Single-Layer Perceptron (SLP)** è una rete neurale in avanti con $n \in \mathbb{N}_+$ ingressi reali, nessuno strato nascosto, $m \in \mathbb{N}_+$ uscite reali e una funzione di attivazione $g : \mathbb{R} \rightarrow \mathbb{R}$. Si noti che si utilizza la notazione (n, m) -SLP per indicare esplicitamente n e m , mentre si parla semplicemente di SLP se n e m sono noti dal contesto. Come discusso nel seguito, un (n, m) -SLP può essere utilizzato come un approssimatore di una funzione $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ di cui si conoscano i valori per alcuni vettori. Alcuni dei vettori per cui la funzione è nota vengono quindi raccolti in un training set, e i rimanenti vengono raccolti in un test set, in modo da poter utilizzare l'addestramento supervisionato per trovare i pesi del SLP in grado di approssimare la funzione oggetto di studio. Si noti subito che il numero di ingressi n e il numero di uscite m fissano immediatamente la struttura del SLP e quindi è ragionevole ritenere che un SLP non possa approssimare una funzione arbitraria da \mathbb{R}^n a \mathbb{R}^m . Figura 5 mostra un SLP con quattro ingressi e un'uscita.

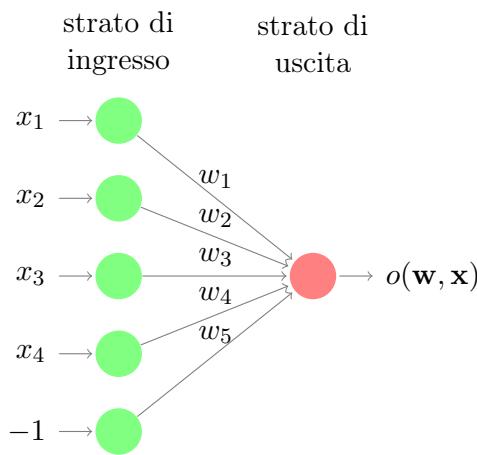


Figura 5: Esempio di un $(4, 1)$ -SLP

Si consideri un $(n-1, 1)$ -SLP e, per semplificare la notazione, senza comunque perdere di generalità, si ipotizzi che tutti i vettori di \mathbb{R}^n utilizzati come ingressi del SLP, compresi quelli raccolti nel training set e nel test set, siano del tipo

$$\mathbf{x} = (x_1, x_2, \dots, x_{n-1}, x_n = -1) \quad (10)$$

Naturalmente, se si vuole effettivamente disporre di n valori di ingresso per il SLP, sarà sufficiente considerare un $(n, 1)$ -SLP.

Dato un vettore di ingresso $\mathbf{x} \in \mathbb{R}^n$, il valore calcolato dal SLP per un certo vettore dei pesi $\mathbf{w} \in \mathbb{R}^n$ vale

$$o(\mathbf{w}, \mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x}) = g\left(\sum_{i=1}^{n-1} w_i x_i - w_n\right) \quad (11)$$

parte lineare del calcolo dell'unità

toglie il peso di bias

fine attiv.

somma pesata dei corrispettivi pesi

quello che si vuol fare è andare a minimizzare il più possibile il valore assoluto di $|f(x) - \hat{f}(x)|$ (differenza tra output atteso e output ottenuto). Se $f(x)$ e x sono fissati, perché l'ultima componente del vettore \mathbf{x} , quindi x_n , consente di trattare l'ultima componente di \mathbf{w} , quindi w_n , come peso di bias.

L'addestramento supervisionato ha lo scopo di trovare un vettore dei pesi del SLP in modo da minimizzare la distanza media tra il valore calcolato dal SLP e il corrispondente valore della funzione da approssimare. Si noti che il valore della funzione da approssimare è noto per ipotesi per tutti i vettori del training set, appunto perché si sta considerando un addestramento supervisionato. In più, si noti che un vettore dei pesi che minimizza la distanza media tra il valore calcolato dal SLP e il corrispondente valore della funzione da approssimare è tale da minimizzare le singole distanze tra il valore calcolato dal SLP

e il corrispondente valore della funzione da approssimare perché le distanze sono sempre positive o nulle. Quindi, è sufficiente cercare di minimizzare la distanza tra il valore calcolato dal SLP e il corrispondente valore della funzione da approssimare per ogni vettore del training set per trovare un vettore dei pesi adeguato.

Detto $\mathbf{x} \in \mathbb{R}^n$ uno dei vettori nel training set, l'errore compiuto dal SLP nell'approssimazione della funzione f per lo specifico \mathbf{x} vale

$$\text{fissato } \mathbf{x} \text{ e } f(\mathbf{x}) \quad * \epsilon(\mathbf{w}, \mathbf{x}) = f(\mathbf{x}) - o(\mathbf{w}, \mathbf{x}) = f(\mathbf{x}) - g(\mathbf{w} \cdot \mathbf{x}) \quad (12)$$

Si noti che l'approssimazione della funzione f per il vettore \mathbf{x} è tanto migliore quanto $|\epsilon(\mathbf{w}, \mathbf{x})|$ è piccolo. Quindi, la scelta del vettore dei pesi sarà tanto migliore quanto sarà in grado di rendere $|\epsilon(\mathbf{w}, \mathbf{x})|$ piccolo. In sintesi, fissato un vettore \mathbf{x} , il problema di individuare un vettore dei pesi \mathbf{w} in grado di approssimare bene $f(\mathbf{x})$ può essere espresso mediante la ricerca di un \mathbf{w} che minimizzi $|\epsilon(\mathbf{w}, \mathbf{x})|$.

Con l'obiettivo di utilizzare le comuni tecniche di ricerca dei minimi di funzioni reali in più variabili reali, normalmente si assume che la funzione di attivazione g sia di classe $C^\infty(\mathbb{R})$ e si cerca il minimo dell'errore quadratico

a noi interessa il valore assoluto di ϵ , un modo per minimizzare il v. assoluto di qualcosa è minimizzarne il quadrato, quindi anziché utilizzare il v. assoluto (che da problemi in 0 con le derivate) usiamo ϵ^2 : è un modo per studiare un f.n. diversa che ha la caratteristica "è minima dove è minima" il v. assoluto in comune con il v. ass. e ϵ^2 divenne derivabile anziché di $|\epsilon(\mathbf{w}, \mathbf{x})|$. Naturalmente, un vettore dei pesi \mathbf{w} in grado di rendere minimo $\epsilon(\mathbf{w}, \mathbf{x})$ è anche in grado di rendere minimo $|\epsilon(\mathbf{w}, \mathbf{x})|$.

ci permettiamo di usarlo perché serve unicamente a mandarlo via quando calcoliamo le derivate. Non serve a niente e lo si mette (13) cosicché non rimanga un $2x$ nella formula finale

cerchiamo il vettore \mathbf{w} per cui $\frac{1}{2} \epsilon^2(\mathbf{x}, \mathbf{w})$ risulta minima.

Non ci sarà un unico minimo, non necessariamente si troverà il min. globale

In sintesi, fissato un vettore $\mathbf{x} \in \mathbb{R}^n$, l'errore quadratico è una funzione di \mathbf{w} di cui si può cercare il minimo mediante l'algoritmo di discesa del gradiente. Quindi, impostando il problema di approssimare f in questi termini, è necessario esprimere il gradiente di $e(\mathbf{w})$ in modo da utilizzarlo per aggiornare i pesi del SLP mediante la regola

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla e(\mathbf{w}) \quad \begin{array}{l} \xrightarrow{\text{coefficiente di apprendimento}} \\ \xrightarrow{\text{il nuovo vettore delle variabili lo otteniamo sottraendo alpha per il gradiente dell'errore quadratico medio (14) nello stesso vettore di partenza}} \end{array}$$

dove $\alpha \in \mathbb{R}_+$ prende il nome di coefficiente di apprendimento (o learning rate).

Per potere applicare l'algoritmo di discesa del gradiente è necessario esprimere il gradiente di $e(\mathbf{w})$ in funzione di \mathbf{w} mediante il calcolo delle n derivate parziali di $e(\mathbf{w})$. Quindi, fissato $1 \leq i \leq n$, quello che interessa è

$$\boxed{\text{ID1} \quad \frac{\partial e}{\partial w_i}(\mathbf{w}) = \frac{1}{2} \cdot 2 \cdot \epsilon(\mathbf{w}) \frac{\partial \epsilon}{\partial w_i}(\mathbf{w}) = \epsilon(\mathbf{w}) \frac{\partial \epsilon}{\partial w_i}(\mathbf{w})} \quad (15)$$

ma

$$\frac{\partial \epsilon}{\partial w_i}(\mathbf{w}) = -\frac{\partial o}{\partial w_i}(\mathbf{w}) \quad \begin{array}{l} \xrightarrow{\text{derivata di epsilon}} \\ \xrightarrow{\text{l'uscita prodotta dall'unità di McCulloch Pitts una volta fissato w e x}} \end{array} \quad (16)$$

perché $f(\mathbf{x})$ non dipende dal vettore dei pesi. Quindi, ricordando che l'uscita del SLP dipende unicamente da \mathbf{w} perché \mathbf{x} è fissato, si ottiene

$$\frac{\partial o}{\partial w_i}(\mathbf{w}) = \frac{\partial g}{\partial w_i}(\mathbf{w} \cdot \mathbf{x}) = g'(\mathbf{w} \cdot \mathbf{x}) \frac{\partial (\mathbf{w} \cdot \mathbf{x})}{\partial w_i} = g'(\mathbf{w} \cdot \mathbf{x}) x_i \quad (17)$$

perché

La derivata di o in w_i è la derivata di g applicato al prodotto scalare $w \cdot x$

$$\frac{\partial (\mathbf{w} \cdot \mathbf{x})}{\partial w_i} = \sum_{j=1}^n x_j \frac{\partial w_j}{\partial w_i} = x_i \quad \begin{array}{l} \xrightarrow{\text{derivando in } w_i, \text{ gli altri componenti non i posizioni } i, \text{ sono costanti e quindi si possono non considerare}} \\ (18) \end{array}$$

visto che è semplice constatare che

$$\frac{\partial w_j}{\partial w_i} = \begin{cases} 1 & \text{se } i = j \\ 0 & \text{altrimenti} \end{cases} \quad (19)$$

Fissato un w_i iniziale, lo correggiamo per ottenere un w_i al ciclo successivo aggiungendo α (coeff. di apprendimento) per $\epsilon(w)$ per la derivata di g in w_i (= il componente i-esimo del gradiente).

Questo algoritmo ci dice che ci muoveremo andando nella discesa del gradiente (verso dove diminuisce), ottenendo nuovi pesi tutte le volte, e ogni volta, avendo un'approssimazione migliore della funzione che stiamo cercando di approssimare.

Per questo alg. modifichiamo il criterio d'arresto: non più quando il gradiente è molto prossimo a 0, ma ci fermiamo quando l'errore medio compiuto su tutto il training set è abbastanza piccolo. Questo NON va bene, fissiamo l'arresto dopo tot epoch (=giro completo sul training set).

Quindi, in sintesi, fissato $x \in \mathbb{R}^n$ nel training set è possibile aggiornare il vettore dei pesi del SLP in modo da spostare i pesi verso un minimo locale dell'errore compiuto nell'approssimazione di f usando la regola

$$w_i \leftarrow w_i + \alpha \epsilon(w) g'(\mathbf{w} \cdot \mathbf{x}) x_i = w_i + \alpha (f(\mathbf{x}) - g(\mathbf{w} \cdot \mathbf{x})) g'(\mathbf{w} \cdot \mathbf{x}) x_i \quad (20)$$

per $1 \leq i \leq n$ e assumendo che l'ultima componente di \mathbf{x} valga -1 .

Si noti che la formula per aggiornamento dei pesi di un SLP è spesso espressa nell'ipotesi che la funzione di attivazione g sia la funzione sigmoide

$$S(x) = \frac{1}{1 + e^{-x}} \quad (21)$$

perché vale la seguente proprietà della derivata della funzione sigmoide

$$S'(x) = S(x)(1 - S(x)) \quad (22)$$

e quindi

$$S'(\mathbf{w} \cdot \mathbf{x}) = S(\mathbf{w} \cdot \mathbf{x})(1 - S(\mathbf{w} \cdot \mathbf{x})) = o(\mathbf{w}, \mathbf{x})(1 - o(\mathbf{w}, \mathbf{x})) \quad (23)$$

che permette di aggiornare i pesi senza dover necessariamente valutare esplicitamente la derivata della funzione di attivazione.

Aver esplicitato la formula per l'aggiornamento dei pesi di un SLP in modo da cercare di approssimare una funzione f di cui si conoscano alcuni valori permette di esplicitare l'algoritmo di addestramento supervisionato per un SLP. Questo algoritmo è descritto nell'Algoritmo 2 e prevede di partire da un vettore dei pesi iniziale, tipicamente casuale, aggiornando questo vettore dei pesi per ogni vettore del training set. In particolare, l'algoritmo descritto nell'Algoritmo 2 lavora su un $(n-1, m)$ -SLP e ha i seguenti parametri:

- T : il training set formato da coppie $(\mathbf{x}, f(\mathbf{x}))$;
- α : il coefficiente di apprendimento;
- δ : la tolleranza accettata sull'errore medio; e
- ρ : il numero massimo di epoch.

L'algoritmo termina producendo un vettore dei pesi in grado di approssimare la funzione di cui si conoscono alcuni valori quando:

1. L'errore medio calcolato su tutto il training set è sufficientemente piccolo; oppure
2. È stato raggiunto un numero massimo di epoch (di apprendimento), dove un'epoch non è altro che un'iterazione completa dell'algoritmo su tutto il training set.

Si noti che la terminazione causata dal raggiungimento del numero massimo di epoch non garantisce che l'algoritmo trovi sempre un'approssimazione sufficientemente buona della funzione. Però, è necessario introdurre questa condizione di terminazione perché non è possibile garantire che esista un vettore dei pesi in grado di rendere l'errore medio sul training set sufficientemente piccolo. In altri termini, non è possibile garantire che un SLP sia in grado di approssimare bene quanto si vuole una qualsiasi funzione. Come caso estremo, ad esempio, si può considerare il tentativo di utilizzare un SLP per approssimare la funzione XOR dell'algebra di Boole, come discusso nell'Esercizio 1.

Si noti che queste condizioni di terminazione non sono quelle dell'algoritmo di discesa del gradiente comunemente utilizzato e descritto nell'Algoritmo 1. Infatti, queste condizioni

Algoritmo 2 Algoritmo di addestramento supervisionato per un $(n - 1, 1)$ -SLP

```

function SLP_LEARN( $T, \alpha, \delta, \rho$ )
    randomize  $\mathbf{w} \in \mathbb{R}^n$                                  $\triangleright$  inizializza  $\mathbf{w}$ 
    for  $1 \leq r \leq \rho$  do
         $\bar{\epsilon} \leftarrow 0$                                  $\triangleright$  ripeti per non più di  $\rho$  epochi
        for  $(\mathbf{x}, f(\mathbf{x})) \in T$  do
             $a \leftarrow \mathbf{w} \cdot \mathbf{x}$                        $\triangleright$  inizializza l'errore medio sul training set
            for  $1 \leq i \leq n$  do                             $\triangleright$  per ogni vettore del training set
                 $w_i \leftarrow w_i + \alpha (f(\mathbf{x}) - g(a)) g'(a) x_i$        $\triangleright$  inizializza  $a \in \mathbb{R}$ 
            end for                                          $\triangleright$  per ogni componente di  $\mathbf{x}$ 
             $\bar{\epsilon} \leftarrow \bar{\epsilon} + \frac{|f(\mathbf{x}) - g(\mathbf{w} \cdot \mathbf{x})|}{|T|}$            $\triangleright$  aggiorna il peso  $w_i$ 
        end for                                          $\triangleright$  aggiorna l'errore medio sul training set
        if  $\bar{\epsilon} < \delta$  then                                 $\triangleright$  se l'errore medio è sufficientemente piccolo
            return  $\mathbf{w}$                                  $\triangleright$   $\mathbf{w}$  è il vettore dei pesi cercato
        end if
    end for
    return  $\mathbf{w}$                                      $\triangleright$   $\mathbf{w}$  è il vettore dei pesi cercato dopo  $\rho$  epochi
end function

```

di terminazione si adattano meglio all'addestramento supervisionato visto che si basano sull'errore compiuto nell'approssimazione e non sul gradiente di questo errore. Infine, si noti che, alle volte, queste condizioni di terminazione vengono sostituite da altre che tengono conto delle caratteristiche specifiche della funzione che si sta cercando di approssimare.

L'algoritmo descritto nell'[Algoritmo 1](#) può essere facilmente esteso per utilizzare un (n, m) -SLP, tipo quello mostrato in Figura 6, per approssimare una funzione $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$. Per farlo è sufficiente disporre di un vettore dei pesi per ognuna delle m uscite richieste. In questo caso, gli m vettori dei pesi richiesti vengono raggruppati in un **2-tensore**, che non è altro che un elemento di $(\mathbb{R}^{n+1})^m$. Quindi, si potrà usare un 2-tensore $W = (\mathbf{w}_i)_{i=1}^m$ le cui m componenti $\mathbf{w}_i \in \mathbb{R}^{n+1}$, con $1 \leq i \leq m$, sono i vettori dei pesi necessari ad approssimare la funzione \mathbf{f} . Si noti che, una volta introdotta la nomenclatura dei tensori, è comune indicare i vettori di \mathbb{R}^n , per un qualche $n \in \mathbb{N}_+$ con il termine **1-tensori**.

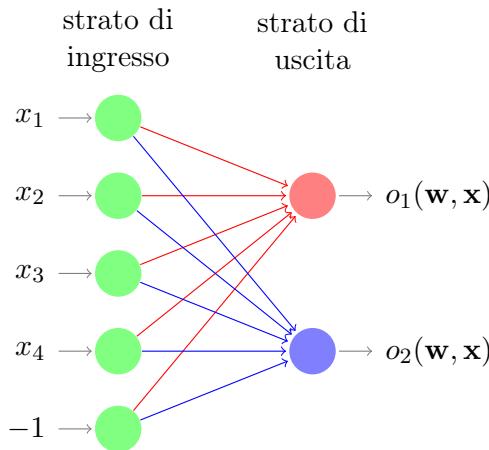


Figura 6: Esempio di un $(4, 2)$ -SLP

3 Multi-Layer Perceptron e Approssimazione di Funzioni

Come discusso in precedenza, un **Multi-Layer Perceptron (MLP)** è una rete neurale in avanti con uno o più ingressi reali, degli strati nascosti, ognuno dei quali è caratterizzato da un numero specifico di neuroni, e una o più uscite reali. Per semplificare la notazione, ottenendo comunque risultati interessanti, conviene concentrarsi su MLP con un solo strato nascosto e una sola uscita. In questa ipotesi, si parla di $(n, m, 1)$ -MLP per indicare un MLP con $n \in \mathbb{N}_+$ ingressi, un solo strato nascosto caratterizzato da $m \in \mathbb{N}_+$ neuroni e un'uscita. Si noti che, in analogia a quanto fatto per i SLP, i valori di bias vengono gestiti mediante neuroni aggiuntivi nello strato di ingresso e nell'unico strato nascosto. In Figura 7 viene mostrato un $(3, 4, 1)$ -MLP.

Si consideri un $(n - 1, m - 1, 1)$ -MLP caratterizzato da una funzione di attivazione $g : \mathbb{R} \rightarrow \mathbb{R}$ utilizzata in tutte le unità di McCulloch-Pitts. L'obiettivo è usare il MLP come un approssimatore di una funzione $f : \mathbb{R}^n \rightarrow \mathbb{R}$ di cui si conoscano i valori per alcuni vettori. Come nel caso del SLP, alcuni dei vettori in cui la funzione è nota vengono raccolti in un training set, mentre i rimanenti vengono raccolti in un test set. Per semplificare la notazione, senza comunque perdere di generalità, si ipotizza che tutti i vettori $\mathbf{x} \in \mathbb{R}^n$ utilizzati come ingressi del MLP, compresi quelli raccolti nel training set e nel test set, siano del tipo

$$\mathbf{x} = (x_1, x_2, \dots, x_{n-1}, x_n = -1) \quad (24)$$

dimensione vettori di input

In più, si ipotizza che tutti i vettori $\mathbf{y} \in \mathbb{R}^m$ che vengono posti in ingresso al neurone di uscita siano del tipo

$$\mathbf{y} = (y_1, y_2, \dots, y_{m-1}, y_m = -1) \quad (25)$$

dimensione dei vettori dello strato intermedio

Naturalmente, se si vuole effettivamente disporre di n valori di ingresso per il MLP, sarà sufficiente considerare un MLP con $n + 1$ neuroni di ingresso. Allo stesso modo, se si vuole disporre effettivamente di m valori in ingresso al neurone di uscita, sarà sufficiente considerare uno strato nascosto composto da $m + 1$ neuroni.

Sotto queste ipotesi, se $\mathbf{w}_j \in \mathbb{R}^n$ è il vettore dei pesi associati agli archi che collegano lo strato di ingresso con il j -esimo neurone dello strato nascosto, con $1 \leq j \leq m - 1$, sarà

$$y_j = g(\mathbf{w}_j \cdot \mathbf{x}) \quad (26)$$

vetture dei pesi che collegano lo strato di ingresso a j -esimo neurone dello strato nascosto
f.ne attivazione
+ output tra vettore primo strato e strato nascosto

In più, se $\mathbf{v} \in \mathbb{R}^m$ è il vettore dei pesi associati agli archi che collegano lo strato nascosto con il neurone di uscita, l'uscita della rete sarà

$$o(\mathbf{w}, \mathbf{x}) = g(\mathbf{v} \cdot \mathbf{y}) \quad (27)$$

vetture dei pesi associati tra strato nascosto e l'uscita.
+ output di rete

dove è stato introdotto il **vettore dei pesi del MLP** $\mathbf{w} \in \mathbb{R}^{m+n \cdot (m-1)}$. Questo vettore dei pesi viene ottenuto concatenando gli m pesi di \mathbf{v} con gli n pesi \mathbf{w}_1 , gli n pesi di \mathbf{w}_2 e procedendo così fino ad arrivare agli n pesi di \mathbf{w}_{m-1} . Questo vettore raccoglie tutti i pesi del MLP e viene usato unicamente per raggruppare in modo ordinato tutti i pesi del MLP.

Quindi, scrivendo per esteso i prodotti scalari,

$$o(\mathbf{w}, \mathbf{x}) = g \left(\sum_{j=1}^m v_j y_j \right) = g \left(\sum_{j=1}^{m-1} v_j g \left(\sum_{i=1}^{n-1} w_{j,i} x_i - w_{j,n} \right) - v_m \right) \quad (28)$$

dove $w_{j,i}$, con $1 \leq j \leq m - 1$ e $1 \leq i \leq n$, è la i -esima componente del vettore \mathbf{w}_j , che non è altro che il vettore che collega lo strato di ingresso con il j -esimo neurone dello strato nascosto. Utilizzando la nomenclatura dei tensori, è possibile dire che $W = (\mathbf{w})_{j=1}^{m-1}$ è il 2-tensore che caratterizza il calcolo svolto dallo strato nascosto del MLP.

voglio esplicitare da cosa dipende o:
da tutti gli x , da tutti i componenti del vettore v ,
da tutti i w_j e di conseguenza da tutti i componenti di
ognuno dei w_j .

matrice composta da
m-1 vettori colonna.
Ogni vettore colonna rappresenta i pesi delle
connessioni tra un neurone dello strato precedente
e tutti i neuroni dello strato nascosto.

Stessa logica di utilizzo: vogliamo approssimare una f.n.e (in questo disegno da \mathbb{R}^3 a \mathbb{R}), per farlo avremo un training set con i punti e i valori della f.n.e, per verificare la nostra bravura avremo un test set.

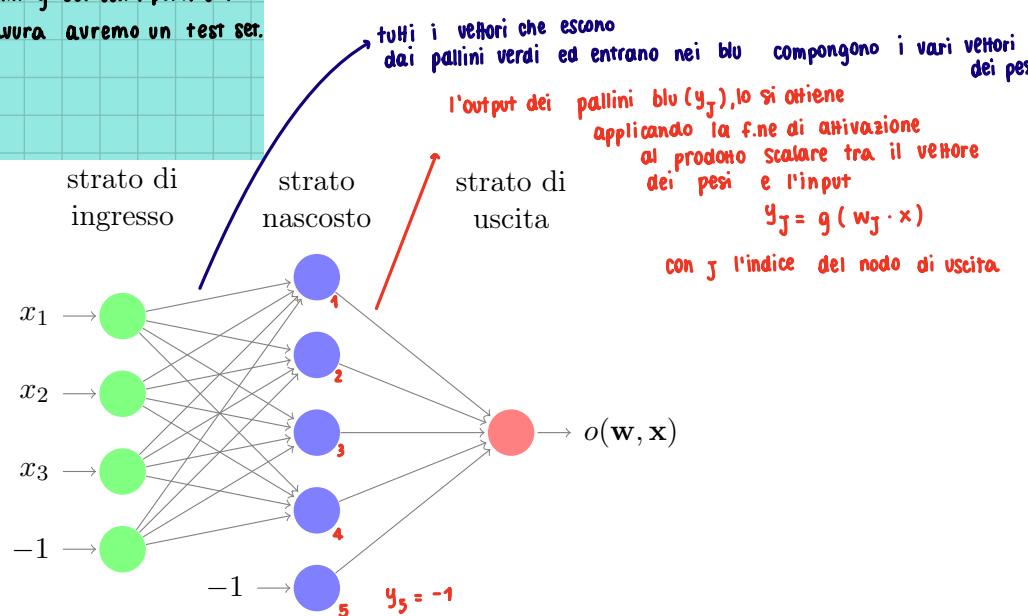


Figura 7: Esempio di un $(3, 4, 1)$ -MLP

Fissato un vettore di ingresso $\mathbf{x} \in \mathbb{R}^n$ del training set, l'errore commesso dal MLP nell'approssimare la funzione f per lo specifico \mathbf{x} vale

$$\epsilon(\mathbf{w}, \mathbf{x}) = f(\mathbf{x}) - o(\mathbf{w}, \mathbf{x}) \quad (29)$$

Introducendo l'errore quadratico e ipotizzando che $g \in C^\infty(\mathbb{R})$, esattamente com'è stato fatto per il SLP,

$$e(\mathbf{w}, \mathbf{x}) = \frac{1}{2} \epsilon^2(\mathbf{w}, \mathbf{x}) \quad (30)$$

si può cercare un vettore dei pesi del MLP utilizzando l'algoritmo di discesa del gradiente nel tentativo di minimizzare l'errore quadratico.

Nell'ipotesi che sia stato fissato un vettore di ingresso $\mathbf{x} \in \mathbb{R}^n$ del training set, allora l'uscita del MLP, l'errore e l'errore quadratico dipendono solo dal vettore dei pesi del MLP.

Si consideri il peso v_j , con $1 \leq j \leq m$, associato all'arco che collega il j -esimo neurone dello strato nascosto con il neurone di uscita.

$$\text{andiamo a calcolare la derivata dell'errore rispetto a } v_j \quad \frac{\partial e}{\partial v_j}(\mathbf{w}) = \frac{1}{2} \cdot 2 \cdot \epsilon(\mathbf{w}) \frac{\partial \epsilon}{\partial v_j}(\mathbf{w}) \quad (31)$$

ma

$$\frac{\partial \epsilon}{\partial v_j}(\mathbf{w}) = -\frac{\partial o}{\partial v_j}(\mathbf{w}) \quad (32)$$

perché $f(\mathbf{x})$ non dipende dal vettore dei pesi del MLP. Ricordando ora la definizione di $o(\mathbf{w})$, si ottiene

$$\frac{\partial o}{\partial v_j}(\mathbf{w}) = g'(\mathbf{v} \cdot \mathbf{y}) \frac{\partial (\mathbf{v} \cdot \mathbf{y})}{\partial v_j} = g'(\mathbf{v} \cdot \mathbf{y}) y_j \quad (33)$$

e quindi, coerentemente con il fatto che, rispetto allo strato nascosto, l'uscita del MLP si comporta come un SLP, è possibile esprimere la seguente regola per l'aggiornamento dei pesi che collegano lo strato nascosto con l'uscita

$$\text{Prima formula di aggiornamento} \quad v_j \leftarrow v_j + \alpha \frac{(f(\mathbf{x}) - g(\mathbf{v} \cdot \mathbf{y}))}{\text{Coef. di apprendimento}} g'(\mathbf{v} \cdot \mathbf{y}) y_j \quad (34)$$

per $1 \leq j \leq m$ e assumendo che l'ultima componente di \mathbf{y} valga -1 . Si noti che questa regola di aggiornamento dei pesi è identica a quella trovata per i SLP.

Ma cosa succede se andiamo a considerare non la variabile di j , ma una delle variabili raggruppate in uno dei vettori dei pesi che abbiamo per collegare l'input con uno dei nodi dello strato nascosto?

Questa regola consente di modificare i pesi che collegano lo strato nascosto con il neurone di uscita in modo da cercare un minimo locale dell'errore commesso dal MLP quando riceve in ingresso il vettore \mathbf{x} . Si noti che prima viene applicato il MLP all'ingresso \mathbf{x} per calcolare il vettore \mathbf{y} e viene applicata in direzione opposta la regola di aggiornamento dei pesi. Questo andamento dell'elaborazione, prima in avanti attraversando il MLP e poi all'indietro, sempre attraversando lo stesso MLP, è caratteristico dell'addestramento supervisionato applicato agli MLP.

Sempre nell'ipotesi che sia stato fissato un vettore di ingresso $\mathbf{x} \in \mathbb{R}^n$, si consideri ora $\mathbf{w}_j \in \mathbb{R}^n$, che è il vettore dei pesi associati agli archi che collegano lo strato di ingresso con il j -esimo neurone dello strato nascosto, e quindi $1 \leq j \leq m - 1$. La derivata dell'errore quadratico rispetto alla i -esima componente di \mathbf{w}_j vale

$$\frac{\partial e}{\partial w_{j,i}}(\mathbf{w}) = \frac{1}{2} \cdot 2 \cdot \epsilon(\mathbf{w}) \frac{\partial \epsilon}{\partial w_{j,i}}(\mathbf{w}) \quad (35)$$

w_j è il vettore dei pesi che collega l'input con quel nodo i . Fissato j abbiamo un vettore, fissato i , al suo interno abbiamo la variabile per w .

Le v sono state messe a posto prima, qua si ottiene una formula di aggiornamento ma, come già discusso, nascondendo che consideriamo nodo dello strato

$$\frac{\partial \epsilon}{\partial w_{j,i}}(\mathbf{w}) = -\frac{\partial o}{\partial w_{j,i}}(\mathbf{w}) \quad (36)$$

Le formule di aggiornamento non sono una sola ma sono tante quante gli strati di cui si calcola l'uscita. perché $f(\mathbf{x})$ non dipende dal vettore dei pesi del MLP. Ora, ricordando la definizione di $o(\mathbf{w})$, si ottiene

$$\frac{\partial o}{\partial w_{j,i}}(\mathbf{w}) = g'(\mathbf{v} \cdot \mathbf{y}) \frac{\partial (\mathbf{v} \cdot \mathbf{y})}{\partial w_{j,i}} \rightarrow \begin{array}{l} \text{da risolvere perché } \mathbf{v} \text{ non dipende} \\ \text{da } \mathbf{w}, \text{ ma } \mathbf{y} \text{ sì} \end{array} \quad (37)$$

y è il valore che si ottiene applicando la regola di McCulloch Pitts per il vettore dei pesi giusto

ma ora, contrariamente a quanto accaduto precedentemente, è necessario considerare la dipendenza del vettore \mathbf{y} dal vettore dei pesi del MLP

$$\frac{\partial (\mathbf{v} \cdot \mathbf{y})}{\partial w_{j,i}} = \mathbf{v} \cdot \frac{\partial \mathbf{y}}{\partial w_{j,i}} = \sum_{k=1}^m v_k \frac{\partial y_k}{\partial w_{j,i}} = \sum_{k=1}^{m-1} v_k \frac{\partial g(\mathbf{w}_k \cdot \mathbf{x})}{\partial w_{j,i}} \quad (38)$$

da cui

$$\frac{\partial (\mathbf{v} \cdot \mathbf{y})}{\partial w_{j,i}} = \sum_{k=1}^{m-1} v_k g'(\mathbf{w}_k \cdot \mathbf{x}) \frac{\partial (\mathbf{w}_k \cdot \mathbf{x})}{\partial w_{j,i}} \quad (39)$$

ma

$$\frac{\partial (\mathbf{w}_k \cdot \mathbf{x})}{\partial w_{j,i}} = \sum_{r=1}^n x_r \frac{\partial w_{k,r}}{\partial w_{j,i}} = \begin{cases} x_i & \text{se } r = i, k = j \\ 0 & \text{altrimenti} \end{cases} \quad (40)$$

e quindi

$$\frac{\partial (\mathbf{v} \cdot \mathbf{y})}{\partial w_{j,i}} = v_j g'(\mathbf{w}_j \cdot \mathbf{x}) x_i \quad (41)$$

Quindi, in sintesi, la regola di aggiornamento dei pesi $w_{j,i}$ associati agli archi che collegano i neuroni di ingresso con il j -esimo neurone dello strato nascosto è

$$\text{Seconda formula di aggiornamento} \quad w_{j,i} \leftarrow w_{j,i} + \alpha (f(\mathbf{x}) - g(\mathbf{v} \cdot \mathbf{y})) g'(\mathbf{v} \cdot \mathbf{y}) v_j g'(\mathbf{w}_j \cdot \mathbf{x}) x_i \quad (42)$$

con $1 \leq j \leq m - 1$ e $1 \leq i \leq n$.

$$\downarrow \frac{\partial o}{\partial w_{j,i}}(\mathbf{w}) \quad \downarrow \frac{\partial (\mathbf{v} \cdot \mathbf{y})}{\partial w_{j,i}}$$

Come si evince dalla regola di aggiornamento dei pesi tra l'ingresso e lo strato nascosto, l'errore compiuto dal MLP viene distribuito tra i pesi in modo simile a come viene distribuito nei pesi associati agli archi che collegano lo strato nascosto con l'uscita del MLP. Infatti, riassumendo quanto ottenuto, le regole di aggiornamento dei pesi sono

$$\mathbf{v} \leftarrow \mathbf{v} + \alpha \epsilon(\mathbf{w}) g'(\mathbf{v} \cdot \mathbf{y}) \mathbf{y} \quad (43)$$

$$\mathbf{w}_j \leftarrow \mathbf{w}_j + \alpha \epsilon(\mathbf{w}) g'(\mathbf{v} \cdot \mathbf{y}) v_j g'(\mathbf{w}_j \cdot \mathbf{x}) \mathbf{x} \quad (44)$$

Le due regole di aggiornamento dei pesi si esprimono così, ma non vanno calcolate in questo modo, perché la seconda formula dipende da v_j e sono quelli che abbiamo usato prima di applicare l'aggiornamento della regola di v .

In realtà quello che c'è da fare è 1) far passare l'input e produrre l'output (calcolo l'ingresso, calcolo del valore sui nodi dello strato intermedio e quindi calcolo gli y , una volta ottenuti gli y calcolo l'uscita [mi serve per calcolare l'errore]) 2) a questo punto, siccome stiamo ragionando per un x nel training set, sappiamo calcolare ϵ ; a questo punto x, y, ϵ li abbiamo, quindi la seconda formula dipende unicamente dal nodo j : cominciamo e facciamo variare i nodi dello strato intermedio uno dopo l'altro con j , da 1 a $m-1$; fissiamo $j=1$ quello che calcoliamo è un aggiornamento di w che dipenderà da x, w attuali, v attuali; il resto lo abbiamo. Facciamo questa operazione e aggiorniamo tutti i w .

Una volta che abbiamo aggiornato i w , possiamo andare ad aggiornare v , perché a questo punto v non dipende più dai w .

e quindi l'errore viene ripartito con coefficiente pari a 1 per l'aggiornamento del vettore dei pesi v e con coefficiente pari a $g'(v \cdot y)v_j$ per l'aggiornamento del vettore dei pesi w_j , con $1 \leq j \leq m-1$. Si noti che le formule di aggiornamento dei pesi di un MLP vengono spesso espresse nell'ipotesi che la funzione di attivazione g sia una funzione logistica in modo da potere sfruttare la proprietà della derivata di questa funzione già discussa nel caso del SLP. Sotto questa ipotesi, valgono le seguenti regole di aggiornamento dei pesi

$$\text{Se la f.n. di attivazione è } s(x) = \frac{1}{1 + e^{-x}} \quad \left\{ \begin{array}{l} v \leftarrow v + \alpha (f(x) - o(w, x)) o(w, x) (1 - o(w, x)) y \xrightarrow{\text{dallo strato nascosto all'uscita}} \quad (45) \\ w_j \leftarrow w_j + \alpha (f(x) - o(w, x)) o(w, x) (1 - o(w, x)) v_j y_j (1 - y_j) x \end{array} \right. \quad (46)$$

e la sua derivata:

$s'(x) = s(x)(1 - s(x))$ Infine, si noti come le regole di addestramento dei pesi ottenute per un MLP con uno strato

i pesi w_j che sono l'ingresso al j -esimo nodo dello strato nascosto

nascosto e una sola uscita abbiano una struttura caratteristica che è comune a tutti gli MLP, qualsiasi sia il numero di strati nascosti e di uscite. Per prima cosa è necessario calcolare i valori delle uscite dei neuroni nei vari strati prima di procedere ad aggiornare i pesi. Infatti, prima si applica il MLP al vettore di ingresso per ottenere l'uscita mediante un processo detto **forward propagation**. Quindi, si valuta l'errore e si ripartisce sui vari strati l'errore commesso, uno strato alla volta, mediante un processo detto **backward propagation**. Per questa sua caratteristica, l'algoritmo di addestramento che prevede di minimizzare l'errore quadratico compiuto dal MLP sui vettori del training set prende il nome di **algoritmo di backpropagation**. L'algoritmo di backpropagation per un $(n-1, m-1, 1)$ -MLP è mostrato nell'Algoritmo 3.

Algoritmo 3 Algoritmo di addestramento supervisionato per un $(n-1, m-1, 1)$ -MLP

```

function MLP _ LEARN( $T, \alpha, \delta, \rho$ )*
    randomize  $v \in \mathbb{R}^m$                                  $\triangleright$  inizializza  $v$ 
    for  $1 \leq j \leq m-1$  do
        randomize  $w_j \in \mathbb{R}^n$                        $\triangleright$  inizializza  $w_j$ 
    end for
    for  $1 \leq r \leq \rho$  do                                 $\triangleright$  ripeti per non più di  $\rho$  epoch
         $\bar{\epsilon} \leftarrow 0$                                  $\triangleright$  inizializza l'errore medio sul training set
        for  $(x, f(x)) \in T$  do                          $\triangleright$  per ogni vettore del training set
             $a \leftarrow v \cdot y$                              $\triangleright$  inizializza  $a \in \mathbb{R}$ 
            for  $1 \leq j \leq m-1$  do  $\triangleright$  per i nodi dello strato nascosto tranne quello di bias
                 $b \leftarrow w_j \cdot x$                          $\triangleright$  inizializza  $b \in \mathbb{R}$ 
                for  $1 \leq i \leq n$  do                       $\triangleright$  aggiorna i pesi usati nello strato nascosto
                     $w_{j,i} \leftarrow w_{j,i} + \alpha (f(x) - g(a)) g'(a) v_j g'(b) x_i$ 
                end for
            end for
            for  $1 \leq i \leq m$  do                       $\triangleright$  aggiorna i pesi usati nello strato di uscita
                 $v_i \leftarrow v_i + \alpha (f(x) - g(a)) g'(a) y_i$ 
            end for
             $\bar{\epsilon} \leftarrow \bar{\epsilon} + \frac{|f(x) - g(v \cdot y)|}{|T|}$        $\triangleright$  aggiorna l'errore medio sul training set
        end for
        if  $\bar{\epsilon} < \delta$  then                                 $\triangleright$  se l'errore medio è sufficientemente piccolo
            return  $w$                                  $\triangleright$   $w$  è il vettore dei pesi dell'MLP cercato
        end if
    end for
    return  $w$                                      $\triangleright$   $w$  è il vettore dei pesi dell'MLP cercato dopo  $\rho$  epoch
end function
```

*

- T : il training set formato da coppie $(x, f(x))$;
- α : il coefficiente di apprendimento;
- δ : la tolleranza accettata sull'errore medio; e
- ρ : il numero massimo di epoch.

Si noti che questo algoritmo ha gli stessi parametri dell'algoritmo di addestramento supervisionato trovato per i SLP. Infatti, l'algoritmo per i SLP non è altro che un caso particolare dell'algoritmo di backpropagation.

- ✓ **Esempio 2.** Un uso tradizionale degli MLP e dell'algoritmo di backpropagation è nel riconoscimento del testo manoscritto. Un esempio ormai ritenuto classico è l'utilizzo di un $(784, 800, 10)$ -MLP per riconoscere le cifre manoscritte opportunamente isolate in piccole immagini formate da 28×28 pixel. Queste immagini usano una scala di 256 grigi, come mostrato in Figura 2, e quindi possono essere direttamente elaborate dal MLP senza necessità di pre-elaborazione.

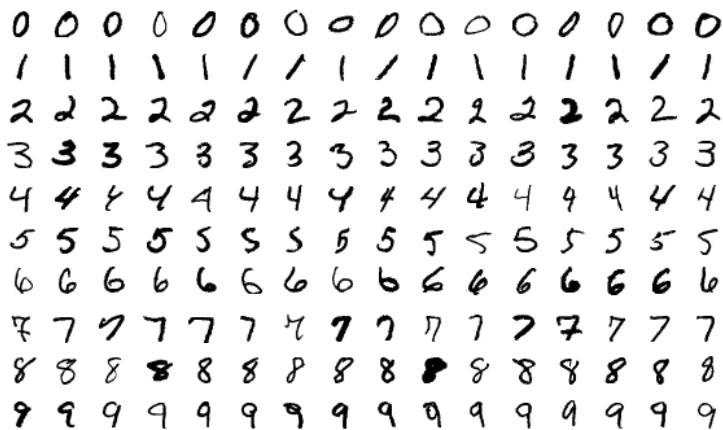


Figura 8: Esempi di cifre manoscritte riconosciute da un $(784, 800, 100)$ -MLP

Il risultato dell'elaborazione del MLP è un vettore di 10 numeri reali in cui la posizione di una delle componenti massime rappresenta la cifra riconosciuta. Quindi, ad esempio, se il risultato del MLP è un vettore in cui la componente massima è in posizione 3, allora è possibile dire che il MLP ha classificato l'immagine in ingresso come una cifra 3. Si noti che il risultato della classificazione del MLP non è univocamente definito se il vettore risultato ha più componenti massime.

L'addestramento del MLP utilizzato per questo compito di riconoscimento di cifre manoscritte avviene usando l'algoritmo di backpropagation con un training set formato da 60 000 immagini pre-classificate. La misura delle prestazioni del MLP ottenuto mediante l'algoritmo di backpropagation avviene con un test set formato da 10 000 immagini pre-classificate. Il risultato documentato in letteratura parla di un MLP con un errore di classificazione sul test set inferiore al 2%, coerente con il grafico mostrato in Figura 9.

Il seguente **teorema di approssimazione universale** consente di usare gli MLP come strumenti generali per l'approssimazione di funzioni continue. Si noti comunque che esiste una variante del teorema di approssimazione universale che esplicita come sia possibile utilizzare un MLP per approssimare con tolleranza piccola a piacere anche funzioni con un numero finito di discontinuità nel dominio considerato. Questo teorema non viene però discusso perché richiede una notazione più complessa di quella utilizzata in questo testo.

In sintesi, il seguente teorema di approssimazione universale può essere letto come segue. Fissata una certa tolleranza $\epsilon \in \mathbb{R}_+$ nell'approssimazione di una funzione f continua su un compatto, è possibile trovare un MLP che possa essere utilizzato come approssimatore della

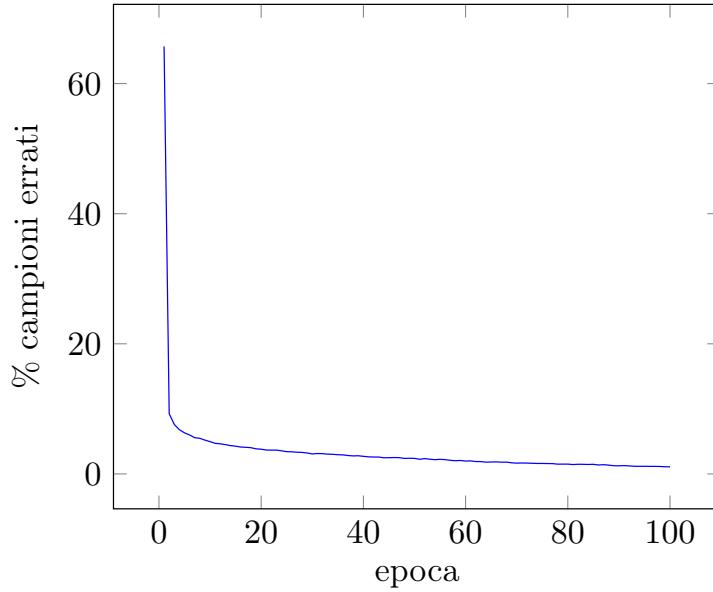


Figura 9: Andamento della percentuale di campioni errati nel training set in 100 epoche usando un $(784, 800, 100)$ -MLP

funzione f con la tolleranza richiesta. La struttura del MLP che è possibile utilizzare per l'approssimazione viene fissata dal teorema anche se, comunque, il teorema lascia notevoli gradi di libertà nella scelta dello specifico MLP.

Teorema 1 (Di Approssimazione Universale). Dato $n \in \mathbb{N}_+$, sia $D \subset \mathbb{R}^n$ compatto e sia $f : S \rightarrow \mathbb{R}$ una funzione reale di più variabili reali continua nell'aperto $S \subseteq \mathbb{R}^n$ con $D \subset S$. Sia $g : \mathbb{R} \rightarrow \mathbb{R}$ una funzione continua, monotona strettamente crescente e limitata. Per ogni $\epsilon \in \mathbb{R}_+$ esistono

- $m \in \mathbb{N}_+$
- v_j, θ_j , con $1 \leq j \leq m$
- $w_{j,i} \in \mathbb{R}$, con $1 \leq i \leq n$ e $1 \leq j \leq m$

tali che la funzione

$$h(\mathbf{x}) = \sum_{j=1}^m v_j g \left(\sum_{i=1}^n w_{j,i} x_i - \theta_j \right)$$

↓
approssimazione

J neuroni di McCulloch Pitts

bias

$$\max_{\mathbf{x} \in D} |f(\mathbf{x}) - h(\mathbf{x})| < \epsilon$$

soddisfi

In modo meno formale: se noi consideriamo una funzione continua in un compatto chiuso e limitato e consideriamo un MLP con 1 strato nascosto, i pesi del MLP li troviamo una volta che abbiamo fissato una tolleranza che ci interessa.
Fissiamo una tolleranza ϵ , questo teorema ci garantisce che esistono i pesi di un MLP che ci permette di approssimare, con una tolleranza ϵ , la funzione f su tutto il compatto D che abbiamo considerato.
Quindi MLP non è qualiasi: n ingressi, uno strato nascosto di cui non conosciamo il n. di neuroni, i neuroni dello strato nascosto usano una funzione g abbastanza generica, l'unico strato di uscita ha un unico neurone che ha una funzione di attivazione identità

(48)

Si noti che non è fissata la funzione di attivazione utilizzata per i neuroni dello strato nascosto. Viceversa, si assume che il MLP utilizzi una funzione di attivazione identità per il calcolo dell'uscita. Quest'ultima circostanza è del tutto ragionevole perché la scelta di una funzione di attivazione limitata avrebbe anche limitato a priori l'uscita del MLP e quindi avrebbe reso il MLP meno efficace nell'approssimare f , visto che non è stata posta una limitazione a priori ai valori di f .

"Per quello che abbiamo visto fino ad ora, questo argomento sembrerà marginale" cit.

4 DNN e Approssimazione di Funzioni

Nonostante il teorema di approssimazione universale garantisca che un MLP con uno strato nascosto sia sufficiente per approssimare bene funzioni continue, come verrà discusso nel seguito, è spesso utile avere a disposizione MLP con molti strati nascosti. Conventionalmente, un MLP con più di due strati nascosti viene chiamato **rete neurale profonda** (**Deep Neural Network, DNN**). Normalmente, il numero di strati nascosti delle DNN usate nella pratica supera abbondantemente la decina.

Una DNN non si differenzia in modo sostanziale da un MLP con uno strato nascosto e l'unico problema da affrontare è come estendere l'algoritmo di backpropagation al caso di più di uno strato nascosto. Fortunatamente, l'algoritmo di backpropagation può essere facilmente esteso a più di uno strato nascosto perché il calcolo del relativo gradiente può essere facilmente generalizzato. Non verrà affrontato il calcolo del gradiente nel dettaglio ma, sfruttando la notazione vista per gli MLP con uno strato nascosto, non è difficile arrivare a dimostrare i risultati che qui verranno solo presentati.

Si consideri un MLP con $h \in \mathbb{N}_+$ strati nascosti ognuno dei quali è caratterizzato da un numero specifico di neuroni. Si ipotizzi inoltre che il MLP abbia il massimo numero di archi compatibile con la sua struttura a strati e che tutti i neuroni utilizzino la stessa funzione di attivazione $g : \mathbb{R} \rightarrow \mathbb{R}$. Se $(m_i)_{i=1}^h$ è il vettore di \mathbb{N}_+^h che raccoglie il numero di neuroni di ogni strato nascosto, comprensivi di neuroni fissati a -1 per gestire i pesi di bias, allora la rete è completamente definita a meno del numero di valori di input e di output. Una rete di questo tipo viene normalmente chiamata $(n-1, m_1-1, m_2-1, \dots, m_h-1, m_{h+1})$ -MLP, dove $n \in \mathbb{N}_+$ è il numero di neuroni nello strato di input e $m_{h+1} \in \mathbb{N}_+$ è il numero di neuroni nello strato di output. Si noti che, per semplificare la trattazione, normalmente si definiscono $m_0 = n$ e $m_{h+1} = 1$. In più, una volta che è stata adottata questa notazione, si parla di k -esimo strato del MLP, con $0 \leq k \leq h+1$, senza specificare se lo strato è di input, di output o nascosto. In Figura 10 viene mostrata una DNN del tipo $(3, 4, 2, 4, 1)$ -MLP e, quindi, caratterizzata da tre strati nascosti.

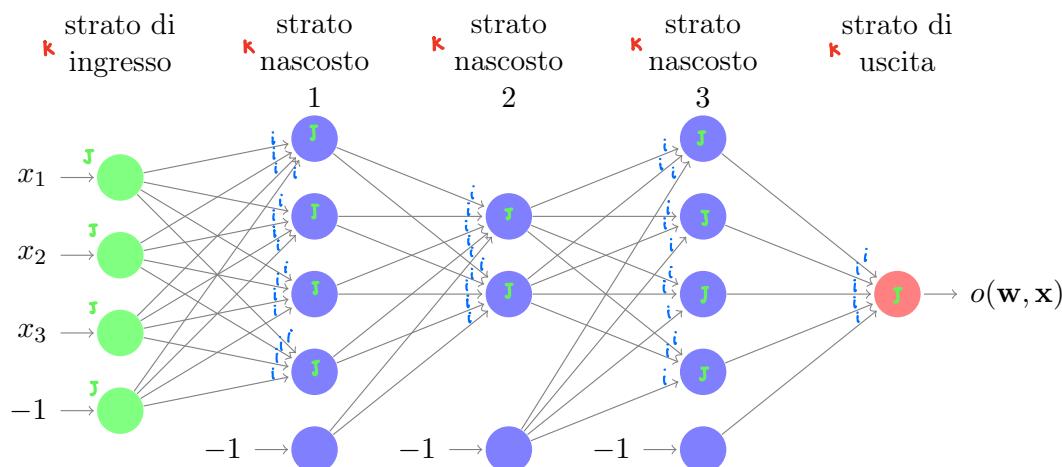


Figura 10: Esempio di DNN del tipo $(3, 4, 2, 4, 1)$ -MLP

→ con questa dicitura vanno considerati il num. max di archi tra i vari strati

Si consideri un $(n-1, m_1-1, m_2-1, \dots, m_h-1, 1)$ -MLP caratterizzato da una funzione di attivazione $g : \mathbb{R} \rightarrow \mathbb{R}$ utilizzata in tutte le unità di McCulloch-Pitts. L'obiettivo è usare il MLP come un approssimatore di una funzione $f : \mathbb{R}^n \rightarrow \mathbb{R}$ di cui si conoscano i valori per alcuni vettori. Come nel caso del MLP con uno strato nascosto, alcuni dei vettori in cui la funzione è nota vengono raccolti in un training set, mentre i rimanenti vengono raccolti

Nel momento in cui andiamo a guardare il teorema di approssimazione universale, ci rendiamo conto che ci basta una rete con uno strato nascosto per poter approssimare una f.n. continua; esiste una variante del teorema che ci permette di dire che, se la f.n. non è continua sul dominio compatto, allora ci bastano 2 strati nascosti; se con 2 strati nascosti riusciamo ad approssimare bene una qualsiasi f.n., anche non continua sul dominio, fine...

"introduciamo cose superflue che forse però ci potrebbero servire" cit.

1. Preattivazione: $y = g(x \cdot w)$ la preattivazione è lo scalare risultato da $w \cdot x$ prima di aver applicato la f.n.e di attivazione (lui lo segna spesso con a)
2. o_k è l'output dello strato k , anziché dire y_k
3. Numerari gli strati è comodo Strato 0 = input, strato 1 = primo strato nascosto, ..., strato $h+1$ = livello uscita

in un test set. Come di consueto, per semplificare la notazione, senza comunque perdere di generalità, si ipotizza che tutti i vettori del training set, del test set e i vettori posti in ingresso agli strati nascosti abbiano l'ultima componente fissata a -1 .

In queste ipotesi è possibile raggruppare i vettori dei pesi della rete in un **3-tensore \mathbf{W}** le cui componenti valgono

$$w_{k,j,i} \in \mathbb{R} \quad (49)$$

dove

essendo i pesi entranti nello strato, non entrano in

$k=0$

1. L'indice $1 \leq k \leq h + 1$ permette di fare riferimento ai pesi in ingresso al k -esimo strato; = quale strato stiamo guardando
2. L'indice $1 \leq j \leq m_k - 1$ permette di fare riferimento agli $m_k - 1$ vettori dei pesi in ingresso al k -esimo strato; e = quale neurone dello strato
3. L'indice $1 \leq i \leq m_{k-1}$ permette di fare riferimento agli m_{k-1} pesi che formano le componenti di uno degli $m_k - 1$ vettori dei pesi in ingresso al k -esimo strato. = peso che stiamo considerando di tutti i pesi che ci sono nel vettore che collega lo strato precedente col neurone che stiamo considerando

Utilizzando il 3-tensore \mathbf{W} ed introducendo il vettore dei pesi del MLP \mathbf{w} come fatto nel caso dei MLP con uno strato nascosto si possono esprimere le parti che concorrono alla regola di aggiornamento dei pesi che minimizza l'errore quadratico

$$e(\mathbf{w}, \mathbf{x}) = \frac{1}{2}(f(\mathbf{x}) - o(\mathbf{w}, \mathbf{x}))^2 \quad (50)$$

compiuto nell'approssimare il valore della funzione f per uno dei vettori del training set \mathbf{x} con il valore $o(\mathbf{w}, \mathbf{x})$. Si noti che la dipendenza di o e di altre quantità da \mathbf{w} e \mathbf{x} non verrà più esplicitata per semplificare ulteriormente la notazione adottata.

Per esprimere in modo succinto la regola di aggiornamento dei pesi per l'addestramento supervisionato di una DNN, si introducono i vettori $\mathbf{o}_k \in \mathbb{R}^{m_k}$, con $0 \leq k \leq h + 1$, che raccolgono i valori delle uscite degli m_k neuroni del k -esimo strato. Questi valori delle uscite permettono di definire la **preattivazione** del neurone $1 \leq j \leq m_k - 1$ dello strato $1 \leq k \leq h + 1$ come

$$a_{k,j} = \mathbf{w}_{k,j} \cdot \mathbf{o}_{k-1} \quad (51)$$

parte lineare che entrerà nella f.n.e vetore dei pesi entranti nel nodo j
9 (PREATIVAZIONE) uscita del nodo precedente

Si noti che questo valore non è altro che l'argomento della funzione di attivazione del j -esimo neurone del k -esimo strato e, quindi, è un valore che viene comunque calcolato quando si calcola l'uscita della rete.

Fissato un coefficiente di apprendimento $\alpha \in \mathbb{R}_+$, è possibile esprimere la regola di aggiornamento dei pesi per un fissato campione del training set $(\mathbf{x}, f(\mathbf{x}))$ come segue

$$w_{k,j,i} \leftarrow w_{k,j,i} + \alpha \delta_{k,j} o_{k-1,i} \quad (52)$$

simile a quello che abbiamo visto fino ad ora.
dipende dal neurone j dello strato k

dove il termine $\delta_{k,j}$ viene definito in modo induttivo partendo da $\delta_{h+1,1}$ come segue

questo non lo si ottiene con facilità
e

$$\delta_{h+1,1} = g'(\mathbf{w}_{h+1,1} \cdot \mathbf{o}_h)(f(\mathbf{x}) - o_{h+1,1}) = g'(a_{h+1,1})(f(\mathbf{x}) - o_{h+1,1}) \quad (53)$$

strato di unico neurone presente preattivazione del primo e ultimo strato

$$\delta_{k,j} = g'(\mathbf{w}_{k,j} \cdot \mathbf{o}_{k-1}) \sum_{s=1}^{m_{k+1}} w_{k+1,s,j} \delta_{k+1,s} = g'(a_{k,j}) \sum_{s=1}^{m_{k+1}} w_{k+1,s,j} \delta_{k+1,s} \quad (54)$$

Quindi, l'aggiornamento dei pesi di una DNN durante l'addestramento supervisionato una volta che sia stato fissato un campione $(\mathbf{x}, f(\mathbf{x}))$ può essere svolto utilizzando una fase di forward propagation seguita da una fase di backward propagation in modo simile a quanto

struttura simile di MLP con un singolo strato nascosto

Prima formula di aggiornamento

$$v_j \leftarrow v_j + \alpha (f(\mathbf{x}) - g(v \cdot y)) g'(v \cdot y) y_i$$

coeff. di apprendimento output di rete 34 deriva. dell'output di rete

* si calcola come la g' applicata alla preattivazione dello strato k , neurone j , poi viene fuori una combinazione lineare che qua non c'è perché qui c'è un solo neurone d'uscita, e quindi rimane un termine solo; se abbiamo più neuroni d'uscita, perché ogni strato a più neuroni nello strato successivo dobbiamo cercare di capire cosa succede nello strato successivo, e se svolgiamo il calcolo, cosa viene fuori? Una somma che va da 1 ai n di neuroni dello strato successivo per il peso che peschiamo dallo strato successivo al varia re di s per j fissato per s dello strato succ. per s fissato.

fatto per un MLP con un solo strato nascosto. In questo caso, però, conviene memorizzare i valori calcolati nella fase di forward propagation, valori di output e preattivazioni, per usarli poi nella fase di backward propagation.

In particolare, l'algoritmo di addestramento supervisionato può essere riassunto secondo le cinque parti principali descritte nel seguito:

1. Si propaga il vettore di input \mathbf{x} attraverso la rete andando a calcolare i vettori \mathbf{o}_k in uscita dagli strati intermedi e dallo strato di output e, quindi, con $1 \leq k \leq h + 1$;
2. Durante la propagazione del vettore di input \mathbf{x} attraverso la rete, si calcolano le preattivazioni;
3. Si calcola il valore $\delta_{h,1}$ usando l'errore effettivamente compiuto $f(\mathbf{x}) - o_{h+1,1}$ e la preattivazione $a_{h+1,1}$ dell'unico neurone di uscita;
4. Si calcola il valore $\delta_{k,j}$ sfruttando la preattivazione del k -esimo strato e i valori $\delta_{k+1,s}$ dello strato successivo, con $1 \leq s \leq m_{k+1}$; e
5. Per il k -esimo strato, partendo dall'uscita e tornando verso l'ingresso, si aggiornano i pesi applicando la regola di aggiornamento dei pesi.

Si noti che l'algoritmo di backpropagation visto per gli MLP con un solo strato nascosto è un caso particolare dell'algoritmo presentato e, quindi, si riserva il termine di algoritmo di backpropagation proprio a questa versione più generale. Infatti, anche questa versione più generale è strutturata in una fase di forward propagation, descritta dai punti 1 e 2 precedenti, e da una fase di backward propagation descritta dai punti 3, 4 e 5 precedenti. In più, si noti che le formule viste nel caso di un solo strato nascosto non sono altro che casi particolari delle formule presentate in questo caso più generale.

5 CNN e Approssimazione di Funzioni

Uno dei problemi principali che è necessario affrontare per utilizzare le reti neurali per risolvere problemi particolarmente complessi è quello di ridurre i tempi di addestramento. Infatti, la riduzione dei tempi di addestramento è l'unico modo perseguitibile per aumentare il numero di campioni utilizzati durante l'addestramento e quindi potenzialmente aumentare le capacità delle reti di affrontare problemi complessi.

Da questo punto di vista, le DNN sembrano particolarmente svantaggiate perché ogni aumento del numero degli strati implica un corrispettivo aumento del numero di pesi da addestrare e, quindi, un corrispettivo aumento dei tempi di addestramento. In più, il teorema di approssimazione universale dice che un singolo strato nascosto è sufficiente per approssimare bene quanto si vuole una data funzione continua. Quindi, l'utilizzo di DNN sembra ulteriormente penalizzato perché, pur aumentando i tempi di addestramento, non sono in grado di fornire migliori capacità di approssimazione.

Per rendere le DNN utili nella risoluzione di problemi complessi, si preferisce quindi utilizzarle in una configurazione che permetta di ridurre il numero dei pesi da addestrare consentendo comunque di approssimare bene quanto si vuole una data funzione. In particolare, è possibile decidere di fissare il valore di alcuni pesi a zero utilizzando il numero di strati nascosti superiori a due per rendere comunque in grado la rete di approssimare bene la funzione su cui si sta lavorando. Quindi, utilizzando come gradi di libertà il numero degli strati nascosti e il numero di neuroni per ogni strato, è possibile fissare molti pesi a zero in modo da escluderli dall'addestramento con l'effetto quindi di ridurre i tempi di addestramento in modo significativo.

Un approccio tipico per decidere quali pesi fissare a zero è notare che spesso i problemi affrontati mediante le reti neurali sono contraddistinti da **feature** (o **caratteristiche**) locali. Quindi, il valore di output dipende principalmente da quello che succede in un **intorno** dei singoli neuroni di input e dipende poco da quello che succede lontano ad ogni singolo neurone di input. Se è possibile ipotizzare che l'output dipenda unicamente da feature locali dell'input, allora è possibile fissare a zero i pesi che collegano l'ingresso di un neurone con i neuroni ad esso lontani. Se una DNN viene strutturata mediante l'estrazione di feature locali dell'input, allora ogni livello della rete ha il compito di aggregare le feature estratte dal livello precedente fino a produrre l'output. La Figura 11 mostra un esempio di DNN in cui ogni neurone di uno strato è collegato mediante pesi non nulli a solo pochi neuroni dello strato successivo. In questo caso, il numero di pesi da trovare mediante addestramento si è ridotto a 37.

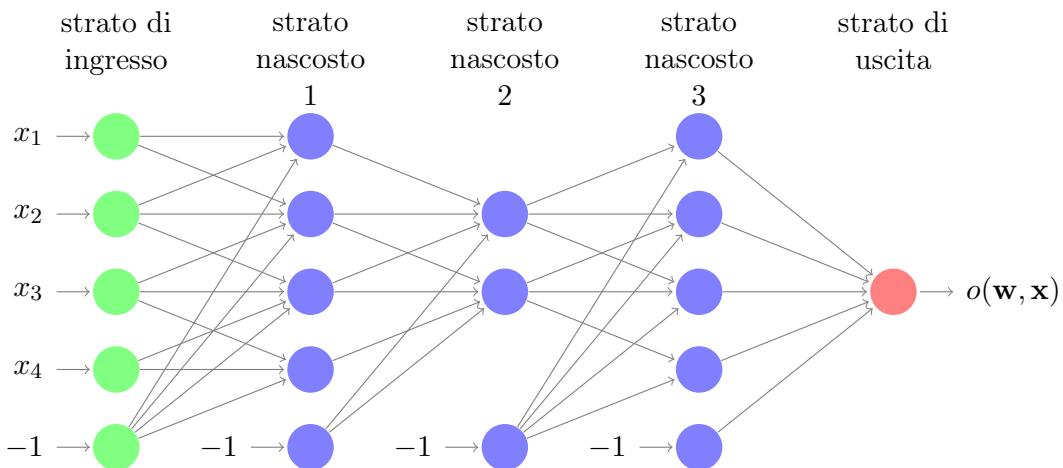


Figura 11: Esempio di DNN che lavora su feature locali

Questo approccio alla progettazione di DNN può essere spinto all'estremo ipotizzando che le elaborazioni svolte in ogni intorno siano sempre le stesse. Quindi, ipotizzando che i pesi che associano i neuroni di un intorno per produrre l'input dello strato successivo siano sempre gli stessi, indipendentemente dall'intorno considerato. Se una rete può essere strutturata in questo modo, allora il tempo di addestramento viene ulteriormente abbattuto perché l'addestramento ha come unico scopo quello di trovare gli unici pesi applicati a tutti gli intorni. Una rete neurale di questo tipo viene detta **convolutiva** (o **convoluzionale** da **Convolutional Neural Network, CNN**) e l'unico vettore dei pesi che viene cercato per ogni livello viene detto **nucleo** (o **kernel**) di **convoluzione**. Se si ipotizza che la rete in Figura 11 sia una CNN, allora ogni strato è caratterizzato unicamente da un nucleo di convoluzione formato da pochi pesi, che sono i pesi che collegano uno strato con il successivo più il peso di bias. Naturalmente, strati diversi possono avere nuclei di convoluzione diversi e non è richiesto che tutti gli strati lavorino in modo convoluzionale. Infatti, normalmente una CNN ha almeno uno strato in cui il numero di archi entranti è massimo, come succede nello strato di uscita della rete in Figura 11. Facendo riferimento alla DNN mostrata in Figura 11, se la rete è in realtà una CNN, il numero di pesi da trovare mediante addestramento si è ridotto a 16.

è un componente che svolge il compito di assegnare un etichetta a un'istanza di input.

l'insieme dei numeri interi tra 1 e m, inclusi

6 Reti Neurali e Classificatori

Per $n \in \mathbb{N}_+$, il problema di classificazione dei vettori di $S \subseteq \mathbb{R}^n$ è il problema che prevede di associare ogni vettore di S ad una classe scelta tra un numero finito e noto di classi tra loro disgiunte. Se $m \in \mathbb{N}_+$ è il numero di classi tra cui scegliere, un classificatore non è altro che una funzione f da S in $[1..m] \subset \mathbb{N}$. Siccome il problema di classificazione può essere ricondotto al problema di approssimare una funzione, è sicuramente possibile utilizzare un MLP come classificatore. Normalmente, si utilizza un MLP con n ingressi e m uscite addestrandolo in modo supervisionato mediante dei campioni che associano vettori di S a dei vettori del tipo

$$(0, 0, \dots, 1, \dots, 0) \quad (55)$$

dove il valore 1 viene messo nella posizione corrispondente alla classe del campione che si sta studiando. Quindi, ad esempio, se si sta usando un MLP per classificare vettori in 4 classi, allora se un campione è di classe 2 il relativo valore della funzione da approssimare viene fissato a $(0, 1, 0, 0)$.

Siccome l'addestramento della rete non riuscirà a garantire che i vettori calcolati abbiano solo componenti con valori pari a 0 o 1, è necessario che i vettori calcolati dalla rete vengano ulteriormente elaborati per stabilire quale sia la classe che la rete ha attribuito al vettore di input. Normalmente, si utilizza una tra due possibili metodi per stabilire la classe del vettore di input una volta noto il vettore di output. Il primo metodo prevede di scegliere in modo arbitrario un indice corrispondente ad una delle componenti massime del vettore calcolato dalla rete e, quindi, la classe stimata viene calcolata come segue

Avendo un vettore di valori, esempio \mathbf{o}_i (che sono le uscite della rete per il nodo $1, 2, \dots, m$), max da solo dice quanto vale il massimo, argmax dice la posizione del massimo

$$k = \underset{1 \leq i \leq m}{\operatorname{argmax}} o_i \quad (56)$$

vettore con m componenti

se il vettore di uscita della rete è $\mathbf{o} = (o_i)_{i=1}^m$. In alternativa, dato \mathbf{o} si può calcolare un nuovo vettore \mathbf{s} mediante la seguente funzione

Prendere il vettore di uscita \mathbf{o} e farlo passare attraverso la f. ne softmax, che ritorna un vettore che va da 1 a m, le cui componenti sono il risultato di questo calcolo.

$$\mathbf{s} = \text{softmax}(\mathbf{o}) = \left(\frac{e^{o_i}}{\sum_{j=1}^m e^{o_j}} \right)_{i=1}^m$$

tutte le componenti tra 0 e 1, indipendentemente dagli \mathbf{o} , che possono anche essere negative o grandi, e la somma di tutte le componenti di \mathbf{s} da 1, questo per come sono fatti gli esponenziali

Il vettore \mathbf{s} ha tutte le componenti comprese tra 0 e 1 e la somma di tutte le componenti risulta 1. Quindi, le componenti di \mathbf{s} possono essere interpretate come delle probabilità ed è possibile stimare la classe del vettore di ingresso estraendo un numero tra 1 e m ed assegnando ad ogni valore possibile $a \in [1..m]$ la probabilità s_a . In questo modo, classi a cui la rete ha associato valori alti sono comunque preferite pur non fissando un criterio rigido nella scelta tra classi con valori tra loro simili.

Si noti che, com'è ragionevole attendersi, la capacità di un MLP di approssimare bene un classificatore dipende molto dal tipo di MLP che si sta considerando. Ad esempio, un $(n, 2)$ -SLP può essere usato per classificare vettori di \mathbb{R}^n in due classi facendolo seguire da un'elaborazione basata su argmax. In questo caso, la rete può solo comportarsi come un classificatore lineare e, quindi, la classificazione sarà totalmente corretta solo se esiste un iperpiano di \mathbb{R}^n in grado di separare i campioni appartenenti alle due classi. In alternativa, l'utilizzo di un MLP con almeno uno strato nascosto, oppure di una DNN, sarà in grado di produrre classificatori più raffinati e in grado di separare le due classi anche in casi complessi, come esemplificato in Figura 12. Si noti che la classificazione in due classi può essere affrontata anche mediante un MLP con un singolo neurone nello strato di output, come mostrato in Figura 12, senza comunque aumentare in alcun modo le capacità di classificazione della rete scelta.

1° metodo
Tipo di classificazione
HARD MAX
↓
RIGIDO

2° metodo

applico argmax
su \mathbf{s}
 $k = \operatorname{argmax}_{1 \leq i \leq m} (s_i)$

* Ricordo la definizione di IPERPANO:

Dato il vettore reale $\mathbf{x} \in \mathbb{R}^n$ e il versore $\mathbf{n} \in \mathbb{R}^n$, l'iperpiano che contiene \mathbf{x} e ha \mathbf{n} come (versore) normale è

$$\mathcal{H}_n(\mathbf{x}) = \{ \mathbf{v} \in \mathbb{R}^n : (\mathbf{v} - \mathbf{x}) \cdot \mathbf{n} = 0 \}$$

assicura che i vettori partano dalla fine di \mathbf{x}

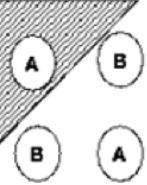
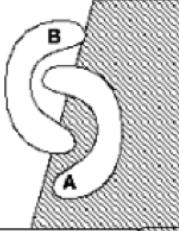
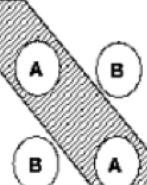
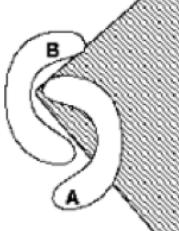
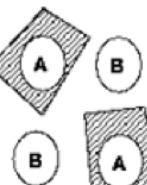
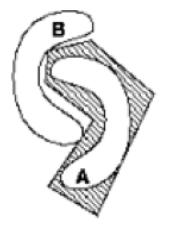
Structure	Regions	XOR	Meshed regions
single layer	Half plane bounded by hyperplane		
two layer	Convex open or closed regions		
three layer	Arbitrary (limited by # of nodes)		

Figura 12: Esempi di reti neurali usate come classificatori per 2 classi

Si noti che aumentare l'accuratezza con cui la rete classifica bene i vettori del training set ha spesso l'effetto di impedire alla rete di **generalizzare**. Quindi, ad esempio, nella Figura 12 l'utilizzo di due strati nascosti permette di migliorare le prestazioni della rete nel classificare i vettori della classe A ma rende la rete troppo permissiva nel classificare i vettori della classe B. Per questo, le prestazioni di un classificatore, indipendentemente dalla tecnica utilizzata per realizzarlo, vengono sempre espresse mediante la relativa **matrice di confusione** definita come $K = (k_{a,n})_{a,n=1}^m$, dove $k_{a,n}$ è la probabilità (condizionata) che il classificatore produca come stima della classe il valore p (da **predicted**) quando la classe effettiva è a (da **actual**). Naturalmente, un classificatore che non commette mai errori ha come matrice di confusione la matrice identità e, quindi, il comportamento di un classificatore è considerato tanto migliore quanto la sua matrice di confusione si avvicina alla matrice identità. Si noti, infine, che alle volte vengono introdotti degli **indici statistici** che riassumono le caratteristiche della matrice di confusione sotto opportune ipotesi.

7 Reti Neurali Ricorrenti e di Hopfield

Se una rete neurale non può essere descritta mediante una struttura a livelli, allora prende il nome di rete neurale **ricorrente** perché, almeno per un neurone, l'input dipende, eventualmente in modo indiretto, dal proprio output. Una rete neurale ricorrente è quindi un grafo orientato pesato i cui nodi sono neuroni di McCulloch-Pitts e, per ogni neurone, i pesi sugli archi sono i pesi associati al neurone. In generale, per una rete ricorrente non è più possibile identificare quale sia l'input e quale sia l'output, anche se è sempre possibile identificare quale sia l'input e quale sia l'output di un singolo neurone perché il grafo è

orientato. Anche in questo caso, per ogni neurone della rete, viene fissato uno dei valori di input ad un valore costante che viene detto valore di bias e normalmente vale -1 . Se non viene esplicitamente indicato il contrario, una rete neurale ricorrente si intende descritta da un grafo completamente connesso privo di autoanelli.

Una rete (neurale) di Hopfield è una rete neurale ricorrente in cui la funzione di attivazione dei neuroni è la funzione **signum**

$$\text{sgn}(x) = \begin{cases} +1 & \text{se } x > 0 \\ 0 & \text{se } x = 0 \\ -1 & \text{se } x < 0 \end{cases} \quad (58)$$

Visto che non è stato detto altrimenti, una rete di Hopfield è descritta da un grafo completamente connesso privo di autoanelli. →(non esiste un arco che collega un neurone direttamente con se stesso)

L'ingresso della rete viene fornito fissando i valori di uscita dei neuroni a valori iniziali nell'insieme $\{-1, +1\}$. L'uscita della rete viene ottenuta leggendo i valori delle uscite dei neuroni dopo un tempo di elaborazione sufficientemente lungo, nell'ipotesi che le uscite si siano stabilizzate a valori asintotici, se esistono. Si noti che, di norma, il valore 0 non viene usato come ingresso e si ipotizza che venga prodotto come uscita solo durante le fasi transitorie in cui l'uscita non è ancora considerata stabile. Quindi, una rete di Hopfield legge un ingresso binario e produce un'uscita binaria, sempre nell'insieme $\{-1, +1\}$.

Una rete di Hopfield è quindi un **sistema dinamico non lineare** il cui comportamento può essere studiato in modo analitico sotto opportune ipotesi. Normalmente, però, una rete di Hopfield viene **simulata a tempo discreto** aggiornando l'uscita di un neurone scelto casualmente per ogni istante di simulazione. Quindi, data una rete di Hopfield, si fissano inizialmente i valori delle uscite dei neuroni e poi, ciclicamente, si sceglie un neurone casualmente e si aggiorna la sua uscita sulla base dei relativi ingressi, fino al raggiungimento di un numero sufficiente di iterazioni o fino alla stabilizzazione dei valori di uscita dei neuroni.

Data una rete di Hopfield con $n \in \mathbb{N}_+$ neuroni, fissato un qualsiasi istante di simulazione, il vettore $\mathbf{s} \in \{-1, 0, +1\}^n$ formato delle uscite dei neuroni prima dell'aggiornamento delle uscite stesse viene detto **stato della rete**. Lo stato della rete evolve dinamicamente partendo da uno stato iniziale corrisponde al vettore di ingresso della rete e giungendo ad uno stato finale corrisponde al vettore di uscita della rete. Questo tipo di simulazione del comportamento di una rete di Hopfield viene detto **simulazione asincrona** e, contrariamente alla **simulazione sincrona** che non verrà discussa qui, descrive bene la rete nei termini di un sistema dinamico.

Si consideri una rete di Hopfield formata da $n \in \mathbb{N}_+$ neuroni, ognuno dei quali è associato ad un vettore dei pesi e ad un valore di bias che vengono entrambi usati per produrre un valore di uscita. I neuroni della rete vengono normalmente numerati da 1 a n e, quindi, l' i -esimo neurone, con $1 \leq i \leq n$, viene descritto da un vettore dei pesi $\mathbf{w}_i \in \mathbb{R}^n$ e da un valore di bias $b_i \in \mathbb{R}$, dove si assume che $w_{i,i} = 0$ perché nella rete non sono presenti autoanelli. Se $\mathbf{s} \in \mathbb{R}^n$ è lo stato attuale della rete, allora l'uscita dell' i -esimo neurone vale

$$o_i = \text{sgn}(\mathbf{w}_i \cdot \mathbf{s} - b_i) = \text{sgn} \left(\sum_{j=1}^n w_{i,j} s_j - b_i \right) \quad (59)$$

↑ n. neuroni
↑ f. ne attiv.
↑ signum
↑ stato attuale
↑ pesi
↑ peso bias

Si noti che, per $1 \leq i \leq n$, $o_i \in \{-1, 0, +1\}$. In più, si noti che l'uscita dell' i -esimo neurone si dice **stabile**, o **stazionaria**, se o_i non varia anche dopo l'aggiornamento causato da una variazione dello stato della rete.

l'output del neurone (o_i) è uguale all'input (s). Se succede che in una rete, da un certo momento, tutti gli o_i sono uguali agli s , allora la rete è stabile.

* Matrice simmetrica: matrice quadrata in cui ogni elemento della diag. principale è uguale al suo simmetrico rispetto alla diagonale principale.

Se A è una matrice $n \times n$, allora | Ricordo che

$$A[i,j] = A[j,i]$$

\uparrow
trasposta: matrice con righe e colonne invertite

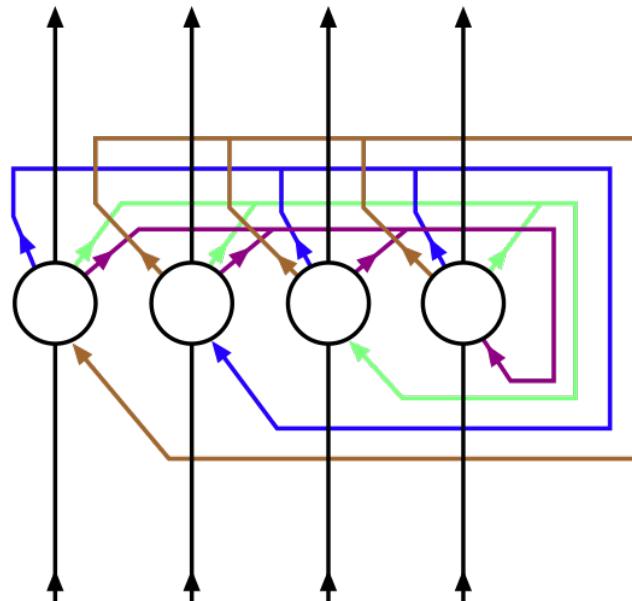


Figura 13: Esempio di reti di Hopfield con 4 neuroni

Si consideri una rete di Hopfield formata da $n \in \mathbb{N}_+$ neuroni, ognuno dei quali è associato ad un vettore dei pesi $w_i \in \mathbb{R}^n$, con $1 \leq i \leq n$, è possibile definire la **matrice dei pesi della rete** $W = (w_{i,j})_{n,n}$. Si noti che, come già discusso, la matrice dei pesi ha solo zeri sulla diagonale principale perché la rete non presenta autoanelli. Fissata una rete di Hopfield mediante la relativa matrice dei pesi della rete W e il relativo **vettore di bias** b , è possibile definire l'**energia della rete** associata ad uno stato $s \in \mathbb{R}^n$ della rete come

$$E(s) = \frac{1}{2} s^\top W s + b \cdot s = -\frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n w_{i,j} s_i s_j + \sum_{i=1}^n b_i s_i \quad (60)$$

serve per indicare che
decrease
vettore di n componenti
con solo -1 o 1
matrice pesi
vettore bias

Si noti che, come di consueto, i vettori di \mathbb{R}^n vengono trattati come vettori colonna quando vengono considerati insieme a delle matrici. In più, si noti che la parola **energia** è stata scelta per la quantità $E(s)$ perché vale il seguente teorema.

Teorema 2 (Di Convergenza delle Reti Neurali di Hopfield). *Si consideri una rete neurale di Hopfield formata da $n \in \mathbb{N}_+$ neuroni e descritta da una matrice dei pesi W e da un vettore di bias b . Se la matrice dei pesi è simmetrica*, allora, indipendentemente dallo stato iniziale, tutti i neuroni della rete tenderanno ad uno stato stazionario che corrisponde ad un minimo locale dell'energia della rete.*

Dimostrazione. Si consideri un passo della simulazione asincrona della rete e sia $1 \leq a \leq n$ l'indice del neurone il cui stato viene aggiornato nel passo considerato. Se $s \in \mathbb{R}^n$ è lo stato della rete prima del passo di simulazione e $s' \in \mathbb{R}^n$ è lo stato della rete dopo il passo di simulazione, allora la variazione dell'energia da $E(s)$ a $E(s')$ vale $\Delta E = E(s') - E(s)$

$$\Delta E = -\frac{1}{2} \left(\sum_{i=1}^n w_{i,a} s'_i s'_a + \sum_{j=1}^n w_{a,j} s'_a s'_j - \sum_{i=1}^n w_{i,a} s_i s_a - \sum_{j=1}^n w_{a,j} s_a s_j \right) + b_a (s'_a - s_a) \quad (61)$$

dove si è sfruttato il fatto che $w_{a,a} = 0$ per ipotesi. Ma, siccome, per ipotesi, la matrice dei pesi della rete è simmetrica, vale

$$\Delta E = - \left(\sum_{i=1}^n w_{i,a} s'_i s'_a - \sum_{i=1}^n w_{i,a} s_i s_a \right) + b_a (s'_a - s_a) \quad (62)$$

e quindi

$$\begin{aligned} \Delta E &= - \left(s'_a \sum_{i=1}^n w_{i,a} s'_i - s_a \sum_{i=1}^n w_{i,a} s_i \right) + b_a (s'_a - s_a) \\ &= -(s'_a - s_a) \left(\sum_{i=1}^n w_{i,a} s_i - b_a \right) \end{aligned} \quad (63)$$

perché, nel passo di aggiornamento considerato, solo s_a può cambiare e quindi, per ogni $1 \leq i \leq n$, con $i \neq a$, vale $s'_i = s_i$. In più, si noti che

$$\left(\sum_{i=1}^n w_{i,a} s_i - b_a \right) \xrightarrow{\text{Preattività}} \quad (64)$$

è il valore dell'uscita del neurone a prima dell'applicazione della funzione signum e quindi il suo segno è uguale al segno di s'_a . Quindi,

- Se l'uscita del neurone a non varia a causa dell'aggiornamento, allora $\Delta E = 0$ e $E'(\mathbf{s}) = E(\mathbf{s})$;
- Se l'uscita del neurone a varia a causa dell'aggiornamento e diventa 0, allora $\Delta E = 0$ e $E'(\mathbf{s}) = E(\mathbf{s})$;
- Se l'uscita del neurone a varia a causa dell'aggiornamento e diventa +1, allora $\Delta E < 0$ e $E'(\mathbf{s}) \leq E(\mathbf{s})$; e
- Se l'uscita del neurone a varia a causa dell'aggiornamento e diventa -1, allora $\Delta E < 0$ e $E'(\mathbf{s}) \leq E(\mathbf{s})$

Quindi, in sintesi, l'evoluzione dinamica dello stato della rete può solo fare decrescere o rimanere inalterata l'energia della rete, che quindi arriverà ad assestarsi asintoticamente ad un minimo locale. \square

Si noti che, fissata una rete di Hopfield, E non è l'unica funzione che non cresce mai durante l'evoluzione dinamica della rete e, quindi, esistono altre funzioni che possono essere usate come energia della rete.

memoria che viene acceduta per contenuto, non per indirizzo

Una rete di Hopfield può essere usata come memoria associativa se, dato un insieme di vettori di ingresso detto training set, è possibile costruire una rete il cui stato evolva dinamicamente e si assesti ad uno dei vettori del training set simile al vettore di ingresso. In sostanza, quindi, lo stato della rete viene inizializzato con un vettore di ingresso e si prevede che la rete converga asintoticamente ad uno stato, tra quelli nel training set, per cui è alta la somiglianza con lo stato di ingresso.

Dato un training set, l'addestramento di una rete di Hopfield prevede di calcolare una matrice dei pesi e un vettore di bias che si comportino bene quando utilizzati con un adeguato test set. Si noti che, come si evince dal seguente teorema, l'addestramento di una rete di Hopfield è non supervisionato perché non si prevede che i vettori del training

* MUTUAMENTE ORTOGONALI

Un insieme finito di vettori reali non nulli $O \subset \mathbb{R}^n$ si dice formato da vettori **mutuamente ortogonali** se e soltanto se, qualsiasi siano $\mathbf{x} \in O$ e $\mathbf{y} \in O$, vale $\mathbf{x} \cdot \mathbf{y} = 0$. Si noti che gli insiemi di vettori mutuamente ortogonali sono particolarmente interessanti perché vale la seguente proposizione.

Proposizione 3. *Dato $n \in \mathbb{N}_+$, se $O \subset \mathbb{R}^n$ è un insieme di vettori reali mutuamente ortogonali, allora è anche un insieme di vettori linearmente indipendenti.*

set siano associati a valori di uscita corretti. Questo rende le reti di Hopfield più semplici da utilizzare con training set di grandi dimensioni perché non è necessario associare ad ogni vettore del training set il corrispettivo valore atteso.

Teorema 3 (Di Addestramento con la Regola di Hebb). *Data una rete neurale di Hopfield con $n \in \mathbb{N}_+$ neuroni, si consideri un training set $T = \{\mathbf{x}_k\}_{k=1}^m$ formato da $m \in \mathbb{N}_+$ vettori di \mathbb{R}^n tra loro mutuamente ortogonali e privi di componenti nulle, se la rete ha vettore di bias nullo e matrice dei pesi*

$$W = \frac{1}{m} \sum_{k=1}^m \mathbf{x}_k \mathbf{x}_k^\top - I_n \quad (65)$$

dove I_n è la matrice identità $n \times n$, allora l'energia della rete ammette minimi locali in corrispondenza dei vettori del training set.

Dimostrazione. Per prima cosa si noti che i vettori del training set sono vettori di \mathbb{R}^n e, quindi, sarà $m \leq n$ perché è noto che un insieme di vettori mutuamente ortogonali è anche un insieme di vettori linearmente indipendenti. In più, si noti che la matrice considerata ha i seguenti elementi, per $1 \leq i \leq n$ e $1 \leq j \leq n$,

$$w_{i,j} = \frac{1}{m} \sum_{k=1}^m x_{k,i} x_{k,j} - \delta_{i,j} \quad (66)$$

dove $\delta_{i,j}$ è la **delta di Kronecker**

$$\delta_{i,j} = \begin{cases} 1 & \text{se } i = j \\ 0 & \text{se } i \neq j \end{cases} \quad (67)$$

Quindi, la matrice W è simmetrica e ha solo zeri sulla diagonale principale perché si è ipotizzato che i vettori del training set non abbiano componenti nulle. Si consideri la derivata dell'energia rispetto ad una generica componente dello stato della rete di indice $1 \leq a \leq n$

$$\frac{\partial E}{\partial s_a}(\mathbf{s}) = -\frac{1}{2} \left(\sum_{i=1}^n w_{i,a} s_i + \sum_{j=1}^n w_{a,j} s_j \right) \quad (68)$$

dove si è sfruttato il fatto che $w_{a,a} = 0$ e si è ricordato che, per ipotesi, il vettore di bias della rete è nullo. Siccome la matrice dei pesi considerata è simmetrica, vale

$$\frac{\partial E}{\partial s_a}(\mathbf{s}) = - \sum_{i=1}^n w_{i,a} s_i \quad (69)$$

e quindi

$$\frac{\partial E}{\partial s_a}(\mathbf{s}) = -\frac{1}{m} \sum_{k=1}^m \sum_{i=1}^n s_i x_{k,i} x_{k,a} + \sum_{i=1}^n \delta_{i,a} s_i \quad (70)$$

Si ricordi ora che i vettori del training set sono tra loro mutuamente ortogonali e quindi, per ogni $1 \leq k \leq m$ e $1 \leq h \leq m$, con $h \neq k$, vale

$$\mathbf{x}_k \cdot \mathbf{x}_h = \sum_{r=1}^n x_{k,r} x_{h,r} = 0 \quad (71)$$

Quindi, se la derivata dell'energia viene valutata in un generico vettore del training set di indice $1 \leq h \leq m$, vale

$$\frac{\partial E}{\partial s_a}(\mathbf{x}_h) = -\frac{1}{m} m x_{h,a} + x_{h,a} = 0 \quad (72)$$

derivata dell'Energia in uno stato a , in un vettore del training set n

perché, siccome i vettori del training set non hanno componenti nulle, vale che $x_{h,i} x_{h,i} = 1$ per ogni $1 \leq h \leq m$ e $1 \leq i \leq n$. \square

Si noti che il teorema precedente non assicura che gli unici minimi locali della funzione energia siano quelli relativi ai vettori del training set e, quindi, fissato uno stato iniziale, non è garantito che la rete converga proprio ad uno dei vettori del training set. In generale, quindi, il **richiamo** di un vettore dalla rete, cioè la sua lettura a fronte di un vettore di ingresso, potrebbe essere non esatto. In sintesi, nonostante l'addestramento di una rete di Hopfield non sia supervisionato, l'accuratezza nel richiamo dei vettori deve sempre essere quantificata mediante un'analisi con un opportuno test set.

Seguono alcuni esempi di richiamo di vettori mediante una rete addestrata con la **regola di Hebb**, che è la regola per la costruzione della matrice dei pesi del teorema precedente. Si noti che i vettori utilizzati per l'addestramento non sono mutuamente ortogonali, ma viene comunque usata la regola di Hebb ipotizzando di ottenere comunque buoni risultati.

Si consideri una rete di Hopfield in cui è stato memorizzato, mediante la regola di Hebb, il vettore mostrato nella parte destra della Figura 14.

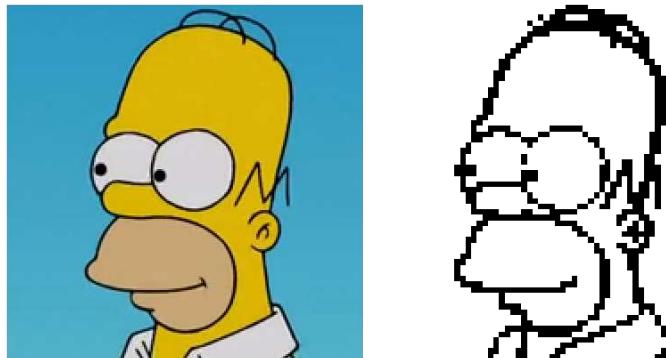


Figura 14: Immagine usata per alcuni esperimenti con una rete di Hopfield con $n = 64 \times 64$ neuroni (sinistra) e il relativo vettore binario usato per l'addestramento della rete (destra)

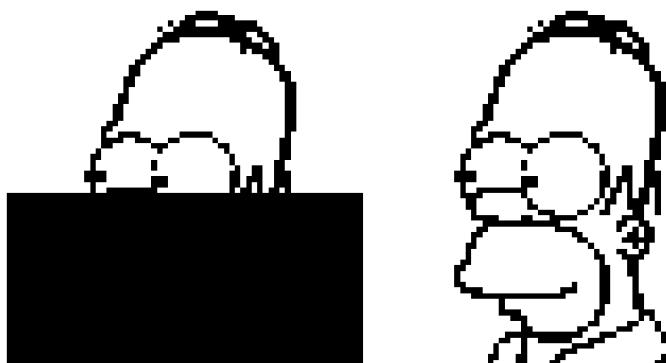


Figura 15: Vettore usato come ingresso della rete di Hopfield addestrata con il vettore di Figura 14 (sinistra) e l'uscita ottenuta dalla rete (destra)

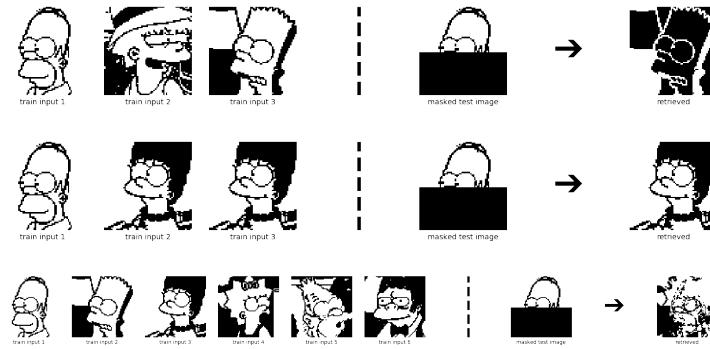


Figura 16: Alcuni esempi in cui la regola di Hebb per l’addestramento di una rete di Hopfield non consente di richiamare correttamente alcuni vettori

A fronte del vettore mostrato in Figura 15 la rete ha richiamato il vettore originale, come richiesto. Quindi, in questo caso, la rete è stata in grado di richiamare il vettore corretto anche in presenza di un forte rumore.

Però, la stessa rete, una volta che è stato aumentato il numero di vettori memorizzati, come mostrato in Figura 16 inizia a comportarsi male e il suo comportamento peggiora al crescere del numero di vettori memorizzati. Si noti che gli errori nel richiamo dei vettori non sono necessariamente legati all’eccessivo numero di vettori memorizzati nella rete. Ad esempio, in questo caso, il numero di vettori memorizzati è sufficientemente piccolo da ritenere che gli errori non siano dovuti alla quantità di vettori memorizzati, ma alla loro dipendenza lineare. In questo caso, infatti, si vede come la rete tenda a richiamare dei vettori ottenuti componendo vari vettori memorizzati.

Appunti del Corso di Intelligenza Artificiale

Problemi di Soddisfacimento di Vincoli

Prof. Federico Bergenti

21 aprile 2024

1 Problemi di Soddisfacimento di Vincoli

Un **problema di soddisfacimento di vincoli** (**Constraint Satisfaction Problem, CSP**) è una tripla $\langle V, D, C \rangle$ in cui

- $V \neq \emptyset$ è un insieme non vuoto e finito di simboli detti variabili con $n = |V|$;
- $D \neq \emptyset$ è un insieme non vuoto di insiemi detti **dominî** delle variabili con $|D| \leq n$; e
- C è un insieme finito di **vincoli**.

Detta $dom : V \rightarrow D$ una funzione suriettiva totale che associa un dominio ad ogni variabile, si può parlare, per ogni variabile $x \in V$, del suo dominio $dom(x)$. In più, si suppone sempre che esista un ordinamento totale delle variabili e che questo ordinamento venga sempre usato in senso crescente quando vengono elencate le variabili. Normalmente, l'ordinamento utilizzato per le variabili è lasciato implicito ed evidente dal contesto. Utilizzando l'ordinamento delle variabili è possibile definire il dominio di un CSP \mathcal{P} come

$$dom(\mathcal{P}) = \prod_{x \in V} dom(x) \quad (1)$$

dove V è l'insieme delle variabili del CSP e le variabili vengono elencate nella costruzione del prodotto cartesiano in senso crescente secondo l'ordinamento scelto.

Dato un CSP con variabili in V , ogni vincolo $c \in C$ è una coppia $\langle V_c, \Delta_c \rangle$ dove $V_c \subseteq V$ è un insieme di variabili del CSP e Δ_c è

$$\Delta_c \subseteq \prod_{x \in V_c} dom(x) \quad (2)$$

dove le variabili vengono elencate nella costruzione del prodotto cartesiano in senso crescente secondo l'ordinamento scelto e Δ_c viene detto insieme degli **assegnamenti (parziali) consistenti (o ammissibili)** del vincolo c . Se $|V_c| = 1$, allora c viene detto **unario**, se $|V_c| = 2$, allora c viene detto **binario**, mentre c viene detto **globale** in tutti gli altri casi. Infine, si noti che $vars(c) = V_c$ è un modo per fare riferimento alle variabili di un vincolo e

$$dom(c) = \prod_{x \in V_c} dom(x) \quad (3)$$

viene detto dominio del vincolo c se si assume che le variabili vengano elencate in senso crescente secondo l'ordinamento scelto.

Fissato un CSP \mathcal{P} , è sempre possibile estendere Δ_c ad un sottoinsieme del dominio del CSP $\Delta = \text{dom}(\mathcal{P})$ aggiungendo alle ennuple di Δ_c le componenti mancanti. Siccome se una variabile non è coinvolta in un vincolo i suoi valori sono tutti e soli quelli del proprio dominio, le componenti aggiunte alle ennuple di Δ_c potranno assumere un valore qualsiasi nei dominî delle rispettive variabili. Detta img una funzione suriettiva totale che associa ad ogni vincolo l'estensione a Δ dell'insieme degli assegnamenti (parziali) consistenti, si può parlare, per ogni vincolo $c \in C$, del suo insieme degli **assegnamenti totali consistenti (o ammissibili)** $\text{img}(c) \subseteq \Delta$.

Se tutti i dominî delle variabili di un vincolo sono finiti, allora è possibile descrivere il vincolo elencando tutti gli assegnamenti parziali consistenti in modo **estensionale** e il vincolo viene detto **tabellare**. La rappresentazione tabellare diventa impraticabile anche per dominî con pochi elementi e quindi spesso si preferisce la rappresentazione **intensionale**.

Esempio 1. Si consideri un CSP con tre variabili, x , y e z , tali che $\text{dom}(x) = \text{dom}(y) = \text{dom}(z) = [1..6]$. L'insieme delle variabili del CSP è $V = \{x, y, z\}$ e il dominio del CSP è $\Delta = [1..6]^3$ dove, come di consueto, le variabili vengono ordinate in senso alfabetico. Il CSP contiene un vincolo c definito dalla proprietà

$$x \text{ e } z \text{ devono essere diversi ed entrambi pari}$$

Quindi, $c = \langle V_c, \Delta_c \rangle$ dove $V_c = \{x, z\}$ e Δ_c può essere espresso in forma estensionale come segue

$$\Delta_c = \{(2, 4), (2, 6), (4, 2), (4, 6), (6, 2), (6, 4)\} \quad (4)$$

dove, come di consueto, è stato utilizzato l'ordinamento alfabetico per le variabili. Si noti che l'estensione di Δ_c ad un sottoinsieme di Δ può essere espressa in forma intensionale come segue

$$\text{img}(c) = \{(2, 1, 4), (2, 2, 4), \dots, (6, 6, 4)\} \quad (5)$$

dove ogni elemento di Δ_c è stato utilizzato per produrre 6 elementi di $\text{img}(c)$. L'insieme degli assegnamenti parziali consistenti di c può essere espresso in forma intensionale come segue

$$\Delta_c = \{(x, z) : x \in I \wedge z \in I \wedge x \neq z \wedge x \bmod 2 = 0 \wedge z \bmod 2 = 0\} \quad (6)$$

e quindi

$$\text{img}(c) = \{(x, y, z) : x \in I \wedge y \in I \wedge z \in I \wedge x \neq z \wedge x \bmod 2 = 0 \wedge z \bmod 2 = 0\} \quad (7)$$

La possibilità di estendere gli insiemi degli assegnamenti parziali consistenti ad insiemi di assegnamenti totali consistenti permette di definire in modo semplice cosa si intenda per soluzione di un CSP. Dato un CSP $\mathcal{P} = \langle V, D, C \rangle$, l'insieme delle **soluzioni** di \mathcal{P} è

$$\text{sol}(\mathcal{P}) = \bigcap_{c \in C} \text{img}(c) \quad (8)$$

e il \mathcal{P} si dice **risolubile** (o **risolvibile**) se questo insieme non è vuoto. Dati due CSP \mathcal{P}_1 e \mathcal{P}_2 definiti sullo stesso insieme di variabili, si dice che \mathcal{P}_1 è **equivalente** a \mathcal{P}_2 se $\text{sol}(\mathcal{P}_1) = \text{sol}(\mathcal{P}_2)$. Si noti che questa definizione può essere estesa a problemi con lo stesso numero di variabili rinominando opportunamente le variabili dei problemi.

Dato un CSP è interessante cercare delle trasformazioni in grado di generare dei CSP ad esso equivalenti ma che siano più utili per la ricerca delle soluzioni. In generale, però,

sono interessanti anche le trasformazioni in grado di produrre dei CSP non equivalenti a quello di partenza ma che permettano, comunque, di studiare le proprietà di quest'ultimo.

Esempio 2. Si consideri un CSP \mathcal{P} con due variabili x e y entrambe con dominio $[-10..10]$. Il CSP ha un unico vincolo descritto dalla proprietà $y = x^2$. Si noti che il CSP è risolubile perché, ad esempio, $x = -2$ e $y = 4$ è una soluzione. Il CSP \mathcal{P}_1 che restringe il dominio di y a $[0..10]$ è equivalente a \mathcal{P} perché è facile vedere che entrambi ammettono lo stesso insieme di soluzioni. Viceversa, il CSP \mathcal{P}_2 che restringe anche il dominio di x all'insieme $[0..10]$ non è equivalente a \mathcal{P} perché, ad esempio, $x = -2$ e $y = 4$ è una soluzione di \mathcal{P} ma non è una soluzione di \mathcal{P}_2 . Però, il CSP \mathcal{P}_2 , che è ottenuto da \mathcal{P} con una trasformazione detta **di rottura della simmetria** (o **symmetry breaking**), è comunque interessante perché una volta risolto è possibile ottenere le soluzioni di \mathcal{P} . In più, \mathcal{P}_2 ha un numero inferiore di elementi nel dominio di x e quindi la ricerca esaustiva delle soluzioni è facilitata.

Dato un CSP, ogni trasformazione che produce un CSP ad esso equivalente ma con meno vincoli o meno valori nei dominî viene detta **propagazione dei vincoli**. Se la trasformazione utilizza un sottoinsieme dei vincoli, allora viene detta **propagazione locale (dei vincoli)**. Se la trasformazione utilizza un sottoinsieme dei vincoli per rimuovere alcuni valori dai dominî delle variabili coinvolte nei vincoli considerati, allora viene detta **filtro (dei valori)**.

Normalmente, le trasformazioni tra CSP equivalenti vengono utilizzate per ottenere nuovi CSP con proprietà interessanti. Ad esempio, una trasformazione che spesso viene utilizzata permette di ottenere un CSP nodo-consistente equivalente ad un CSP dato. Si noti che un CSP si dice **nodo-consistente** (o **node-consistent**) se per ogni vincolo unario c vale la seguente proprietà

$$\forall v \in \text{dom}(x), v \in \Delta_c \quad (9)$$

dove x è l'unica variabile del vincolo c e Δ_c è l'insieme degli assegnamenti parziali consistenti di c .

Un **algoritmo di nodo-consistenza** lavora su un CSP per produrre un CSP nodo-consistente ad esso equivalente eliminando dai dominî delle variabili tutti i valori che non rispettano la definizione precedente. In più, un algoritmo di nodo-consistenza rimuove anche tutti i vincoli unari dal CSP perché, una volta eliminati i valori inconsistenti dai dominî, i vincoli unari non sono più necessari.

In modo simile, un **algoritmo di arco-consistenza** lavora su un CSP per produrre un CSP arco-consistente ad esso equivalente eliminando dai dominî delle variabili tutti i valori che non rispettano la definizione di arco-consistenza. Un CSP si dice **arco-consistente** (o **arc-consistent**) se per ogni vincolo binario c valgono le seguenti proprietà

$$\forall v_x \in \text{dom}(x), \exists v_y \in \text{dom}(y) : (v_x, v_y) \in \Delta_c \quad (10)$$

$$\forall v_y \in \text{dom}(y), \exists v_x \in \text{dom}(x) : (v_x, v_y) \in \Delta_c \quad (11)$$

dove x e y sono le due variabili del vincolo c e Δ_c è l'insieme degli assegnamenti parziali consistenti di c .

Si noti che esistono vari algoritmi di arco-consistenza che assumono che i dominî delle variabili siano finiti. L'algoritmo normalmente più utilizzato è AC-3, che ha una complessità temporale asintotica di caso pessimo di classe $O(e k^3)$, dove e è il numero di vincoli binari e k è la cardinalità massima dei dominî delle variabili coinvolte nei vincoli binari. Si noti, comunque, che la complessità temporale asintotica di caso pessimo di un algoritmo

di arco-consistenza ottimo è di classe $O(e k^2)$, a cui si può arrivare con l'algoritmo AC-4 mediante un significativo incremento della memoria utilizzata rispetto ad AC-3.

In più, si noti che la proprietà di arco-consistenza può essere estesa a vincoli globali introducendo la cosiddetta **iperarco-consistenza**. Infine, si noti che la nodo-consistenza coinvolge una sola variabile mentre la arco-consistenza coinvolge due variabili. Quindi è ragionevole aspettarsi di poter definire la cosiddetta **consistenza di percorso** costruendo percorsi di più di due variabili collegate da vincoli binari.

2 Problemi con Vincoli Polinomiali

La consistenza di nodo e la consistenza di arco permettono di rimuovere valori dai dominî delle variabili e sono spesso efficaci, specialmente se i vincoli sono espressi in forma tabellare. Viceversa, se i vincoli sono espressi in forma intensionale, ad esempio perché i dominî delle variabili non sono finiti, vengono spesso utilizzate altre forme di consistenza. Un esempio interessante a questo riguardo è rappresentato dai vincoli espressi mediante equazioni, disequazioni e disuguaglianze tra polinomi a coefficienti di variabili reali.

Si consideri un CSP \mathcal{P} con $n \in \mathbb{N}_+$ variabili in $V = \{x_i\}_{i=1}^n$ alle quali sono associati i dominî $dom(x_i) = [\underline{x}_i, \bar{x}_i]$ rappresentati da intervalli chiusi in \mathbb{R} . Il problema può contenere anche più variabili, ma quelle di V sono quelle interessanti per i vincoli espressi mediante polinomi. Un vincolo c sulle variabili $V_c \subseteq V$ di \mathcal{P} si dice **polinomiale** se il suo insieme degli assegnamenti parziali consistenti può essere espresso in modo intensionale come

$$\Delta_c = \{\mathbf{x} \in dom(c) : p(\mathbf{x}) \odot q(\mathbf{x})\} \quad (12)$$

dove $\odot \in \{<, \leq, =, \neq, \geq, >\}$, $p : \mathbb{R}^k \rightarrow \mathbb{R}$ e $q : \mathbb{R}^k \rightarrow \mathbb{R}$ sono due funzioni polinomiali e $k = |V_c|$.

Si noti subito che ogni vincolo polinomiale può essere ridotto ad una delle due seguenti forme

$$\{\mathbf{x} \in dom(c) : p(\mathbf{x}) \geq 0\} \quad \text{oppure} \quad \{\mathbf{x} \in dom(c) : p(\mathbf{x}) > 0\} \quad (13)$$

utilizzando semplici manipolazioni algebriche e sfruttando il fatto che, data una funzione polinomiale $p : \mathbb{R}^k \rightarrow \mathbb{R}$, vale, per ogni $\mathbf{x} \in \mathbb{R}^k$,

$$p(\mathbf{x}) = 0 \iff p(\mathbf{x}) \leq 0 \wedge p(\mathbf{x}) \geq 0 \quad (14)$$

$$p(\mathbf{x}) \neq 0 \iff p^2(\mathbf{x}) > 0 \quad (15)$$

Infine, si noti che è sempre possibile riscrivere i vincoli in modo che tutte le variabili abbiano dominî definiti come intervalli chiusi di \mathbb{R}_+ mediante opportune trasformazioni affini operate sulle variabili. Infatti, essendo tutti i dominî delle variabili degli intervalli chiusi, è sufficiente traslare tutti gli intervalli per garantire che le variabili abbiano dominî definiti come intervalli chiusi di \mathbb{R}_+ .

Un CSP si dice **consistente sugli intervalli** (o **bounds consistent**) se per ogni vincolo polinomiale c vale che per ognuna delle sue variabili x con dominio $[\underline{x}, \bar{x}]$ esiste almeno un assegnamento parziale consistente del vincolo che abbia il valore \underline{x} per x ed, eventualmente un altro, assegnamento parziale consistente che abbia il valore \bar{x} per x .

Quindi, indipendentemente dalla consistenza dei valori all'interno dell'intervallo $[\underline{x}, \bar{x}]$, la bounds consistency considera solo la consistenza dei valori ai due estremi di ogni intervallo coinvolto nel vincolo. Si noti che normalmente si è interessati al più piccolo intervallo, nel senso del contenimento, che garantisca la bounds consistency di una variabile in uno o più vincoli del problema.

Un algoritmo di bounds consistency è un algoritmo che trasforma un CSP in un secondo CSP ad esso equivalente in cui gli intervalli che definiscono i domini delle variabili garantiscono la bounds consistency. Come già discusso, un algoritmo di bounds consistency può considerare solo variabili definite su \mathbb{R}_+ e vincoli in forma di disequazione, stretta o meno. In queste ipotesi, è possibile definire un opportuno algoritmo di bounds consistency sfruttando la seguente proposizione.

Proposizione 1. *Dato $n \in \mathbb{N}$, sia $p : \mathbb{R}^n \rightarrow \mathbb{R}$ una funzione polinomiale a coefficienti reali positivi, se $\underline{\mathbf{x}} \in \mathbb{R}_+^n$ e $\bar{\mathbf{x}} \in \mathbb{R}_+^n$, allora per ogni $\mathbf{v} \in [\underline{\mathbf{x}}, \bar{\mathbf{x}}]$ vale*

$$p(\underline{\mathbf{x}}) \leq p(\mathbf{v}) \leq p(\bar{\mathbf{x}}) \quad (16)$$

nell'ipotesi non restrittiva che per ogni $1 \leq i \leq n$ valga $\underline{x}_i \leq \bar{x}_i$.

La descrizione generale di un algoritmo di bounds consistency che sfrutti la proposizione precedente richiede l'introduzione della notazione dei **multi-indici**. Quindi, anziché descrivere un algoritmo di bounds consistency nella sua generalità, verrà descritto sommariamente un metodo che permette di ridurre i domini delle variabili sfruttando il fatto che queste siano definite su intervalli chiusi di \mathbb{R}_+ .

Si consideri vincolo descritto dalla proprietà $p(\mathbf{x}) \geq 0$ nel caso in cui p sia una funzione lineare a coefficienti reali di $n \in \mathbb{N}_+$ variabili

$$p(\mathbf{x}) = \sum_{i=1}^n a_i x_i + b \quad (17)$$

con, per semplicità $b \geq 0$. La proprietà che definisce il vincolo può essere riscritta spostando a destra tutti i termini di p con coefficienti negativi e quindi

$$\sum_{a_i > 0} a_i x_i + b \geq \sum_{a_i < 0} -a_i x_i \quad (18)$$

Fissata una variabile di indice k , se $a_k > 0$ è possibile scrivere

$$\sum_{i \neq k, a_i > 0} a_i x_i + a_k x_k + b \geq \sum_{a_i < 0} -a_i x_i \quad (19)$$

Sfruttando la proposizione precedente è possibile ottenere

$$\sum_{i \neq k, a_i > 0} a_i x_i \leq \sum_{i \neq k, a_i > 0} a_i \bar{x}_i = u \quad l = \sum_{a_i < 0} -a_i \underline{x}_i \leq \sum_{a_i < 0} -a_i x_i \quad (20)$$

e quindi la proprietà che descrive il vincolo impone che

$$x_k \geq \frac{l - u - b}{a_k} \quad (21)$$

per ogni variabile x_k che viene moltiplicata in p per un coefficiente positivo. Si noti subito che è semplice rimuovere l'ipotesi $b \geq 0$ e che, ragionamenti simili a quelli appena visti, consentono di identificare una disequazione da utilizzare per le variabili che vengono moltiplicate per un coefficiente negativo in p . Entrambe queste disequazioni possono anche essere utilizzate per trattare vincoli descritti mediante disequazioni strette, previo cambiamento del simbolo di diseguaglianza. Si capisce quindi facilmente come la possibilità di

stimare i valori massimi e minimi di un polinomio di una variabile in un intervallo chiuso sia di fondamentale importanza per gli algoritmi di bounds consistency.

Tutte queste disequazioni possono essere utilizzate ripetutamente per restringere i dominî delle variabili del vincolo considerato fino al raggiungimento di almeno una delle seguenti condizioni:

1. L'applicazione delle disequazioni non genera nuovi restringimenti dei dominî e quindi è possibile cercare le soluzioni del CSP nei nuovi dominî; oppure
2. L'applicazione delle disequazioni ha svuotato almeno uno dei dominî e quindi è stato dimostrato che il CSP non ammette soluzioni.

I due esempi che seguono mostrano come applicare questo metodo nei due casi appena descritti. In particolare, il seguente esempio consente di restringere i dominî delle variabili.

Esempio 3. Si consideri un CSP con tre variabili x , y e z i cui dominî sono $\text{dom}(x) = \text{dom}(y) = \text{dom}(z) = [1, 10]$. Si consideri il vincolo definito dalla proprietà

$$2x + 3y - z \leq 1 \quad (22)$$

La proprietà che definisce il vincolo può essere riscritta come

$$z + 1 \geq 2x + 3y \quad (23)$$

e quindi

$$z + 1 \geq 2x + 3y \geq 5 \quad (24)$$

da cui si evince che $z \geq 4$. Ma, ragionando sulla variabile x si ottiene

$$11 \geq z + 1 \geq 2x + 3y \geq 2x + 3 \quad (25)$$

da cui si evince che $x \leq 4$. Infine, ragionando sulla variabile y si ottiene

$$11 \geq z + 1 \geq 2x + 3y \geq 2 + 3y \quad (26)$$

da cui si evince che $y \leq 3$. Si noti che questi ragionamenti possono essere iterati finché nessun dominio viene più modificato oppure finché almeno uno dei dominî non diventa l'insieme vuoto. Nel primo caso si è ottenuto un CSP bounds consistent equivalente al CSP iniziale, mentre nel secondo caso è stato dimostrato che il CSP iniziale non ammette soluzioni. Provando ad iterare i ragionamenti fatti si nota subito che i dominî non vengono ulteriormente ridotti e quindi è stato ottenuto un nuovo CSP, equivalente a quello iniziale ma bounds consistent, con tre variabili x , y e z aventi dominî

$$\text{dom}(x) = [1, 4] \quad \text{dom}(y) = [1, 3] \quad \text{dom}(z) = [4, 10] \quad (27)$$

e un solo vincolo che richiede che valga la proprietà $2x + 3y - z \leq 1$. Si noti che il vincolo non può essere rimosso perché la proprietà di bounds consistency non fornisce informazioni sulla consistenza dei valori interni agli intervalli considerati. La bounds consistency studia unicamente gli estremi dei dominî delle variabili.

Il seguente esempio mostra come sia possibile stabilire che il CSP considerato non ammette soluzioni. Infatti, l'applicazione delle disequazioni studiate in precedenza consente di ridurre all'insieme vuoto il dominio di almeno una variabile, terminando il processo di riduzione e stabilendo che il CSP non ammette soluzioni.

Esempio 4. Si consideri un CSP con tre variabili x , y e z i cui dominî sono $dom(x) = dom(y) = dom(z) = [1, 10]$. Si consideri il vincolo definito dalla proprietà

$$12x + 8y - z \leq 1 \quad (28)$$

La proprietà che definisce il vincolo può essere riscritta come

$$z + 1 \geq 12x + 8y \geq 20 \quad (29)$$

e quindi

$$z + 1 \geq 12x + 8y \geq 20 \quad (30)$$

da cui si evince che il vincolo richiede che $z \geq 19$. Però, $z \leq 10$ per ipotesi e quindi è stato facilmente dimostrato che il CSP non ammette soluzioni.

Si noti che il metodo per la riduzione dei dominî delle variabili descritto sommariamente nel caso di funzioni lineari può essere esteso facilmente al caso di funzioni polinomiali non lineari ricordando che, nelle ipotesi considerate, se una variabile x ha per dominio $dom(x) = [\underline{x}, \bar{x}] \subseteq \mathbb{R}_+$, allora se $m \in \mathbb{N}_+$

$$\underline{x}^{m-1} x \leq x^m \leq \bar{x}^{m-1} \bar{x} \quad (31)$$

e quindi i termini non lineari del tipo x^m possono essere ridotti a termini lineari. In modo simile, se una seconda variabile y ha dominio $dom(y) = [\underline{y}, \bar{y}] \subseteq \mathbb{R}_+$, allora

$$\underline{x}\underline{y} \leq xy \leq \bar{x}\bar{y} \quad \text{e} \quad \underline{y}\underline{x} \leq xy \leq \bar{y}\bar{x} \quad (32)$$

e quindi i termini non lineari del tipo xy possono essere ridotti a termini lineari.

I vincoli polinomiali vengono spesso utilizzati restringendo i dominî delle variabili a intervalli in \mathbb{Z}_+ e richiedendo che i coefficienti delle funzioni polinomi siano in \mathbb{Z} . In questo caso si parla di **vincoli polinomiali a dominî finiti** ed è possibile sfruttare le particolarità di questo tipo di vincoli per definire degli algoritmi di propagazione dei vincoli efficaci ed efficienti. In particolare, il fatto che le variabili abbiano dominî in \mathbb{Z}_+ e il fatto che i coefficienti delle funzioni polinomiali siano in \mathbb{Z} permette di ridurre lo studio di questo tipo di vincoli a sole disequazioni non strette perché vale

$$p(\mathbf{x}) > 0 \iff p(\mathbf{x}) - 1 \geq 0 \quad (33)$$

per una qualsiasi funzione polinomiale a coefficienti interi e variabili intere positive. In più, si noti che, siccome le variabili sono ristrette ad assumere solo valori interi, in questo caso viene adottata la notazione degli intervalli interi e si scrive $[\underline{x}.. \bar{x}]$, con $\underline{x} \in \mathbb{Z}$, $\bar{x} \in \mathbb{Z}$ e $\underline{x} \leq \bar{x}$, per indicare l'intervallo contenente tutti gli interi maggiori o uguali a \underline{x} e minori o uguali a \bar{x} . Naturalmente, $[\underline{x}.. \bar{x}] = \emptyset$ se $\underline{x} > \bar{x}$.

Appunti del Corso di Intelligenza Artificiale

Programmazione Logica

Prof. Federico Bergenti

10 maggio 2024

1 Paradigmi di Programmazione

Seguendo l'approccio introdotto da *Robert W. Floyd*, ogni linguaggio di programmazione può essere descritto nei termini di uno specifico **paradigma di programmazione** che ne riassume le peculiarità elencando in modo astratto le caratteristiche degli **esecutori** in grado di produrre una **computazione** partendo da un programma scritto nel linguaggio. Spesso i linguaggi di programmazione seguono più paradigmi, ma normalmente è possibile identificare un paradigma principale. Solo raramente i linguaggi di programmazione sono veramente **multi-paradigma**.

I paradigmi di programmazione che vengono tradizionalmente identificati sono:

1. Paradigma **imperativo**: in cui ogni esecutore è una macchina di Turing e un programma descrive in modo esplicito e dettagliato quali azioni deve compiere l'esecutore per risolvere un problema.
2. Paradigma **dichiarativo**: in cui ogni esecutore è in grado di trovare una soluzione ad una classe di problemi mediante una tecnica risolutiva di uso generale e un programma descrive in modo esplicito e dettagliato un problema da risolvere.

Quindi, nella programmazione imperativa, un programma descrive una procedura per risolvere una classe di problemi e l'esecutore si limita ad applicare la procedura per risolvere una specifica istanza del problema. Viceversa, nella programmazione dichiarativa, un programma descrive un problema da risolvere in modo che l'esecutore possa risolverlo mediante l'applicazione del metodo risolutivo che ha a disposizione. Usando uno slogan: *Declarative programming tells what to do. Imperative programming tells how to do it.*

Nell'ambito della programmazione imperativa, vengono normalmente identificati due paradigmi principali:

1. Paradigma **procedurale**: in cui i comandi (o, in modo improprio, **statement**) da svolgere vengono forniti ad un esecutore raggruppandoli in **procedure**. Ogni procedura manipola lo stato dell'intera computazione in modo esplicito.
2. Paradigma **object-oriented**: in cui sono presenti più esecutori detti **oggetti** che interagiscono mediante lo scambio di **messaggi**. I comandi da svolgere da parte di ogni esecutore vengono raggruppati in procedure che vengono eseguite a seguito della ricezione dei messaggi. Normalmente, ogni procedura manipola solo lo stato dell'esecutore che la esegue e non lo stato dell'intera computazione.

Nell’ambito della programmazione dichiarativa, vengono normalmente identificati due paradigmi principali:

1. Paradigma **funzionale**: in cui il problema da risolvere viene descritto mediante un insieme di oggetti e un insieme di **funzioni** tra gli oggetti (relazioni per cui ogni elemento del dominio viene associato ad un unico elemento del codominio) e un esecutore è in grado di ragionare sulle funzioni e sulla loro composizione per risolvere il problema.
2. Paradigma **logico**: in cui il problema da risolvere viene descritto mediante un insieme di oggetti e un insieme di relazioni tra gli oggetti e un esecutore è in grado di ragionare sulle relazioni per risolvere il problema.

La programmazione funzionale viene usata molto spesso nell’intelligenza artificiale, specialmente nella tradizione statunitense. I linguaggi di programmazione funzionale più tradizionali sono *Lisp* (da *LISt Processor*) e i suoi dialetti e derivati (ad esempio *Scheme*). Il linguaggio di programmazione funzionale oggi più utilizzato è sicuramente *Haskell* (www.haskell.org).

La programmazione logica viene usata molto spesso nell’intelligenza artificiale, specialmente nella tradizione europea e giapponese. I linguaggi di programmazione logica più tradizionali sono *Prolog* (da *PROgrammation en LOGique*) e i suoi dialetti e derivati. Il linguaggio di programmazione logica oggi più utilizzato è ancora Prolog, eventualmente arricchito dalle funzionalità della *programmazione logica con vincoli* (*CLP*, da *Constraint Logic Programming*), e l’implementazione più diffusa di Prolog è SWI-Prolog (www.swi-prolog.org).

Dal punto di vista dei costrutti linguistici messi a disposizione dal linguaggio, la differenza principale tra un linguaggio che segue il paradigma imperativo e un linguaggio che segue il paradigma dichiarativo è l’assenza in quest’ultimo dell’**assegnazione (o assegnamento) distruttiva**. L’assegnazione distruttiva viene utilizzata nel paradigma imperativo per consentire al programma di modificare esplicitamente lo stato della computazione. L’assenza dell’assegnazione distruttiva limita la possibilità di realizzare le comuni strutture di controllo della programmazione imperativa e, di fatto, rende la ricorsione uno strumento imprescindibile della programmazione funzionale e della programmazione logica.

Si noti che molti linguaggi seguono più di un paradigma e quindi vengono detti multi-paradigma. Ad esempio, il linguaggio *Kotlin* (www.kotlinlang.org) fonde in modo organico la programmazione object-oriented di Java con la programmazione funzionale ispirata ad Haskell.

Infine, si noti che il paradigma di programmazione seguito da un linguaggio non ne limita in alcun modo la potenza. Infatti, tutti i linguaggi menzionati sono **Turing equivalenti (o completi)** in quanto sono in grado di istruire i relativi esecutori a calcolare una qualsiasi funzione (Turing) computabile.

2 Il Linguaggio dei Termini

I linguaggi di programmazione dichiarativa sono spesso caratterizzati dall’uso di tipi dato strutturati che permettono la cosiddetta **programmazione simbolica** e che, quindi, rappresentano una base solida per l’intelligenza artificiale **simbolica**, che raccoglie gli approcci all’intelligenza artificiale che prevedono la manipolazione di simboli come strumento a supporto del ragionamento umano o razionale. Il linguaggio dei termini introdotto nel seguito è l’esempio più classico di tipo di dato introdotto esplicitamente per supportare la programmazione simbolica.

Si considerino due insiemi di simboli A e V non entrambi vuoti e tra loro disgiunti e, quindi, $(A \neq \emptyset \vee V \neq \emptyset) \wedge A \cap V = \emptyset$. L'insieme A viene detto insieme degli **atomi** (o dei **simboli atomici**) e l'insieme V viene detto insieme delle **variabili** (o dei **simboli di variabile**). L'insieme T dei **termini (del primo ordine)** ottenibili da A e da V è costruito secondo le regole:

1. Una variabile $v \in V$ è un termine;
2. Un atomo $a \in A$ è un termine;
3. Se $f \in A$, $k \in \mathbb{N}_+$ e $\{t_i\}_{i=1}^k \subseteq T$, allora $f(t_1, t_2, \dots, t_k)$ è un termine; e
4. Nient'altro è un termine.

Quindi, il **linguaggio dei termini (del primo ordine)** T è il linguaggio che si appoggia all'alfabeto improprio (perché non necessariamente finito) $A \cup V$ e si basa sulla seguente grammatica non contestuale:

$$\begin{array}{lcl} \text{Term} & \rightarrow & \text{Variable} \mid \text{Atom} \mid \text{Atom}(\text{Arguments}) \\ \text{Arguments} & \rightarrow & \text{Term} \mid \text{Term}, \text{Arguments} \\ \text{Variable} & \rightarrow & \{v \in V\} \\ \text{Atom} & \rightarrow & \{a \in A\} \end{array} \quad (1)$$

Un termine t si dice **non strutturato** se è formato unicamente da un atomo o da una variabile. Viceversa, tutti gli altri termini si dicono **strutturati**. L'atomo più a sinistra di un termine strutturato viene detto **testa** del termine e, si noti, non può essere una variabile. Il numero di argomenti di un termine strutturato si dice **arità** del termine. Si noti che non è prevista un'arità fissa associata ad un atomo usato come testa di un termine strutturato. Quindi, ad esempio, in uno stesso termine un atomo può essere usato con arità diverse senza generare ambiguità. Per convenzione, l'arità di un termine non strutturato vale zero, ma spesso si evita di attribuire un'arità alle variabili.

In alcuni contesti, agli atomi non è attribuita un'arità e, quando viene fatta questa scelta, gli atomi vengono separati in due insiemi disgiunti chiamati, rispettivamente, insieme delle **costanti** (o dei **simboli di costante**) e insieme delle **funzioni** (o dei **simboli di funzione**) e questi insiemi vengono scelti disgiunti dall'insieme delle variabili.

Seguono alcuni esempi di termini (con $A = \{a, b, f, g\}$ e $V = \{x, y\}$):

- a e x sono termini.
- $f(a)$, $f(x)$, $f(a, f(x))$, $f(g(a))$, $f(g(y, b))$ sono termini.

Si noti che i termini strutturati possono essere usati per descrivere degli alberi e, spesso, vengono descritti mediante dei diagrammi. Ad esempio, Figura 1 riporta il diagramma che descrive l'albero relativo al termine $f(g(a, b), f(x, g(c, y)))$ (con $A = \{a, b, c, f, g\}$ e $V = \{x, y\}$).

Dato un termine $t \in T$, $\text{vars}(t)$ è l'insieme delle variabili presenti in t . Se $\text{vars}(t) = \emptyset$, allora t viene detto termine **ground**. Dato un insieme di atomi $A \neq \emptyset$, l'insieme dei termini ottenibili usando l'insieme di variabili $V = \emptyset$ è un insieme di termini ground e viene detto **universo di Herbrand** ottenibile da A .

Dati due termini $t_1 \in T$ e $t_2 \in T$, si dice che t_1 è **sintatticamente equivalente** a t_2 se t_1 e t_2 sono lo stesso elemento di T .

Quando si lavora con un insieme di termini è spesso utile permettere di trattare alcuni termini con varianti sintattiche speciali che ne semplificano la lettura e la scrittura oltre a

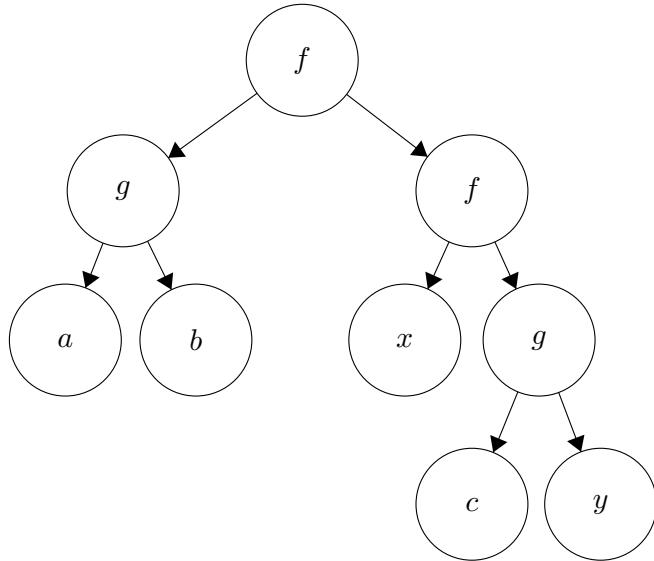


Figura 1: L’albero descritto dal termine $f(g(a, b), f(x, g(c, y)))$.

renderne più esplicito il significato. Come discusso nel seguito, queste varianti sintattiche permetteranno di scrivere termini anche complessi in modo semplice esplicitandone anche il significato. Un caso tipico di variante sintattica è quello che prevedere che l’insieme degli atomi contenga almeno due atomi:

1. Un atomo, normalmente indicato con il simbolo .. , che viene utilizzato solo per formare termini strutturati con due argomenti; e
2. Un atomo, normalmente indicato con il simbolo $[]$, che viene utilizzato solo per formare termini non strutturati.

In questo caso, si utilizza la sintassi delle **liste** per semplificare la lettura e la scrittura di termini complessi. In particolare:

1. La lista $[]$ è un termine che viene chiamato **lista vuota**;
2. La lista $[h|r]$ è il termine $.(h, r)$, dove h è un termine che viene detto **testa della lista** e r è una lista detta **resto della lista**;
3. La lista $[t_1, t_2, \dots, t_n]$ è il termine $.(t_1, .(t_2, \dots, .(t_n, []) \dots))$; e
4. La lista $[t_1, t_2, \dots, t_n|r]$ è il termine $.(t_1, .(t_2, \dots, .(t_n, r) \dots))$, dove r è una lista che, anche in questo caso, viene detta resto della lista.

Si noti però che se, ad esempio $A = \{a, b\}$, allora il termine $[a|b]$ è un termine sintatticamente corretto ed equivale a $.(a, b)$, ma questo termine non viene considerata una lista perché si prevede che il resto di una lista sia a sua volta una lista, eventualmente vuota. L’utilizzo di termini tipo $[a|b]$ è però interessante perché permette di introdurre le **ennuple**. Infatti, un termine tipo $[a|b]$ non è altro che una coppia.

L’introduzione della sintassi delle liste non aggiunge niente all’insieme dei termini su cui si sta lavorando, ma permette di scrivere in modo semplice ed esplicito alcuni termini, quali, ad esempio (con $A = \{a, b, c\}$ e $V = \{x\}\}:$

- $[a, b, c]$ è la lista formata dagli atomi a , b e c , in questo ordine.
- $[a|x]$ è una lista che inizia con l'atomo a e prosegue con un resto che viene associato alla variabile x .

Oltre alle liste, viene normalmente data la possibilità di introdurre degli operatori unari o binari per descrivere in modo semplice alcuni termini complessi. Quando viene data la possibilità di utilizzare operatori per costruire termini, viene anche data la possibilità di raggruppare gli operatori e i relativi argomenti mediante le parentesi tonde, come normalmente si fa nella scrittura delle espressioni matematiche. Seguendo una tradizione consolidata, ogni operatore viene descritto mediante le seguenti tre proprietà:

1. L'atomo da utilizzare per indicare l'operatore e quindi, ad esempio, $+$ o $-$;
2. L'indice di precedenza dell'operatore; e
3. Il tipo di operatore, descritto mediante una delle sette tipologie ammesse.

L'indice di precedenza di un operatore è un numero naturale positivo con la convenzione che al crescere dell'indice di precedenza decresce la precedenza degli operatori. Infatti, viene utilizzata la convenzione che l'indice di precedenza delle variabili e degli atomi, eventualmente utilizzati come teste di termini strutturati, vale zero. In più, vale zero anche l'indice di precedenza dei termini racchiusi tra parentesi tonde.

Le sette tipologie di operatori ammesse sono descritte dalle seguenti etichette:

1. fx , fy per gli operatori **unari prefissi**;
2. xfx , xyf , yfx per gli operatori **binari infissi**; e
3. xf , yf per gli operatori **unari postfissi**.

Le etichette utilizzate per identificare le sette tipologie possono essere lette informalmente nel seguente modo:

1. Il simbolo f viene usato per individuare la posizione dell'operatore;
2. Il simbolo x viene letto: una variabile, un atomo, eventualmente usato come testa di un termine strutturato, un termine tra parentesi tonde o un operatore con indice di precedenza strettamente inferiore all'indice di precedenza dell'operatore che si sta descrivendo; e
3. Il simbolo y viene letto: una variabile, un atomo, eventualmente usato come testa di un termine strutturato, un termine tra parentesi tonde o un operatore con indice di precedenza minore o uguale all'indice di precedenza dell'operatore che si sta descrivendo.

Seguono alcune definizioni di operatori molto comuni indicando nelle triplettre prima l'indice di precedenza, poi la tipologia e, per ultimo, l'atomo:

- $(700, xfx, <)$, $(700, xfx, = <)$, $(700, xfx, =)$, $(700, xfx, >=)$, $(700, xfx, >)$.
- $(500, yfx, +)$, $(500, yfx, -)$.
- $(400, yfx, *)$, $(400, yfx, /)$.
- $(200, fy, +)$, $(200, fy, -)$.

L'introduzione della sintassi degli operatori unari e binari non aggiunge niente all'insieme dei termini su cui si sta lavorando ma permette di scrivere in modo semplice ed esplicito alcuni termini complessi. Seguono alcuni esempi dell'uso dei precedenti operatori per la costruzione di termini (con $A = \{a, b, c, f\}$ e $V = \{x, y\}$) in cui è stata usata la convenzione comune di racchiudere un atomo tra apici per garantire che non venga interpretato come un operatore:

1. Il termine $f(x) + b$ equivale a ' $+ '(f(x), b)$;
2. Il termine $a + b - c$ equivale a ' $- ' + '(a, b), c)$;
3. Il termine $a + b * c$ equivale a ' $+ '(a, ' $* '(b, c))$;$
4. Il termine $-a + b/y$ equivale a ' $+ '(-'(a), '/'(b, y))$
5. Il termine $a + b = < c * a$ equivale a ' $= < ' + '(a, b), ' $* '(c, a))$.$

Si noti che l'introduzione contemporanea di liste e operatori viene normalmente accompagnata dalla definizione dell'operatore `.` come operatore unario postfisso con basso indice di precedenza (normalmente, 100). In questo caso, l'atomo `.` utilizzato per le liste deve essere racchiuso tra apici per evitare che venga interpretato come operatore unario postfisso.

Si noti che le informazioni fornite per descrivere gli operatori consentono di identificare in modo univoco a quale termine si sta facendo riferimento utilizzando gli operatori, purché non vengano introdotti degli errori sintattici. L'introduzione degli operatori aumenta la possibilità di introdurre errori sintattici perché le sette tipologie di operatori vincolano i modi in cui gli operatori possono essere combinati. Ad esempio, i seguenti sono alcuni esempi di errori sintattici dovuti all'introduzione degli operatori per descrivere termini:

1. Il termine $a + b >=$ utilizza l'operatore `>=` come unario postfisso anziché binario;
2. Il termine $a = < b = < c$ utilizza l'operatore `= <` con un secondo operatore con lo stesso indice di precedenza a sinistra (o a destra); e
3. Il termine $(a + b$ non ha le parentesi bilanciate.

Infine, conviene notare che anche se l'introduzione di operatori per usi specifici può semplificare la manipolazione di termini complessi, non conviene ricorrere troppo spesso a questi operatori.

3 Unificazione di Termini

Dato un insieme di termini T costruito su un insieme di atomi A e un insieme di variabili V , una **sostituzione** è un insieme finito ed eventualmente vuoto della forma

$$\{t_1/x_1, t_2/x_2, \dots, t_n/x_n\} \quad (2)$$

dove

1. x_1, x_2, \dots, x_n sono variabili;
2. t_1, t_2, \dots, t_n sono termini;
3. Per ogni $1 \leq i \leq n$, $t_i \neq x_i$; e
4. Per ogni $1 \leq i \leq n$ e $1 \leq j \leq n$, se $i \neq j$ allora $x_i \neq x_j$.

Se t è un termine e θ è una sostituzione, allora $t\theta$ è il termine che si ottiene sostituendo in t *simultaneamente* tutte le variabili nelle parti destre degli elementi della sostituzione θ con i rispettivi termini.

Ad esempio, se $\theta = \{f(z, z)/x, c/z\}$ (con $A = \{c, f\}$ e $V = \{x, z\}$) allora

$$p(f(x, y), x, g(z))\theta = p(f(f(z, z), y), f(z, z), g(c)) \quad (3)$$

Date due sostituzioni $\theta = \{t_1/x_1, t_2/x_2, \dots, t_n/x_n\}$ e $\sigma = \{u_1/y_1, u_2/y_2, \dots, u_m/y_m\}$ la **sostituzione composta** $\theta \circ \sigma$ si ottiene dal seguente insieme

$$\{t_1\sigma/x_1, t_2\sigma/x_2, \dots, t_n\sigma/x_n, u_1/y_1, u_2/y_2, \dots, u_m/y_m\} \quad (4)$$

eliminando tutti gli elementi

1. $t_j\sigma/x_j$ se vale $t_j\sigma = x_j$; e
2. u_j/y_j se $y_j \in \{x_1, x_2, \dots, x_n\}$.

Quindi, ad esempio, se $\theta = \{f(y)/x, z/y\}$ e $\sigma = \{a/x, b/y, y/z\}$ (con $A = \{a, b, f\}$ e $V = \{x, y, z\}$) allora

$$\theta \circ \sigma = \{f(b)/x, y/z\} \subset \{f(b)/x, y/y, a/x, b/y, y/z\} \quad (5)$$

La proprietà fondamentale della composizione di sostituzioni, che ne giustifica il nome, è che per ogni coppia di sostituzioni θ, σ vale

$$t(\theta \circ \sigma) = (t\theta)\sigma \quad (6)$$

per ogni termine t . Quindi, ad esempio, se $\theta = \{f(y)/x, z/y\}$, $\sigma = \{a/x, b/y, y/z\}$ e $t = h(x, g(y), z)$ (con $A = \{a, b, f, g, h\}$ e $V = \{x, y, z\}$), allora

$$t\theta = h(f(y), g(z), z) \quad (t\theta)\sigma = h(f(b), g(y), y) \quad (7)$$

e, infatti, $t(\theta \circ \sigma) = h(f(b), g(y), y)$.

Una sostituzione θ si dice **unificatore** per l'insieme di $k \geq 2$ termini $S = \{t_i\}_{i=1}^k$ se vale la seguente catena di uguaglianze

$$t_1\theta = t_2\theta = \dots = t_k\theta \quad (8)$$

Un insieme di termini si dice **unificabile** se esiste almeno un unificatore per l'insieme. Si noti che un insieme di termini può ammettere più unificatori, ad esempio, perché alcuni unificatori utilizzano più variabili del necessario. L'insieme di due termini $\{p(x), p(y)\}$ (con $A = \{p\}$ e $V = \{x, y, z\}$) ammette almeno tre unificatori:

$$\{x/y\} \quad \{y/x\} \quad \{z/x, z/y\} \quad (9)$$

dove i primi due unificatori possono essere resi uguali semplicemente cambiando in modo coerente i nomi delle variabili e il terzo unificatore è stato ottenuto mediante l'introduzione della variabile, non necessaria, z .

Viceversa, se un insieme di termini S è unificabile allora ammette un unico **unificatore più generale** (*Most General Unifier, MGU*), a meno di cambiamenti coerenti dei nomi delle variabili. Il MGU di un insieme di termini S è definito come un unificatore $mgu(S)$ tale che qualsiasi sia un unificatore θ di S esiste almeno una sostituzione σ tale che $\theta = mgu(S) \circ \sigma$. Quindi, a giustificazione del nome, il MGU di un insieme di termini non compie

sostituzioni non necessarie ma compie solo le sostituzioni necessarie a rendere gli elementi dell'insieme congiuntamente uguali. Dei tre unificatori elencati nell'esempio precedente, solo i primi due sono MGU e, infatti, sono uguali a meno di cambiamenti coerenti dei nomi delle variabili.

Ad esempio, se $S = \{p(x), p(f(y))\}$ allora $\theta = \{f(a)/x, a/y\}$ (con $A = \{a, f, p\}$ e $V = \{x, y\}$) è un unificatore di S , infatti

$$p(x)\theta = p(f(a)) \quad p(f(y))\theta = p(f(a)) \quad (10)$$

ma θ non è il MGU di S perché, per unificare gli elementi di S , è sufficiente $\alpha = \{f(y)/x\}$ e, infatti, $\theta = \alpha \circ \{a/y\}$.

Dato un insieme di termini T costruito su un insieme di atomi A e un insieme di variabili V , un **problema di unificazione (sintattica)** è un insieme del tipo

$$\{l_1 \doteq r_1, l_2 \doteq r_2, \dots, l_n \doteq r_n\} \quad (11)$$

dove $\{l_i\}_{i=1}^n \subseteq T$ e $\{r_i\}_{i=1}^n \subseteq T$ e ogni elemento del problema viene detto **equazione**. Dato un problema di unificazione, si cerca una sostituzione θ tale che valga $l_i\theta = r_i\theta$ per ogni $1 \leq i \leq n$. Se una tale sostituzione esiste, allora il problema viene detto risolubile e la sostituzione trovata ne è una soluzione.

Si noti che vengono definite le seguenti funzioni per un problema di unificazione S , generalizzando opportunamente le relative funzioni definite per i termini:

1. $vars(S)$ è l'insieme delle variabili contenute nelle parti sinistre e nelle parti destre delle equazioni di S ; e
2. Se θ è una sostituzione, $S\theta$ è il problema di unificazione ottenuto applicando la sostituzione alle parti sinistre e alle parti destre di tutte le equazioni in S .

Il seguente algoritmo è in grado di stabilire se un problema di unificazione è risolubile e, in caso, di trovarne una soluzione. L'algoritmo (**di unificazione di Martelli e Montanari**) parte da un problema di unificazione S e termina producendo uno dei seguenti risultati:

1. Il simbolo \perp se il problema di unificazione non è risolubile; oppure
2. Un problema di unificazione equivalente ad S ma con solo variabili nelle parti sinistre delle equazioni, e quindi nella forma $\{x_1 \doteq t_1, x_2 \doteq t_2, \dots, x_m \doteq t_m\}$, da cui è possibile costruire una soluzione $\{t_1/x_1, t_2/x_2, \dots, t_m/x_m\}$.

L'algoritmo può essere descritto come la seguente **funzione non deterministica** $unify$:

1. $unify(G \cup \{f(l_1, l_2, \dots, l_m) \doteq f(r_1, r_2, \dots, r_m)\}) = unify(G \cup \{l_1 \doteq r_1, l_2 \doteq r_2, \dots, l_m \doteq r_m\})$ (caso *decompose*);
2. $unify(G \cup \{x \doteq t\}) = unify(G\{t/x\} \cup \{x \doteq t\})$ se $x \in vars(G)$ e $x \notin vars(t)$ (caso *eliminate*);
3. $unify(G \cup \{f(l_1, l_2, \dots, l_m) \doteq x\}) = unify(G \cup \{x \doteq f(l_1, l_2, \dots, l_m)\})$ se $x \in V$ (caso *swap*);
4. $unify(G \cup \{t \doteq t\}) = unify(G)$ (caso *delete*);
5. $unify(G \cup \{f(l_1, l_2, \dots, l_m) \doteq g(r_1, r_2, \dots, r_k)\}) = \perp$ se $f \neq g \vee m \neq k$ (caso *conflict*);
6. $unify(G \cup \{x \doteq f(t_1, t_2, \dots, t_m)\}) = \perp$ se $x \in vars(f(t_1, t_2, \dots, t_m))$ (caso *check*).

Si noti che l'algoritmo è utilizzabile per calcolare il MGU di due termini l e r tra loro unificabili. Infatti, l'applicazione dell'algoritmo al problema di unificazione $\{l \doteq r\}$ consente di trovare immediatamente $mgu(\{l, r\})$.

In più, si noti che il caso check aumenta la complessità computazionale temporale asintotica di caso pessimo dell'algoritmo e quindi, alle volte, si preferisce omettere questo caso per ottenere una procedura di unificazione *senza occurs check*. In questo caso è anche necessario rimuovere le restrizioni dal caso eliminato per garantire di potere elaborare problemi tipo $\{x \doteq f(x)\}$ e, quindi, non sarà più possibile garantire la terminazione della procedura (e non più algoritmo) di unificazione.

Infine, si noti che i problemi di unificazione possono essere generalizzati a **problemi di unificazione modulo teoria** se la relazione \doteq ammette delle proprietà aggiuntive fornite dalla teoria considerata oltre ad essere una relazione di equivalenza tra termini. In questi casi, l'algoritmo visto deve essere opportunamente esteso per tenere conto delle proprietà fornite dalla teoria considerata. Ad esempio, se l'atomo f denota un'operazione commutativa della teoria considerata, allora il problema di unificazione $\{f(a, b) \doteq f(x, y)\}$ dovrà ammettere due soluzioni $\{a/x, b/y\}$ oppure $\{b/x, a/y\}$. Quindi, non solo si dovrà modificare l'algoritmo di unificazione per tenere conto della commutatività dell'operazione denotata da f ma si dovrà anche ammettere la possibilità che due termini unificabili possano avere più MGU tra loro effettivamente diversi.

4 Sintassi di Prolog₀

Si consideri un insieme di atomi A e un insieme di variabili V tali che:

1. A è formato da tutte e sole le parole che iniziano con una lettera minuscola e proseguono con, eventualmente zero, lettere, numeri e underscore; e
2. V è formato da tutte e sole le parole che iniziano con una lettera maiuscola o un underscore e proseguono con, eventualmente zero, lettere, numeri e underscore senza però essere formate unicamente da underscore.

Si noti che $A \neq \emptyset$, $V \neq \emptyset$, $A \cap V = \emptyset$ ed entrambi gli insiemi sono infiniti numerabili. In più, si noti che non sono presenti simboli di punteggiatura nei due insiemi.

Si consideri il linguaggio *Prolog₀* descritto dalla seguente grammatica che usa l'alfabeto improprio $A \cup V$ e riprende la grammatica dei termini nella parte finale:

$$\begin{aligned}
 Program &\rightarrow Clauses \\
 Clauses &\rightarrow Clause \mid Clause\ Clauses \\
 Clause &\rightarrow Fact \mid Rule \\
 Fact &\rightarrow Head. \\
 Rule &\rightarrow Head \ :-\ Body. \\
 Head &\rightarrow Atom \mid Atom(Arguments) \\
 Body &\rightarrow Conjuncts \\
 Conjuncts &\rightarrow Conjunct \mid Conjunct,\ Conjuncts \\
 Conjunct &\rightarrow Atom \mid Atom(Arguments) \\
 Term &\rightarrow Variable \mid Atom \mid Atom(Arguments) \\
 Arguments &\rightarrow Term \mid Term,\ Arguments \\
 Variable &\rightarrow \{v \in V\} \\
 Atom &\rightarrow \{a \in A\}
 \end{aligned} \tag{12}$$

Il linguaggio Prolog₀ è un dialetto minimalista (ma non troppo) del linguaggio Prolog. In particolare, il linguaggio Prolog₀ isola la porzione di Prolog che realizza la programmazione logica in modo più puro.

La seguente nomenclatura è definita per Prolog₀ ma vale anche per Prolog:

1. Gli elementi derivabili dalla produzione *Program*, e quindi gli elementi π tali che $Program \xrightarrow{*} \pi$, si chiamano **programmi**.
2. Gli elementi derivabili dalla produzione *Clause*, e quindi gli elementi γ tali che $Clause \xrightarrow{*} \gamma$, si chiamano **clausole definite**.
3. Gli elementi derivabili dalla produzione *Fact*, e quindi gli elementi λ tali che $Fact \xrightarrow{*} \lambda$, si chiamano **fatti** e, per ogni fatto, è definita una **testa**.
4. Gli elementi derivabili dalla produzione *Rule*, e quindi gli elementi ρ tali che $Rule \xrightarrow{*} \rho$, si chiamano **regole** e, per ogni regola, è definita una **testa** ed è definito un **corpo**.
5. Gli elementi derivabili dalla produzione *Conjunct*, e quindi gli elementi α tali che $Conjunct \xrightarrow{*} \alpha$, si chiamano **congiunti** e ogni congiunto è un termine che non può essere una variabile isolata.

Infine, si noti che gli elementi derivabili da *Conjuncts* sono sequenze di congiunti e quindi è possibile estendere ad un'intera sequenza una qualsiasi operazione che coinvolga un congiunto semplicemente applicando l'operazione ad ogni congiunto della sequenza e separando i risultati con delle virgole. L'utilizzo delle virgole per separare i risultati consente di ottenere ancora un elemento derivabile da *Conjuncts*. In particolare, se θ è una sostituzione, e $Conjuncts \xrightarrow{*} \beta$, allora è possibile parlare di $\beta \theta$ applicando la sostituzione ad ogni congiunto di β e separando i risultati ottenuti con delle virgole.

Dato un programma scritto in Prolog₀ è possibile ottenere una computazione abbinando al programma un **goal**. Una computazione così ottenuta, se produce un risultato, produce una o più sostituzioni, eventualmente vuote, che coinvolgono un sottoinsieme delle variabili del goal. Non tutte le computazioni, però, producono un risultato perché, per alcune computazioni, il risultato non è definito mentre, per altre computazioni, non necessariamente diverse dalle precedenti, la computazione non termina.

Dal punto di vista sintattico, un goal è descritto da una regola privata della testa. Un elemento dell'insieme delle clausole definite unito all'insieme dei goal viene detto **clausola di Horn**. Si noti che i goal non possono essere utilizzati nel testo di un programma Prolog₀ mentre possono essere utilizzati nel testo di un programma Prolog con l'ovvio significato di attivare una computazione per ogni goal incontrato durante il caricamento in memoria del programma.

Ad esempio, si consideri il seguente programma *Prolog₀*:

```
m(a).
m(b).

f(c).
f(d).

c(X) :- m(X).
c(X) :- f(X).
```

Questo programma può essere utilizzato con i seguenti goal, com'è semplice verificare utilizzando SWI-Prolog o la sua interfaccia Web *SWISH* (swish.swi-prolog.org). Si

noti che, per fornire un goal all'interfaccia interattiva di SWI-Prolog o a SWISH, deve essere omesso il simbolo `:-` iniziale. In più, si noti che i risultati successivi al primo vengono ottenuti premendo ; quando si usa l'interfaccia interattiva di SWI-Prolog mentre vengono ottenuti con gli appositi pulsanti grafici quando si usa SWISH. Infine, si noti che viene stampato `true` quando il risultato della computazione è la sostituzione \emptyset e viene stampato `false` quando non è definito un risultato per la computazione.

```

:- m(a).
true

:- m(b).
true

:- m(w).
false.

:- f(X).
X=c ; X=d

:- c(X).
X=a ; X=b ; X=c ; X=d

```

La sintassi del linguaggio Prolog₀ può essere estesa sfruttando la possibilità di introdurre liste e operatori come varianti sintattiche del linguaggio dei termini, eventualmente estendendo l'insieme degli atomi in modo opportuno. Infatti, il linguaggio Prolog₀⁺ estende Prolog₀ mediante l'introduzione di liste e operatori nelle teste di fatti e regole, nei congiunti e nei termini. Ovviamente, anche i goal da utilizzare con i programmi Prolog₀⁺ potranno sfruttare liste e operatori.

Ad esempio, il seguente è un programma scritto in Prolog₀⁺ sfruttando le liste:

```

m(H, [H | X]). 
m(H, [Y | R]) :- m(H, R).

```

Il programma può essere utilizzato, ad esempio, con il goal `m(X, [a, b, c])`.

Si noti che l'introduzione di liste e operatori non estende in modo significativo il linguaggio Prolog₀ perché liste e operatori non sono altro che varianti sintattiche del linguaggio dei termini. Viceversa, le due seguenti estensioni introdotte in Prolog₀⁺ non si limitano a varianti sintattiche, ma hanno anche un aspetto semantico. La prima di queste estensioni prevede che sia sempre disponibile in Prolog₀⁺ un operatore definito dalla tripla $(700, xfx, =)$, richiedendo quindi che l'atomo `=` sia presente nell'insieme degli atomi di Prolog₀⁺. La semantica di questo operatore verrà discussa in seguito. La seconda di queste estensioni prevede che l'insieme delle variabili di Prolog₀⁺ contenga anche un elemento formato unicamente da un underscore, elemento che è esplicitamente escluso dall'insieme delle variabili di Prolog₀. Questa variabile, che viene detta **dummy** (o **muta**), ha un significato particolare in quanto si prevede che venga sostituita con una nuova variabile non presente nel programma o nel goal tutte le volte che viene incontrata.

Ad esempio, l'utilizzo della variabile dummy e dell'operatore `=` permette di riscrivere il programma dell'esempio precedente come segue:

```

m(H, [X | _]) :- X = H.
m(H, [_ | R]) :- m(H, R).

```

5 Semantica di Prolog₀

La *sintassi* di Prolog₀ è definita dalla grammatica che viene utilizzata per costruire l'insieme dei programmi partendo dall'insieme degli atomi A e dall'insieme delle variabili V . La *semantica* di Prolog₀ viene definita nel seguito esplicitando formalmente la computazione che si ottiene da un programma utilizzato insieme ad un opportuno goal.

La sintassi di Prolog₀⁺ viene definita mediante semplici regole sintattiche che permettono di rimuovere liste, operatori e variabili dummy. Quindi, la semantica di Prolog₀⁺ può essere espressa, ad eccezione che per la semantica dell'operatore $=$, sfruttando la semantica dei programmi e dei relativi goal scritti in Prolog₀ dopo la rimozione di operatori, liste e variabili dummy. Quindi, dopo aver definito in modo formale la semantica di Prolog₀ sarà sufficiente definire la semantica dell'operatore $=$ per completare la semantica di Prolog₀⁺.

Dato un programma π scritto in Prolog₀ e un goal ν , la definizione formale della semantica del programma π quando viene utilizzato per **soddisfare** il goal ν viene descritta mediante una funzione non deterministica σ^π . La funzione σ^π riceve come unico argomento il goal ν e produce, in modo non deterministico, zero o più sostituzioni che coinvolgono le variabili di ν . Se almeno una sostituzione viene calcolata, anche se vuota, allora il goal si dice **soddisfacibile**. Viceversa, il goal si dice **insoddisfacibile**. Si noti, però, che non è possibile garantire che il calcolo di $\sigma^\pi(\nu)$ termini per qualsiasi coppia programma π e goal ν e, quindi, è possibile che in alcune situazioni l'enumerazione dei risultati del calcolo di $\sigma^\pi(\nu)$ possa non essere completata.

Prima di dettagliare σ^π è però necessario introdurre alcune funzioni che verranno utilizzate nella definizione di σ^π . Dato un programma π scritto in Prolog₀, la funzione $head_P(\pi)$ produce la prima clausola definita del programma, che è sempre presente visto che la grammatica di Prolog₀ non ammette programmi vuoti. In più, la funzione $rest_P(\pi)$ produce π privato della prima clausola definita, se il programma contiene almeno due clausole definite, oppure \perp se il programma è formato unicamente da una clausola definita.

In modo simile, dato un goal ν scritto in Prolog₀, la funzione $head_G(\nu)$ produce il primo congiunto di ν , che è sempre presente visto che la grammatica delle regole di Prolog₀ non ammette regole senza corpo. In più, la funzione $rest_G(\nu)$ produce \perp se il goal è formato unicamente da un congiunto, oppure la sequenza dei congiunti di ν a cui è stato rimosso il primo congiunto, separando i congiunti con delle virgolette.

Infine, la funzione non deterministica $(.)'$, data una clausola definita, produce una nuova clausola definita, detta sua **variante fresh (o fresca)**, ottenuta applicando una sostituzione che associa ad ogni variabile della clausola definita una nuova variabile non ancora utilizzata nella computazione. Quindi, ad esempio, una variante fresh del fatto $h(X, Y, X)$. è

$$(h(X, Y, X).)' = h(X1, Y1, X1). \quad (13)$$

nell'ipotesi che le variabili $X1$ e $Y1$ non siano ancora state usate nella computazione. In modo simile, una variante fresh della regola $h(X, Y) :- f(X), g(Y, X)$. è

$$(h(X, Y) :- f(X), g(Y, X).)' = h(X1, Y1) :- f(X1), g(Y1, X1). \quad (14)$$

sempre nell'ipotesi che le variabili $X1$ e $Y1$ non siano ancora state usate nella computazione. Si noti che la funzione $(.)'$ è una funzione non deterministica perché la scelta delle nuove variabili è del tutto arbitraria, purché queste non siano ancora state utilizzate nella computazione. In più, si noti che la produzione della variante fresh di una regola richiede di memorizzare le variabili utilizzate durante la computazione. Però, per non appesantire troppo la notazione, si preferisce lasciare implicito l'insieme delle variabili utilizzate durante una computazione senza, comunque, perdere di generalità o di correttezza.

Dato un programma π scritto in Prolog e un goal ν , la funzione non deterministica σ^π può essere esplicitata come segue:

$$\sigma^\pi(\nu) = \begin{cases} \sigma_G^\pi(g, \pi) & \text{se } g = \text{head}_G(\nu) \wedge \perp = \text{rest}_G(\nu) \\ \theta \circ \sigma^\pi(:- r\theta.) & \text{se } g = \text{head}_G(\nu) \wedge r = \text{rest}_G(\nu) \wedge r \neq \perp \wedge \theta = \sigma_G^\pi(g, \pi) \end{cases}$$

Quindi, il calcolo di $\sigma^\pi(\nu)$ si riduce a due casi non deterministici disgiunti. Se il goal è formato da un unico congiunto, allora viene usata la funzione non deterministica σ_G^π descritta nel seguito. Questa funzione, se è applicabile agli specifici argomenti e se termina, produce la sostituzione cercata. La stessa funzione viene utilizzata per produrre una sostituzione nel caso in cui il goal sia formato da almeno due congiunti. In questo caso, la sostituzione ottenuta da σ_G^π viene applicata alla parte non ancora elaborata del goal per applicare poi ricorsivamente σ^π . Si noti che la natura non deterministica di σ^π deriva unicamente dalla natura non deterministica di σ_G^π perché la sintassi del linguaggio garantisce che i due casi considerati nella definizione di σ^π siano disgiunti.

La funzione σ_G^π riceve come primo argomento un congiunto e come secondo argomento un programma, cioè una sequenza di clausole definite. Se effettivamente applicabile agli argomenti e nell'ipotesi che termini, σ_G^π produce in modo non deterministico delle sostituzioni. In particolare, la funzione σ_G^π è definita come segue:

$$\sigma_G^\pi(g, \pi) = \begin{cases} \sigma_F(g, c) & \text{se } c = \text{head}_P(\pi) \wedge c = h. \\ \sigma_R^\pi(g, c) & \text{se } c = \text{head}_P(\pi) \wedge c = h :- b. \\ \sigma_G^\pi(g, r) & \text{se } r = \text{rest}_P(\pi) \wedge r \neq \perp \end{cases} \quad (15)$$

La funzione è definita da tre casi non deterministici. Il primo descrive il comportamento della funzione se la prima clausola definita è un fatto. Il secondo descrive il comportamento della funzione se la prima clausola definita è una regola. L'ultimo caso descrive il comportamento della funzione se non sono state elaborate tutte le clausole definite del programma. Quindi, i due casi più interessanti sono il primo e il secondo perché il terzo si limita a proseguire il calcolo nel tentativo di coinvolgere tutte le clausole definite del programma.

La funzione σ_F riceve come primo argomento un congiunto di un goal e come secondo argomento un fatto è definita come segue:

$$\sigma_F(g, f) = \theta \quad \text{se } f' = h'. \wedge \theta = \text{mgu}(\{g, h'\}) \wedge \theta \neq \perp \quad (16)$$

Quindi, se è possibile unificare la testa del fatto con il goal, allora il MGU ottenuto dal goal e dalla testa di una variante fresh del fatto ricevuto come argomento è il risultato cercato.

In modo simile, la funzione σ_R^π riceve come primo argomento un congiunto di un goal e come secondo argomento una regola è definita come segue:

$$\sigma_R^\pi(g, r) = \theta \circ \sigma^\pi(:- b'\theta.) \quad \text{se } r' = h' :- b'. \wedge \theta = \text{mgu}(\{g, h'\}) \wedge \theta \neq \perp \quad (17)$$

Quindi, se è possibile unificare la testa della regola con il goal, allora il MGU ottenuto dal goal e dalla testa di una variante fresh della regola ricevuta come argomento è una parte del risultato cercato. Infatti, una volta identificato il MGU, lo si applica al corpo della variante fresh della regola utilizzata in modo da ottenere un nuovo goal da soddisfare mediante l'applicazione di σ^π .

Come già chiarito, la semantica di Prolog⁺ è ottenuta mediante la semantica di Prolog₀ dopo aver rimosso liste, operatori e variabili dummy in modo puramente sintattico. Per

completare la semantica di Prolog₀⁺ è però necessario definire una semantica per l'operatore $=$, che è sempre ritenuto disponibile. La semantica di questo operatore può essere definita prevedendo che venga aggiunto un **prologo** ad ogni programma Prolog₀⁺. Questo prologo contiene la definizione del comportamento dell'operatore $=$ mediante il fatto $X = X$. che, appunto, stabilisce che l'operatore $=$ faccia riferimento all'uguaglianza sintattica.

Il seguente esempio mostra come sia possibile utilizzare Prolog per dimostrare che un goal non è soddisfacibile.

Esempio 1. Dato il programma $\pi = p(a). \ p(b).$, si vuole verificare se il goal $\nu = :- p(c)$. è soddisfacibile. In particolare,

$$\begin{aligned}\sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(c), \pi) \stackrel{1}{=} \sigma_F(p(c), p(a).) = \times \quad mgu(\{p(c), p(a)\}) = \perp \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(c), \pi) \stackrel{2}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(c), \pi) \stackrel{3}{=} \sigma_G^\pi(p(c), p(b).) \stackrel{1}{=} \sigma_F(p(c), p(b).) = \times \quad mgu(\{p(c), p(b)\}) = \perp \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(c), \pi) \stackrel{3}{=} \sigma_G^\pi(p(c), p(b).) \stackrel{2,3}{=} \times \quad \text{non applicabili} \\ \sigma^\pi(\nu) &\stackrel{2}{=} \times \quad \text{non applicabile}\end{aligned}$$

quindi il goal non è soddisfacibile.

Il seguente esempio mostra come sia possibile utilizzare Prolog₀ per trovare una sostituzione che soddisfa un goal.

Esempio 2. Dato il programma $\pi = p(a). \ p(b).$, si vuole verificare se il goal $\nu = :- p(b)$. è soddisfacibile. In particolare,

$$\begin{aligned}\sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(b), \pi) \stackrel{1}{=} \sigma_F(p(b), p(a).) = \times \quad mgu(\{p(b), p(a)\}) = \perp \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(b), \pi) \stackrel{2}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(b), \pi) \stackrel{3}{=} \sigma_G^\pi(p(b), p(b).) \stackrel{1}{=} \sigma_F(p(b), p(b).) = mgu(\{p(b), p(b)\}) = \emptyset\end{aligned}$$

quindi il goal è soddisfacibile perché è stata trovata una sostituzione.

Il seguente esempio mostra come sia possibile utilizzare Prolog₀ per trovare più sostituzioni che soddisfano un goal.

Esempio 3. Dato il programma $\pi = p(a). \ p(b).$, si vuole verificare se il goal $\nu = :- p(X)$. è soddisfacibile. In particolare,

$$\sigma^\pi(\nu) \stackrel{1}{=} \sigma_G^\pi(p(X), \pi) \stackrel{1}{=} \sigma_F(p(X), p(a).) = mgu(\{p(X), p(a)\}) = \{a/X\} \quad (18)$$

quindi il goal è soddisfacibile. Continuando con l'applicazione non deterministica di σ^π si ottiene:

$$\begin{aligned}\sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(X), \pi) \stackrel{2}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(p(X), \pi) \stackrel{3}{=} \sigma_G(p(X), p(b).) \stackrel{1}{=} \sigma_F(p(X), p(b).) = \{b/X\}\end{aligned}$$

e quindi il goal è soddisfacibile anche dalla sostituzione $\{b/X\}$.

Il seguente esempio mostra come sia possibile utilizzare Prolog₀ per trovare una sostituzione che soddisfa un goal.

Esempio 4. Dato il programma $\pi = p(a). q(X) :- p(X).$, si vuole verificare se il goal $\nu = :- q(X)$. è soddisfacibile. In particolare,

$$\begin{aligned}\sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{1}{=} \sigma_F^\pi(q(X), p(a).) = \times \quad mgu(\{q(X), p(a)\}) = \perp \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{2}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{3}{=} \sigma_G^\pi(q(X), q(X) :- p(X).) \stackrel{1}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{3}{=} \sigma_G^\pi(q(X), q(X) :- p(X).) \stackrel{2}{=} \sigma_R^\pi(q(X), q(X) :- p(X).)\end{aligned}$$

ma

$$\sigma_R^\pi(q(X), q(X) :- p(X).) = \{X/X1\} \circ \sigma^\pi(:- p(X1)\{X/X1\}.) = \{X/X1\} \circ \sigma^\pi(:- p(X).)$$

e

$$\sigma^\pi(:- p(X).) \stackrel{1}{=} \sigma_G^\pi(p(X), \pi) \stackrel{1}{=} \sigma_F(p(X), p(a).) = mgu(\{p(X), p(a)\}) = \{a/X\}$$

e quindi il goal è soddisfacibile perché è stata trovata la sostituzione $\{a/X1, a/X\}$.

Il seguente esempio mostra come Prolog₀ possa essere utilizzato per generare computazioni che non terminano.

Esempio 5. Dato il programma $\pi = q(X) :- q(X).$, si vuole verificare se il goal $\nu = :- q(X)$. è soddisfacibile. In particolare,

$$\begin{aligned}\sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{1}{=} \times \quad \text{non applicabile} \\ \sigma^\pi(\nu) &\stackrel{1}{=} \sigma_G^\pi(q(X), \pi) \stackrel{2}{=} \sigma_R^\pi(q(X), q(X) :- q(X).)\end{aligned}$$

ma

$$\sigma_R^\pi(q(X), q(X) :- q(X).) = \{X/X1\} \circ \sigma^\pi(:- q(X1)\{X/X1\}.) = \{X/X1\} \circ \sigma^\pi(:- q(X).)$$

e quindi è dimostrato che la computazione di σ^π non termina perché viene richiesto di soddisfare ciclicamente il goal $:- q(X)$. indefinitamente.

6 Il Linguaggio Prolog_!

Il linguaggio Prolog₀⁺, e quindi il linguaggio Prolog₀, è sufficiente per realizzare programmi che possono essere utilizzati per risolvere problemi interessanti. Ad esempio, Prolog₀⁺ è stato utilizzato per realizzare un programma per descrivere un grafo diretto aciclico che è poi stato usato per soddisfare vari goal interessanti. Però, Prolog₀⁺ non è ancora sufficiente perché non offre alcun modo per bloccare le scelte non deterministiche. Quindi, un programma Prolog₀⁺ è costretto ad esplorare tutte le scelte non deterministiche disponibili anche se non è necessario farlo oppure se per farlo è necessario attivare una computazione che non termina. Questa necessità è spesso considerata troppo vincolante perché, alle volte,

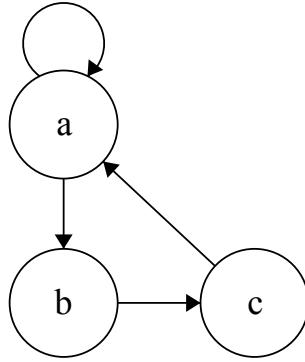


Figura 2: Un semplice grafo diretto con un autoanello e un ciclo.

è necessario poter controllare in modo dettagliato quali scelte non deterministiche attivare e quando attivarle, eventualmente facendo queste scelte in modo dinamico e dipendente dallo stato della computazione.

Si consideri, ad esempio, il grafo diretto (ciclico) riportato in Figura 2. Il grafo può essere descritto in un programma Prolog₀⁺ utilizzando due predicati, come segue:

```

arc(a, a).
arc(a, b).
arc(b, c).
arc(c, a).

path(X, X, [X]).
path(X, Y, [X | R]) :- arc(X, Z), path(Z, Y, R).

```

dove le regole che descrivono `path` sono state introdotte per calcolare il percorso tra due nodi. Quindi, il seguente goal può essere utilizzato per elencare i percorsi nel grafo tra il nodo `a` e il nodo `c`:

```
path(a, c, P).
```

Però, la soluzione proposta presenta due problemi:

1. La presenza dell'autoanello sul nodo `a` innesca immediatamente un ciclo infinito che non permette di ottenere alcune soluzioni al goal; e
2. Anche dopo aver rimosso l'autoanello, ad esempio rimuovendo o commentando il primo fatto, il goal viene soddisfatto infinite volte nonostante i percorsi prodotti siano solo ripetizioni cicliche dello stesso percorso `[a, b, c]`.

Entrambi questi problemi potrebbero essere risolti richiedendo che un percorso non possa contenere due volte lo stesso nodo. Quindi, in presenza di un ipotetico operatore di negazione, sarebbe sufficiente verificare che un nodo non sia presente nel percorso corrente e, solo in questo caso, aggiungerlo.

Però, la negazione è uno strumento complesso da realizzare nell'ambito della programmazione logica soprattutto a causa della presenza di **variabili libere**, che sono variabili non ancora sostituite da termini durante la computazione. Quindi, anziché estendere Prolog₀⁺ con una negazione complessa da realizzare e da utilizzare, è più utile estendere Prolog₀ con uno strumento in grado di inibire alcuni rami della computazione non deterministica.

Il linguaggio Prolog_! è il linguaggio Prolog₀ arricchito di un atomo che può essere utilizzato come un congiunto e, quindi, nei goal e nei corpi delle regole. Questo operatore viene indicato con il simbolo ! (letto **cut** o **simbolo di cut**) e viene usato solo con aritÀ zero. Quindi, la produzione che descrive i congiunti in Prolog_! diventa:

$$\text{Conjunct} \rightarrow \text{Atom} | \text{Atom}(Arguments) | ! \quad (19)$$

Lo strumento del cut viene spesso criticato perché è in grado di introdurre dei comportamenti inattesi se non viene usato con estrema cautela. Quindi, conviene subito sottolineare il fatto che l'uso del cut deve essere limitato allo stretto necessario.

Informalmente, quando viene incontrato un cut nel corpo di una regola $h :- b$. vengono scartate:

1. Tutte le scelte non deterministiche attivate da quando l'esecutore ha iniziato a soddisfare la regola e, quindi, tutte le scelte non deterministiche legate ai congiunti di b che precedono il cut; e
2. Tutte le scelte non deterministiche attivate per cercare un fatto o una regola con testa che unifichi h e, quindi, tutte le scelte non deterministiche legate ai fatti e alle regole che seguono la regola corrente nel testo del programma e che hanno teste che unificano con h .

Si noti che queste scelte non deterministiche vengono scartate anche se la regola in cui è presente il cut fallisce. Quindi, contrariamente agli altri effetti legati al soddisfacimento dei goal, gli effetti di un cut non vengono inibiti da un fallimento.

Ad esempio, il cut presente nel seguente programma:

```
p(a).
p(X) :- q(X), !, r(X).
p(b).
p(c).

q(y).
q(z).

r(y).
r(z).
```

garantisce che, una volta incontrato, vengano scartate le scelte non deterministiche legate a $q(X)$, $p(b)$ e $p(c)$. Quindi, il goal $:- p(W)$. viene soddisfatto unicamente dalle sostituzioni $W=a$ e $W=y$ perché $q(z)$, $p(b)$ e $p(c)$ vengono inibite dal cut.

In modo del tutto analogo, quando viene incontrato un cut in un goal, vengono scartate tutte le scelte non deterministiche attivate da quando l'esecutore ha iniziato a soddisfare il goal e, quindi, tutte le scelte non deterministiche legate ai congiunti del goal che precedono il cut. Quindi, il goal $:- p(W)$, !. applicato al programma precedente viene soddisfatto unicamente dalla sostituzione $W=a$ perché la sostituzione $W=y$ viene inibita dal cut.

Normalmente, l'introduzione del cut viene accompagnata dai seguenti fatti e regole che vengono aggiunte al prologo utilizzato implicitamente con i programmi scritti in Prolog⁺:

```
true.
```

```
fail :- a = b.
```

Il primo fatto introduce **true**, che serve per esprimere dei goal che non falliscono mai. La seconda regola serve per introdurre **fail**, che è un goal che fallisce sempre. Usando **fail** è possibile definire in modo semplice il seguente predicato:

```
X \= X :- !, fail.  
X \= Y.
```

Questo è un predicato binario, espresso mediante un operatore infisso, che serve per definire cosa significhi diverso in Prolog⁺ dicendo che due termini sono diversi tra loro solo se non possono essere unificati. Quindi, oltre a variabili dummy, liste e operatori, Prolog⁺ aggiunge a Prolog! gli operatori = e \=, il fatto **true** e le regole per definire **fail**.

L'introduzione del cut permette di risolvere i problemi dell'esempio discusso in precedenza e relativo ad un grafo diretto non aciclico. In particolare, con la seguente soluzione è possibile impedire che un percorso contenga più volte lo stesso nodo:

```
node(a).  
node(b).  
node(c).  
  
arc(a, a).  
arc(a, b).  
arc(b, c).  
arc(c, a).  
  
path(X, Y, P) :-  
    path1(X, Y, [], P).  
  
path1(_, X, V, _) :-  
    member(X, V),  
    !,  
    fail.  
path1(X, X, V, [X | V]).  
path1(Y, X, V, R) :-  
    arc(Z, X),  
    path1(Y, Z, [X | V], R).
```

L'introduzione del cut in Prolog! e, equivalentemente nel linguaggio esteso Prolog⁺, permette di modificare la computazione svolta dall'esecutore e, quindi, in generale, cambia l'insieme delle sostituzioni che risolvono un goal. Però, alle volte, l'utilizzo del cut non ha effetto sull'insieme delle sostituzioni che soddisfano un goal, ma si limita, eventualmente, solo a cambiare l'ordine con cui le soluzioni al goal vengono calcolate o il numero di volte in cui una soluzione viene prodotta. Dato un programma e un goal, se il cut viene utilizzato per modificare il programma o il goal senza però modificare l'insieme delle soluzioni al goal, allora viene detto *green cut*. Viceversa, in tutti gli altri casi, si parla di *red cut*. I green cut vengono utilizzati solo per migliorare le prestazioni dell'esecutore nel risolvere alcuni goal e, quindi, non cambiano in modo sostanziale la semantica del programma e del goal. Viceversa, i red cut vengono utilizzati per ottenere programmi o goal che si comportino in modo sostanzialmente diverso da quelli che non usano il cut. Normalmente, i green cut vengono considerati utili e non particolarmente pericolosi dal punto di vista della correttezza dei programmi realizzati. Viceversa, i red cut sono considerati uno strumento che andrebbe utilizzato poco e con estrema cautela.

L'introduzione del cut permette di spiegare perché la negazione è ritenuta problematica nella programmazione logica. Si consideri, ad esempio, il problema di stabilire se un elemento non è membro di una lista. Una possibile soluzione scritta in Prolog_! è la seguente:

```
nonmember(_, []).
nonmember(X, [X | _]) :- !, fail.
nonmember(X, [_ | R]) :- nonmember(X, R).
```

Però, questa soluzione, come tutte le soluzioni che coinvolgono la negazione, ha dei comportamenti anomali in presenza di variabili libere. Ad esempio:

```
: - nonmember(R, [a, b, c]).  
false  
  
: - nonmember(z, [a, b, R]).  
false
```

Si noti che non è possibile ovviare a questi problemi anche utilizzando il linguaggio Prolog. Infatti, la negazione offerta da Prolog viene detta **negazione per fallimento** e viene realizzata soddisfacendo un goal negato, per fallimento, solo quando il relativo goal affermato non è soddisfacibile. Quindi, l'utilizzo opportuno del cut consente sempre di evitare l'utilizzo della negazione per fallimento ed è preferibile perché le situazioni anomale introdotte dal cut risultano più evidenti.

La semantica di Prolog_! può essere introdotta formalmente andando ad estendere la semantica di Prolog₀. L'estensione richiede di definire la funzione non deterministica σ^π che definisce la semantica di Prolog_! estendendo quella di Prolog₀ per produrre come risultato una coppia (θ, γ) in cui θ è la sostituzione che rappresenta il risultato della computazione mentre γ viene utilizzata per gestire il cut. In particolare, γ può valere $!$ o \perp e, se vale, $!$ significa che la computazione è stata alterata dall'utilizzo di un cut, mentre se vale \perp significa che la computazione non ha coinvolto alcun cut. Naturalmente, questo richiede di modificare anche la funzione σ_G che definisce il comportamento dell'esecutore durante il soddisfacimento dei goal. Questa funzione estende quella di Prolog₀ prevedendo la presenza di un nuovo tipo di congiunto. Questo nuovo tipo di congiunto, il cut, non fallisce mai, non produce una sostituzione ma solo un valore per γ e la funzione è fatta in modo che gli effetti della presenza di un cut si propaghino anche se i goal che seguono il cut falliscono. Quest'ultima parte è particolarmente insidiosa e complica la funzione in modo sensibile, ma è necessaria per permettere di realizzare, ad esempio, i predicati $\backslash=$ e **nonmember**. Infatti, entrambi questi predicati utilizzano il goal $!, \text{fail}$ per garantire che non venga preso un percorso non deterministico in cui viene richiesto che i goal falliscano. Lo stesso comportamento viene tenuto dal predicato **path** dell'esempio precedente per garantire che, se un nodo è già presente in un percorso, la costruzione del percorso si interrompa.

La funzione non deterministica σ^π che definisce la semantica di Prolog_! estendendo quella di Prolog₀ produce una coppia (θ, γ) in cui θ è la sostituzione che rappresenta il risultato della computazione mentre γ viene utilizzata per gestire il cut. In particolare, γ può valere $!$ o \perp e, se vale, $!$ significa che la computazione è stata alterata dall'utilizzo del cut, mentre se vale \perp significa che la computazione non ha coinvolto il cut. Dati due valori γ_1 e γ_2 che seguono le convenzioni precedenti, viene definito:

$$\gamma_1 \oplus \gamma_2 = \begin{cases} ! & \text{se } \gamma_1 = ! \vee \gamma_2 = ! \\ \perp & \text{altrimenti} \end{cases} \quad (20)$$

In più, per definire σ^π , anche le funzioni non deterministiche σ_G^π , σ_F e σ_R^π vengono modificate in modo che producano una coppia (θ, γ) con le stesse convenzioni adottate per σ^π . Infine, anche le funzioni $head_G$ e $rest_G$ vengono modificate in modo che il cut possa essere presente in un goal.

In particolare, dato un programma π scritto in Prolog! e un goal ν , la funzione σ^π è definita come segue:

$$\sigma^\pi(\nu) = \begin{cases} \sigma_!^\pi(g, \pi) & \text{se } g = head_G(\nu) \wedge \perp = rest_G(\nu) \\ (\theta_g \circ \theta_r, \gamma_g \oplus \gamma_r) & \text{se } g = head_G(\nu) \wedge r = rest_G(\nu) \wedge r \neq \perp \wedge \\ & (\theta_g, \gamma_g) = \sigma_!^\pi(g, \pi) \wedge (\theta_r, \gamma_r) = \sigma^\pi(\cdot : r \theta_g) \end{cases} \quad (21)$$

in cui tutte le scelte non deterministiche vengono scartate se $\gamma_g \oplus \gamma_r = !$ nel secondo caso. Questa possibilità permette di scartare tutte le scelte non deterministiche attivate per soddisfare un goal nel caso in cui il congiunto corrente o i congiunti successivi a quello corrente contengano un cut.

La funzione (deterministica) $\sigma_!^\pi$ riceve come primo argomento un congiunto e come secondo argomento un programma, cioè una sequenza di clausole definite, ed è definita come segue:

$$\sigma_!^\pi(g, \pi) = \begin{cases} (\emptyset, !) & \text{se } g = ! \\ \sigma_G^\pi(g, \pi) & \text{altrimenti} \end{cases} \quad (22)$$

La funzione σ_G^π riceve come primo argomento un congiunto e come secondo argomento un programma, cioè una sequenza di clausole definite. Se effettivamente applicabile agli argomenti e nell'ipotesi che termini, σ_G^π produce in modo non deterministico delle coppie (θ, γ) utilizzando le convenzioni adottate per σ^π . In particolare, la funzione σ_G^π è definita come segue:

$$\sigma_G^\pi(g, \pi) = \begin{cases} \sigma_F(g, c) & \text{se } c = head_P(\pi) \wedge c = h. \\ (\theta_r, \gamma_r) & \text{se } c = head_P(\pi) \wedge c = h : b. \wedge (\theta_r, \gamma_r) = \sigma_R^\pi(g, c) \\ \sigma_G^\pi(g, r) & \text{se } r = rest_P(\pi) \wedge r \neq \perp \end{cases} \quad (23)$$

in cui tutte le scelte non deterministiche vengono scartate se viene scelto il primo caso o se $\gamma_r = !$ nel terzo caso. Questa possibilità consente di scartare tutte le scelte non deterministiche che potrebbero essere attivate proseguendo con il resto del programma nel tentativo di soddisfare il goal.

La funzione σ_F riceve come primo argomento un congiunto di un goal e come secondo argomento un fatto ed è definita come segue:

$$\sigma_F(g, f) = (\theta, \perp) \quad \text{se } f' = h'. \wedge \theta = mgu(\{g, h'\}) \wedge \theta \neq \perp \quad (24)$$

Quindi, se è possibile unificare la testa del fatto con il goal, allora il MGU ottenuto dal goal e dalla testa di una variante fresh del fatto è il risultato cercato. Si noti che la funzione σ_F non coinvolge mai un cut perché un cut può essere utilizzato solo nel corpo di una regola.

In modo simile, la funzione σ_R^π riceve come primo argomento un congiunto di un goal e come secondo argomento una regola ed è definita come segue:

$$\sigma_R^\pi(g, r) = (\theta \circ \theta_r, \gamma_r) \quad \text{se } r' = h' : b'. \wedge \theta = mgu(\{g, h'\}) \wedge \theta \neq \perp \wedge \quad (25)$$

$$(\theta_r, \gamma_r) = \sigma^\pi(\cdot : b' \theta.) \quad (26)$$

Si noti che la funzione σ_R^π non scarta alcuna scelta non deterministica a causa della presenza di un cut nel corpo della regola.

Esempio 6. Dato il programma $\pi = p(a). p(b).$, si vuole determinare per quali sostituzioni il goal $\nu = :- p(X), !.$ è soddisfatto. In particolare,

$$\sigma^\pi(\nu) \stackrel{1}{=} \times \quad \text{non applicabile} \quad (27)$$

$$\sigma^\pi(\nu) \stackrel{2}{=} (\theta_g \circ \theta_r, \gamma_g \oplus \gamma_r) \quad (28)$$

dove:

$$(\theta_g, \gamma_r) = \sigma_!^\pi(p(X), \pi) \stackrel{1}{=} \times \quad \text{non applicabile} \quad (29)$$

$$(\theta_g, \gamma_r) = \sigma_!^\pi(p(X), \pi) \stackrel{2}{=} \sigma_G^\pi(p(X), \pi) \stackrel{1}{=} \times \quad \text{non applicabile} \quad (30)$$

$$(\theta_g, \gamma_r) = \sigma_G^\pi(p(X), \pi) \stackrel{2}{=} \sigma_F(p(X), p(a).) = (\{a/X\}, \perp) \quad (31)$$

e

$$(\theta_r, \gamma_r) = \sigma^\pi(:- !.) \stackrel{1}{=} \sigma_!^\pi(:- !.) \stackrel{1}{=} \sigma_G(!, \pi) \stackrel{1}{=} (\emptyset, !) \quad (32)$$

quindi, una soluzione al goal ν è $(\{a/X\}, !).$ Questa è anche l'unica soluzione perché tutte le altre scelte non deterministiche di

$$\sigma^\pi(\nu) \stackrel{2}{=} (\theta_g \circ \theta_r, \gamma_g \oplus \gamma_r) = (\{a/X\}, !) \quad (33)$$

vengono scartate perché $\gamma_g \oplus \gamma_r = !.$

7 Programmazione Logica con Vincoli

La programmazione logica realizzata mediante Prolog₀ e le sue varianti ed estensioni può essere ulteriormente estesa per essere resa più efficace nell'affrontare problemi comuni quali, ad esempio, i problemi di calcolo. La **programmazione logica con vincoli** (**Constraint Logic Programming, CLP**) estende la programmazione logica vista finora in modo molto generale e applicabile a vari contesti. Quindi, conviene subito notare che la programmazione logica con vincoli non viene introdotta unicamente per trattare problemi di calcolo mediante la programmazione logica. Anzi, è ormai così comune che quando si parla di programmazione logica, normalmente, si intende programmazione logica con vincoli.

Per estendere Prolog₀ e renderlo un linguaggio di programmazione logica con vincoli è necessario introdurre i linguaggi di vincoli. Un **linguaggio di vincoli** è una ennupla:

$$\langle D, C, A, V, sat, post, label \rangle \quad (34)$$

dove:

1. $D \neq \emptyset$ è il **dominio** del linguaggio di vincoli;
2. $C \neq \emptyset$ è un insieme di **simboli di vincolo**;
3. $A \neq \emptyset$ e $V \neq \emptyset$, con $A \cap V = \emptyset$ sono, rispettivamente, un insieme di atomi e un insieme di variabili utilizzati per costruire termini; e
4. $sat, post$ e $label$ sono tre funzioni descritte nel seguito.

L'insieme D può essere utilizzato per costruire dei CSP che abbiano come variabili gli elementi di un sottoinsieme finito di V , ammettendo anche il caso degenere in cui il sottoinsieme scelto sia vuoto. Quindi, dato un insieme finito di variabili $V_P \subseteq V$ è possibile costruire un CSP $\langle V_P, \{D\}, C_P \rangle$ che abbia V_P come insieme di variabili, D come dominio di tutte le variabili in V_P e C_P come vincoli.

Dato uno di questi CSP, la funzione sat dello specifico linguaggio di vincoli è in grado di verificare la soddisfabilità del CSP. In particolare, il risultato dell'applicazione della funzione sat ad un CSP è \perp se la funzione è in grado di stabilire che il CSP è insoddisfacibile. Viceversa, se la funzione sat non è in grado di garantire l'insoddisfabilità, allora il risultato è \top . Quindi, se il risultato della funzione sat applicata ad un CSP è \perp , allora il CSP è sicuramente insoddisfacibile. Viceversa, se il risultato è \top , allora il CSP potrebbe ancora essere soddisfacibile.

La funzione $post$ serve per manipolare i CSP che è possibile costruire con il linguaggio di vincoli a disposizione. Infatti, dato un CSP $\langle V_P, \{D\}, C_P \rangle$, vale:

$$post(\langle V_P, \{D\}, C_P \rangle, c(t_1, t_2, \dots, t_n)) = \langle V_P \cup vars(t_1, t_2, \dots, t_n), \{D\}, C_P \cup \{\gamma\} \rangle$$

dove:

1. $c \in C$ è un simbolo di vincolo a cui è attribuita aritità $n \in \mathbb{N}_+$;
2. I termini in $\{t_i\}_{i=1}^n$ sono costruiti utilizzando A e V ; e
3. γ è un vincolo che viene aggiunto al CSP.

In sintesi, la funzione $post$ riceve come argomento un CSP e un'espressione che denota un vincolo γ e aggiunge il vincolo, insieme alle sue variabili, al CSP. Normalmente, la funzione $post$ propaga il nuovo vincolo in modo da ridurre, se possibile, i valori ammissibili per le variabili. Comunque, non viene richiesto che la $post$ propaghi i vincoli e, anche in caso lo facesse, non è previsto che la propagazione identifichi se il nuovo CSP prodotto è insoddisfacibile, anche perché non è previsto che la funzione $post$ studi la soddisfabilità dei CSP che estende. Quindi, partendo da un CSP improprio, perché senza variabili e senza vincoli, è possibile costruire nuovi CSP mediante la funzione $post$. Si noti che $post$ non è definita per tutte le espressioni che può ricevere come secondo argomento. Infatti, la funzione $post$ dello specifico linguaggio di vincoli a disposizione identifica anche quali sono le espressioni per le quali è possibile costruire nuovi CSP aggiungendo vincoli a CSP disponibili.

Infine, la funzione $label$ ha un comportamento simile alla funzione $post$, ma è non deterministica. In particolare, dato un CSP $\langle V_P, \{D\}, C_P \rangle$, vale:

$$label(\langle V_P, \{D\}, C_P \rangle, x) = \begin{cases} (\langle V_P, \{D\}, C_P \cup \{\gamma\} \rangle, t) & \text{se il CSP ottenuto è} \\ & \text{soddisfacibile per } t \\ \perp & \text{altrimenti} \end{cases} \quad (35)$$

dove:

1. $x \in V_P$ è una variabile del problema;
2. t è un termine costruibile utilizzando A e V tale che esista un unico $d \in D$ associato a t ; e
3. γ è un nuovo vincolo che impone che la variabile x valga $d \in D$, con d valore univocamente associato a t .

Quindi, la funzione *label* riceve come primo argomento un CSP e come secondo argomento una variabile del CSP. La funzione costruisce un nuovo CSP e, se è soddisfacibile, lo ritorna insieme ad un termine. In particolare, il nuovo CSP impone che la variabile valga $d \in D$, dove d è univocamente identificato dal termine t che viene ritornato insieme al CSP.

Si noti che alcuni linguaggi di vincoli forniscono una funzione *label* in grado di verificare la soddisfattibilità del nuovo CSP ottenuto almeno per un valore d , univocamente associato a t , se esiste. Quindi, contrariamente a quanto accade utilizzando *sat*, la funzione *label* di questi linguaggi consente di verificare la soddisfattibilità di un CSP andando ad assegnare un valore alla variabile x che renda il CSP soddisfacibile, se almeno un valore con questa proprietà esiste. Comunque, anche se la funzione *label* non ha questa proprietà, è previsto che la funzione *label* possa sempre stabilire la soddisfattibilità del CSP su cui lavora nel momento in cui viene utilizzata per assegnare l'ultima variabile ancora libera.

Si noti che la funzione *label* è non deterministica perché utilizza in modo non deterministico tutti i valori d , ed i corrispondenti termini t , in grado di rendere soddisfacibile il CSP ottenuto. Viceversa, per tutti i valori $\tilde{d} \in D$ per cui il corrispondente vincolo γ rende insoddisfacibile il CSP vengono scartati.

Esempio 7. Si consideri un linguaggio di vincoli con $D = \mathbb{Z}$ che metta a disposizione un termine numerico per ogni elemento di D . Sia c il vincolo definito dalla proprietà:

$$x^2 - 4 \geq 0 \quad (36)$$

Se $V_P = \{x\}$ è un sottoinsieme dell'insieme delle variabili del linguaggio di vincoli considerato, allora vale:

$$\text{label}(\langle \{x\}, \{\mathbb{Z}\}, \{c\}, x \rangle) = (P_L, t) \quad (37)$$

dove t vale, in modo non deterministico, tutti i termini numerici univocamente associati ai valori nell'insieme:

$$\{z \in \mathbb{Z} : |z| \geq 2\} = \{\pm 2, \pm 3, \dots\} \quad (38)$$

Quindi, se applicata più volte, la funzione *label* avrà come risultato t un termine tipo $2, -2, 3, -3, \dots$ ottenendo termini diversi per ogni applicazione.

Dato un linguaggio di vincoli $\mathcal{D} = \langle D, C, A, V, \text{sat}, \text{post}, \text{label} \rangle$ è possibile estendere Prolog₀ al linguaggio CLP₀(\mathcal{D}) ottenuto permettendo di utilizzare i vincoli come se fossero dei predicati. In particolare, dal punto di vista sintattico:

1. L'insieme degli atomi di CLP₀(\mathcal{D}) contiene gli atomi di Prolog₀, i simboli di vincolo in C , gli atomi in A e la parola *label*; e
2. L'insieme delle variabili di CLP₀(\mathcal{D}) contiene tutte le variabili di Prolog₀ e tutte le variabili in V .

Si noti che non è richiesto che gli insiemi che arricchiscono CLP₀(\mathcal{D}) rispetto a Prolog₀ siano disgiunti dai rispettivi insiemi di Prolog₀. Quindi, normalmente, il linguaggio CLP₀(\mathcal{D}) ha tipicamente lo stesso insieme di variabili di Prolog₀.

Dal punto di vista semantico, le funzioni che calcolano il valore semantico di un programma scritto in CLP₀(\mathcal{D}) per un particolare goal vengono arricchite di un nuovo argomento, che è un CSP costruito mediante il linguaggio di vincoli \mathcal{D} . Oltre a questo nuovo argomento, le funzioni hanno anche un nuovo risultato, che è anch'esso un CSP costruito mediante il linguaggio di vincoli \mathcal{D} . Infatti, dato un programma e un goal scritti in

$\text{CLP}_0(\mathcal{D})$, il valore semantico calcolato è una coppia in cui la prima componente è una sostituzione e la seconda componente è un CSP che esprime i vincoli accumulati durante la computazione e relativi, almeno, alle variabili della sostituzione.

In particolare, la funzione σ_G è estesa in modo che tratti i congiunti che hanno come testa un simbolo di vincolo in modo specifico. Anziché cercare calcolare il valore semantico di un congiunto di questo tipo facendo ricorso all'unificazione, come avviene in Prolog₀, vengono utilizzate le funzioni *sat* e *post*. Quindi, dato un congiunto che abbia come testa un simbolo di vincolo, viene applicata *post* al congiunto e al CSP che viene ricevuto come nuovo argomento di σ_G . Se la funzione *post* è effettivamente definita per il congiunto, allora viene verificata la soddisfabilità del risultato della *post* mediante *sat*. La funzione σ_G produce un risultato, in questo caso deterministico, solo nel caso in cui la funzione *sat* indichi che non è stata in grado di determinare che il CSP ottenuto dalla *post* è insoddisfacibile. In questo caso, il risultato prodotto dalla σ_G è la sostituzione vuota insieme al nuovo CSP.

Si noti che, siccome la funzione *sat* non è sempre in grado di determinare se un CSP è insoddisfacibile, $\text{CLP}_0(\mathcal{D})$ può produrre risultati anche per CSP insoddisfacibili. Però, è possibile garantire uno studio completo della soddisfabilità mediante la funzione *label* che associa un valore alla variabile che riceve come argomento solo quando il CSP è soddisfacibile, nel caso pessimo al momento dell'assegnazione dell'ultima variabile libera. Infatti, $\text{CLP}_0(\mathcal{D})$ include nella propria semantica la possibilità di utilizzare la funzione *label* mediante un predicato che porta lo stesso nome e che può essere applicato ad una lista di variabili che vengono assegnate nell'ordine in cui sono presenti nella lista.

Infine, si noti che la semantica di $\text{CLP}_0(\mathcal{D})$ è coerente con l'unificazione. In particolare, se il linguaggio di vincoli \mathcal{D} mette a disposizione un vincolo per imporre l'uguaglianza tra due valori, allora il vincolo è soddisfatto se e soltanto se le i termini corrispondenti ai due valori sono unificabili.

Un linguaggio di vincoli molto usato è *FD* (da *Finite Domains*), che è un linguaggio di vincoli in cui:

1. Il dominio D è composto da tutti i sottoinsiemi finiti di \mathbb{Z} , da cui il nome;
2. Tra gli atomi sono presenti i simboli univocamente associati ai numeri interi e, quindi, $1, 2, \dots$;
3. Tra gli atomi sono presenti i simboli univocamente associati alle operazioni aritmetiche tra interi e, quindi, $+, -$ e altri;
4. Tra i simboli di vincolo sono presenti i simboli $\#=$, $\#\backslash=$, $\#<$, $\#=<$ e altri, che permettono di imporre i comuni vincoli di confronto tra numeri interi e vengono di solito utilizzati come operatori infissi;
5. Tra i simboli di vincolo è presente *in*, tipicamente usato come operatore infisso, che permette di attribuire un dominio ad una variabile esprimendo il dominio come un'unione di intervalli espressi usando \dots e $\backslash/$, rispettivamente, per denotare gli intervalli e farne l'unione;
6. Tra i simboli di vincolo è presente il simbolo *ins*, tipicamente usato come operatore infisso, che permette di attribuire un dominio, esattamente come *in*, però ad un insieme di variabili elencate in una lista; e
7. Tra i simboli di vincolo sono presenti simboli per esprimere vincoli di utilizzo comune, tra i quali il simbolo *all_distinct* che permette di imporre che la lista di variabili utilizzate come argomento sia formata unicamente da variabili associate a termini tra loro tutti diversi.

Il linguaggio CLP₀(FD) estende Prolog₀ mediante i vincoli del linguaggio di vincoli FD. In modo analogo, il linguaggio CLP(FD) estende Prolog mediante FD. Per utilizzare SWI-Prolog come CLP(FD) è sufficiente includere il relativo modulo mediante il goal

```
: - use_module(library(clpfd)).
```

che viene tipicamente posizionato all'inizio del programma. L'utilizzo di CLP(FD) permette di scrivere un predicato che sia soddisfatto da una lista e dalla relativa lunghezza:

```
length([], 0).
length([_ | L], Length) :-  
    Length #>= 1,  
    N1 #= Length - 1,  
    length(L, N1).
```

Si noti che SWI-Prolog non permette di ridefinire questo predicato perché è uno dei predicatori *built-in* di basso livello del linguaggio.

Esempio 8. Il seguente programma CLP(FD) permette di enumerare tutte le soluzioni del problema di criptaritmetica *send + more = money*:

```
: - use_module(library(clpfd)).

send(Vars) :-  
    Vars = [S,E,N,D,M,O,R,Y],  
    Vars ins 0..9,  
    M #\= 0, S #\= 0,  
    all_distinct(Vars),  
    S*1000 + E*100 + N*10 + D +  
    M*1000 + O*100 + R*10 + E #=  
    M*10000 + O*1000 + N*100 + E*10 + Y,  
    label(Vars).
```

Un altro linguaggio di vincoli interessante è *Dif*. Questo linguaggio offre un solo vincolo **dif** che permette di imporre che il termine a primo argomento sia sempre diverso dal termine a secondo argomento, intendendo che due termini sono diversi se non sono unificabili. Nonostante questo linguaggio di vincoli sia minimale, è particolarmente rilevante perché CLP(*Dif*) permette spesso di ovviare ai noti problemi dei predicatori che implicano una negazione.

Esempio 9. Il seguente predicato permette di imporre che il primo argomento non sia mai incluso nella lista che viene utilizzata come secondo argomento:

```
: - use_module(library(dif)).

nevermember(_, []).
nevermember(X, [H | R]) :-  
    dif(X, H),
    nevermember(X, R).
```

8 Prolog e Logica

Nonostante il linguaggio Prolog sia l'esempio più tipico di un linguaggio di programmazione logica, non è ancora stato esplorato il legame tra Prolog e la Logica. In particolare, non è ancora stato affrontato il problema di capire se i risultati prodotti da un programma scritto in Prolog possono essere considerate conseguenze logiche valide dei fatti e delle regole incluse nel programma. Per affrontare questo problema, seguendo la strada tracciata da *Robert A. Kowalski*, è però prima necessario introdurre i linguaggi della logica dei predicati e descrivere il metodo della risoluzione.

Un linguaggio della **logica dei predicati** (o della **logica del primo ordine**) è un linguaggio che può essere costruito partendo da una **signature** $\langle P, C, F, V \rangle$ dove:

1. $P \neq \emptyset$ è un insieme di **simboli di predicato** (o **predicati**), ognuno dei quali è associato ad un'arità;
2. $C \neq \emptyset$ è un insieme di **simboli di costante** (o **costanti**);
3. F è un insieme di **simboli di funzione** (o **funzioni**), ognuno dei quali è associato ad un'arità; e
4. $V \neq \emptyset$ è un insieme infinito numerabile di **simboli di variabile** (o **variabili**).

Si noti che viene richiesto che $V \cap C = \emptyset$ e che gli insiemi non contengano le parentesi tonde, il punto e alcuni simboli che vengono comunemente utilizzati nei linguaggi di questo tipo per altri scopi, quali, ad esempio, \wedge , \vee e \top .

Partendo da una signature $\langle P, C, F, V \rangle$ è possibile costruire il relativo linguaggio della logica dei predicati mediante una struttura a due livelli. Il primo livello definisce la forma dei **termini (del primo ordine)**, esattamente come descritta in precedenza parlando del linguaggio dei termini. In particolare, usando i simboli di C come atomi con arità fissata a zero, i simboli di F come atomi con arità non nulla e i simboli di V . Poi, partendo dai termini, è possibile costruire il secondo livello della struttura del linguaggio. Questo livello contiene l'insieme delle **formule ben formate** (o **formule ben formulate**) del linguaggio. Le formule ben formulate di un linguaggio di questo tipo vengono chiamate **proposizioni** e vengono descritte partendo dai termini costruiti nel primo livello del linguaggio come segue:

1. Se t_1, t_2, \dots, t_n sono termini e p è un simbolo di predicato di arità $n \in \mathbb{N}_+$, allora $p(t_1, t_2, \dots, t_n)$ è una proposizione che viene comunemente detta *letterale affermato*;
2. I simboli \top (letto *top*) e \perp (letto *bottom*) sono proposizioni;
3. Se A è una proposizione, allora $\neg(A)$ è una proposizione e se A è un letterale affermato, allora $\neg(A)$ è un *letterale negato*;
4. Se A e B sono proposizioni, allora $(A \wedge B)$, $(A \vee B)$, $(A \implies B)$, and $(A \iff B)$ sono proposizioni;
5. Se A è una proposizione e $x \in V$ è una variabile, allora $\exists x.(A)$ e $\forall x.(A)$ sono proposizioni; e
6. Nient'altro è una proposizione.

Si noti che è consentito rimuovere le parentesi tonde se una proposizione è composta da un unico predicato o dai simboli \top e \perp e quando viene adottata la seguente precedenza per gli operatori binari: $\wedge, \vee, \implies, \iff$.

I quantificatori definiscono un *campo d'azione* (o *scope*) per una variabile e , data una proposizione, ogni occorrenza di una variabile nel campo di azione di un quantificatore viene detta *occorrenza vincolata*. Viceversa, ogni occorrenza di una variabile in una proposizione che non sia nel campo d'azione di un quantificatore viene detta *occorrenza libera*. Una proposizione viene detta *chiusa* se ha unicamente variabili con occorrenze vincolate.

Dato un insieme $D \neq \emptyset$, detto **dominio del linguaggio**, e una signature $\langle P, C, F, V \rangle$ è possibile associare un **valore di verità** (o **valore semantico**) alle proposizioni del linguaggio costruito mediante la disposizione scegliendo una **funzione di interpretazione** I con le seguenti caratteristiche:

1. Per ogni $c \in C$ esiste un elemento $I(c) \in D$ chiamato interpretazione del simbolo di costante c ;
2. Per ogni $f \in F$ di arità $n \in \mathbb{N}_+$ esiste una funzione $I(f) : D^n \rightarrow D$ chiamata interpretazione del simbolo di funzione f ; e
3. Per ogni $p \in P$ di arità $n \in \mathbb{N}_+$ esiste una relazione di arità n definita su D , $I(p) \subseteq D^n$, chiamata interpretazione del simbolo di predicato p .

Fissata una funzione di interpretazione e un'**assegnazione** $s : V \rightarrow D$ è possibile associare un'interpretazione ai termini come segue:

1. L'interpretazione di un simbolo di costante $c \in C$ (per l'assegnazione s) è $I_s(c) = I(c)$;
2. L'interpretazione di un simbolo di variabile $x \in V$ per l'assegnazione s è $I_s(x) = s(x)$;
3. L'interpretazione di un *termine strutturato* $t = f(t_1, t_2, \dots, t_n)$ per un assegnazione s è $I_s(t) = I(f)(I_s(t_1), I_s(t_2), \dots, I_s(t_n))$.

Fissata un'interpretazione I e un'assegnazione s è possibile dire quando l'interpretazione I soddisfa una proposizione A secondo l'assegnazione s , $(I, s) \models A$. Si noti subito che, a seconda dell'assegnazione considerata, una singola proposizione potrà essere soddisfatta da un'interpretazione I o meno. Per verificare se un'interpretazione I soddisfa una proposizione A secondo un'assegnazione s si utilizzano le seguenti regole, dove $s_{x \mapsto d}$ è l'assegnazione identica ad s tranne che per la variabile x che viene assegnata a d :

1. $(I, s) \models p(t_1, t_2, \dots, t_n)$ se e soltanto se $(I_s(t_1), I_s(t_2), \dots, I_s(t_n)) \in I(p)$
2. $(I, s) \models \top$ e $(I, s) \not\models \perp$;
3. $(I, s) \models \neg A$ se e soltanto se $(I, s) \not\models A$;
4. $(I, s) \models A \wedge B$ se e soltanto se $(I, s) \models A$ e $(I, s) \models B$;
5. $(I, s) \models A \vee B$ se e soltanto se $(I, s) \models A$ oppure $(I, s) \models B$;
6. $(I, s) \models A \implies B$ se e soltanto se $(I, s) \not\models A$ oppure $(I, s) \models B$;
7. $(I, s) \models A \iff B$ se e soltanto se $(I, s) \models A$ e $(I, s) \models B$, o $(I, s) \not\models A$ e $(I, s) \not\models B$;
8. $(I, s) \models \exists x.A$ se e soltanto se esiste un $d \in D$ tale che $(I, s_{x \mapsto d}) \models A$; e
9. $(I, s) \models \forall x.A$ se e soltanto se per ogni $d \in D$ vale $(I, s_{x \mapsto d}) \models A$.

Si noti che $(I, s) \models A$ viene anche letto: A è vera in I secondo l'assegnazione s . In particolare, si dirà che A è vera in I , $I \models A$, se A è vera in I per ogni sostituzione s . In questo caso, I viene detta **modello** di A . Infine, si dirà che A non è vera in I , $I \not\models A$, se non esiste alcuna sostituzione s tale per cui A è vera in I per s . In questo caso, I viene detta **contromodello** di A .

Si consideri una **teoria (del primo ordine)** T , che non è altro che un insieme di proposizioni. La teoria si dice vera in un'interpretazione I e secondo un'assegnazione s , $(I, s) \models T$, se e soltanto se per ogni proposizione P in T vale $(I, s) \models P$. Una teoria si dice vera in un'interpretazione I , $I \models T$, se la teoria è vera per ogni assegnazione. In questo caso, l'interpretazione I si dice modello della teoria. Analogamente è possibile definire quando un'interpretazione non è vera in una teoria e quindi definire i contromodelli di una teoria. Data una teoria T e una proposizione A , non necessariamente contenuta in T , si dice che A è **conseguenza logica** di T , $T \models A$, se per ogni interpretazione I e sostituzione s che rendono vera la teoria T vale anche che $(I, s) \models A$. Quindi, per ogni interpretazione e assegnazione che rendono vera la teoria anche la conseguenza logica risulta vera. Naturalmente, non viene richiesto che la conseguenza logica sia vera unicamente per le interpretazioni e le sostituzioni che rendono vera T .

Date una teoria T e una proposizione A è possibile studiare se $T \models A$ mediante il **metodo della risoluzione**, che è un particolare tipo di **metodo di refutazione** (o **metodo di riduzione all'assurdo**). I metodi di refutazione si basano sul fatto che $T \models A$ se e soltanto se non esistono un'interpretazione I e una sostituzione s tali che $(I, s) \models T \cup \{\neg A\}$.

Il metodo di risoluzione può essere applicato per studiare la **soddisfacibilità** di un insieme generico di proposizioni, quindi l'esistenza di un modello per l'insieme di proposizioni, mediante il metodo dovuto a *Albert Thoralf Skolem*. Però, nell'ambito della programmazione logica, si prevede che le teorie siano formate unicamente da proposizioni di un particolare tipo detto **clausole di Horn**, dal nome di *Alfred Horn*.

Una clausola di Horn è una proposizione in cui valgono le seguenti restrizioni che è possibile verificare a livello puramente sintattico:

1. Non sono ammessi i quantificatori esistenziali;
2. I quantificatori universali, se presenti, sono unicamente nella parte iniziale della proposizione;
3. La parte della proposizione che segue i quantificatori universali è formata unicamente da una disgiunzione di letterali; e
4. Non più di un letterale può essere positivo.

Quindi, una clausola di Horn è una proposizione del tipo:

$$\forall x_1. \forall x_2. \dots \forall x_n. (L_1 \vee L_2 \vee \dots \vee L_m) \quad (39)$$

e non più di uno dei letterali in $\{L_i\}_{i=1}^m$ è positivo. Una clausola di Horn si dice **clausola definita** se ammette un solo letterale positivo. Mentre una clausola di Horn si dice **clausola goal** se ammette zero letterali positivi. Si noti che si ammette, impropriamente, la possibilità di avere una **clausola (di Horn) vuota**, cioè con zero letterali, per indicare \perp . Infine, conviene notare che se viene tralasciato il vincolo che prevede la presenza di non più di un letterale positivo in una clausola di Horn, allora le proposizioni vengono dette semplicemente **clausole**.

Si consideri una teoria T formata unicamente da clausole di Horn, se esistono due clausole nella teoria

$$\forall x_1. \forall x_2. \dots \forall x_n. (L_1 \vee L_2 \vee \dots \vee L_m) \quad (40)$$

$$\forall y_1. \forall y_2. \dots \forall y_r. (M_1 \vee M_2 \vee \dots \vee M_s) \quad (41)$$

che non condividono variabili e tali che esistano $1 \leq i \leq n$ e $1 \leq j \leq s$ per cui $L_i\theta = \neg M_j\theta$, con $\theta = mgu(L_i, M_j)$, allora è possibile costruire la **clausola risolvente** R di L_i ed M_j :

1. Applicando la sostituzione θ ad ognuno dei letterali delle due clausole, esclusi L_i ed M_j ;
2. Unendo tutti i letterali ottenuti dall'applicazione della sostituzione θ , e quindi escludendo L_i ed M_j , mediante delle disgiunzioni; e
3. Quantificando universalmente su tutte le variabili contenute nei nuovi letterali ottenuti dall'applicazione della sostituzione.

La clausola risolvente R è stata costruita con la cosiddetta **regola di risoluzione** e si può dimostrare che la clausola risolvente R è conseguenza logica di T . Si noti che la regola di risoluzione può produrre la clausola vuota e, in questo caso, il risultato viene più propriamente indicato con \perp . In più, si noti che la clausola risolvente è ottenuta da due clausole che non condividono variabili, quindi se si considerano clausole che condividono variabili sarà sufficiente sostituire queste variabili in una delle due clausole con delle nuove variabili per poter applicare la regola. Infine, si noti che è stata applicata l'unificazione al linguaggio di termini formato da un alfabeto che contiene anche i simboli di predicato.

Data una teoria T formata unicamente da clausole definite è possibile dimostrare che una proposizione G del tipo:

$$\exists x_1. \exists x_2. \dots \exists x_n. L_1 \wedge L_2 \wedge \dots \wedge L_m \quad (42)$$

dove i letterali in $\{L_i\}_{i=1}^m$ sono tutti affermati, è conseguenza logica di T cercando di applicare iterativamente la regola di risoluzione alla teoria $T \cup \{\neg G\}$ fino ad ottenere la clausola vuota. Infatti, è vero che se $T \cup \{\neg G\} \vdash_{RES} \perp$ allora $T \cup \{\neg G\} \models \perp$ e quindi $T \models G$. Si noti che l'applicazione iterata della regola di risoluzione fa aumentare il numero di clausole contenute nella teoria perché, per ogni applicazione della regola, si estende la teoria con la clausola risolvente ottenuta.

Esempio 10. Si consideri un linguaggio della logica dei predicati con simboli di predicato in $P = \{p\}$, simboli di costante in $C = \{a, b, e\}$, simboli di funzione in $F = \{f\}$ e simboli di variabile in V che contiene tutte le parole che iniziano con una lettera maiuscola. Partendo dalla seguente teoria espressa in forma di clausole, dove, come di consueto, sono stati lasciati impliciti i quantificatori:

$$\begin{aligned} (1) \quad & p(e, X, X) \\ (2) \quad & p(f(X, Y), Z, f(X, W)) \vee \neg p(Y, Z, W) \end{aligned}$$

si vuole dimostrare che esiste un T tale che $p(f(a, e), T, f(a, f(b, e)))$. Per farlo, è possibile procedere *in avanti* (o *forward chaining*), come segue:

$$(3) \quad p(f(X, e), S, f(X, S)) \quad RES(1, 2) \quad \{S/W, e/Y, S/Z\}$$

da cui è possibile ottenere che la sostituzione $\{f(b, e)/S, a/X\}$ applicata a (3) porta al risultato cercato applicando la sostituzione $\{f(b, e)/T\}$ alla proposizione goal che si vuole dimostrare. In alternativa è possibile procedere *all'indietro* (*backward chaining*) e quindi per refutazione. Per prima cosa è necessario negare la proposizione goal che si vuole dimostrare ottenendo la la seguente clausola goal:

$$(3) \quad \neg p(f(a, e), T, f(a, f(b, e)))$$

che viene aggiunta alla teoria. A questo punto è possibile procedere come segue:

$$\begin{array}{ll} (4) & \neg p(e, U, f(b, e)) \quad RES(2, 3) \quad \{e/Y, U/Z, f(b, e)/W\} \\ (5) & \perp \qquad \qquad \qquad RES(1, 4) \end{array}$$

da cui si evince, procedendo a ritrso con le sostituzioni, che la conseguenza logica della teoria iniziale si ottiene applicando la sostituzione $\{f(b, e)/T\}$ alla proposizione goal che si vuole dimostrare.

Il legame tra la programmazione in linguaggio Prolog e il metodo della risoluzione risulta evidente riscrivendo le clausole definite come:

$$\forall x_1. \forall x_2. \dots \forall x_n. (P \Leftarrow \exists y_1. \exists y_2. \dots \exists y_r. L_1 \wedge L_2 \wedge \dots \wedge L_s) \quad (43)$$

dove P è l'unico letterale affermato della clausola definita considerata, e dipende solo dalle variabili in $\{x_i\}_{i=1}^n$, e sono state sfruttate alcune **equivalenze logiche** che riguardano i quantificatori, la negazione e l'implicazione per operare la riscrittura. Infatti:

$$\forall x_1. \forall x_2. \dots \forall x_n. \forall y_1. \forall y_2. \dots \forall y_r. (P \vee \neg L_1 \vee \neg L_2 \vee \dots \vee \neg L_s) \quad (44)$$

può essere riscritta come:

$$\forall x_1. \forall x_2. \dots \forall x_n. (P \vee \forall y_1. \forall y_2. \dots \forall y_r. (\neg L_1 \vee \neg L_2 \vee \dots \vee \neg L_s)) \quad (45)$$

e quindi:

$$\forall x_1. \forall x_2. \dots \forall x_n. (P \Leftarrow \neg(\forall y_1. \forall y_2. \dots \forall y_r. (\neg L_1 \vee \neg L_2 \vee \dots \vee \neg L_s))) \quad (46)$$

da cui si ottiene:

$$\forall x_1. \forall x_2. \dots \forall x_n. (P \Leftarrow \exists y_1. \exists y_2. \dots \exists y_r. \neg(\neg L_1 \vee \neg L_2 \vee \dots \vee \neg L_s)) \quad (47)$$

e, infine, risulta:

$$\forall x_1. \forall x_2. \dots \forall x_n. (P \Leftarrow \exists y_1. \exists y_2. \dots \exists y_r. (L_1 \wedge L_2 \wedge \dots \wedge L_s)) \quad (48)$$

Quindi, un programma scritto in Prolog non è altro che una teoria logica espressa mediante clausole definite. I fatti sono clausole definite senza letterali negati e le regole sono clausole definite con almeno un letterale negato. Il goal necessario per attivare la computazione di un programma scritto in Prolog non è altro che una proposizione per la quale si vuole verificare se sia conseguenza logica della teoria. La funzione semantica che descrive la computazione svolta dal programma a fronte di un goal non è altro che l'applicazione della regola di risoluzione secondo una strategia, rigida ma efficiente, che prende il nome di **SLD** (da **Selective Linear Definite clause resolution**). Quindi, qualsiasi sostituzione ottenuta da un programma scritto in Prolog a fronte di un goal è una conseguenza logica valida dei fatti e delle regole scritte nel programma.

9 Problemi di Pianificazione

Dato un **agente** in grado di compiere **azioni** nell'**ambiente** in cui è immerso, un **problema di pianificazione** riguarda la scelta di un **piano** di azioni che l'agente potrà svolgere per portare il **mondo** (ambiente e agente) dallo **stato attuale** in uno degli stati di **goal** che l'agente cerca di raggiungere. Per semplicità, la trattazione seguente è limitata a problemi di pianificazione simili a quelli supportati dallo *Stanford Research Institute Problem Solver (STRIPS)*, che rappresentano un particolare tipo di **problemi di pianificazione proposizionale**. In particolare, è possibile definire un problema di pianificazione come una quintupla:

$$\langle P, A, I, G, \tilde{G} \rangle \quad (49)$$

dove:

1. $P \neq \emptyset$ è un insieme finito di *proposizioni* (della logica proposizionale);
2. $A \neq \emptyset$ è un insieme finito di *azioni*;
3. $I \neq \emptyset$ è un sottoinsieme finito di P detto *stato iniziale* (del mondo);
4. $G \neq \emptyset$ è un sottoinsieme finito di P detto *goal asserito*; e
5. $\tilde{G} \neq \emptyset$ è un sottoinsieme finito di P detto *goal negato*.

In ogni istante, il mondo è caratterizzato da uno *stato* che viene descritto da un sottoinsieme di P che contiene tutte e sole le proposizioni che risultano vere nello stato. Quindi, ogni proposizione non contenuta nella descrizione di uno stato risulta falsa nello stato stesso. In particolare, I contiene tutte e sole le proposizioni ritenute vere nello stato iniziale del problema di pianificazione. Si noti, però, che G e \tilde{G} non sono descrizioni di uno stato, ma sono, rispettivamente, l'insieme delle proposizioni che devono essere vere e quelle che devono essere false in uno stato che possa essere considerato uno *stato di goal* del problema di pianificazione. Una soluzione del problema di pianificazione è una sequenza finita di azioni che, una volta eseguite dall'agente, portino il mondo dallo stato iniziale I ad uno stato di goal F che rispetti i requisiti espressi mediante G e \tilde{G} e, quindi, tale che:

$$G \subseteq F \quad \wedge \quad \tilde{G} \cap F = \emptyset \quad (50)$$

Ogni elemento di A è un'azione ed è una quintupla:

$$\langle n, R, \tilde{R}, T, \tilde{T} \rangle \quad (51)$$

dove:

1. n è un simbolo detto *nome* (o *identificativo*) dell'*azione* e determina univocamente ogni elemento di A ;
2. R è un sottoinsieme di P detto *precondizione asserita*;
3. \tilde{R} è un sottoinsieme di P detto *precondizione negata* tale che $R \cap \tilde{R} = \emptyset$;
4. T è un sottoinsieme di P detto *postcondizione asserita*; e
5. \tilde{T} è un sottoinsieme di P detto *postcondizione negata* tale che $T \cap \tilde{T} = \emptyset$.

Un'azione $a \in A$, con $a = \langle n, R, \tilde{R}, T, \tilde{T} \rangle$, può essere compiuta dall'agente se il mondo si trova in uno stato S tale che:

$$R \subseteq S \quad \wedge \quad \tilde{R} \cap S = \emptyset \quad (52)$$

Se l'agente compie l'azione a nello stato S , che soddisfa le precondizioni precedenti per ipotesi, allora il mondo si porta nello stato S' tale che:

$$S' = (S \setminus \tilde{T}) \cup T \quad (53)$$

Per risolvere un problema di pianificazione è possibile operare una **ricerca nello spazio degli stati** che parta dallo stato iniziale I e proceda fino all'identificazione delle soluzioni al problema, se esistono. La ricerca avviene mediante un insieme di *nodi*, ognuno dei quali contiene uno stato S , l'insieme degli stati attraversati per arrivare dallo stato iniziale fino allo stato S e la sequenza di azioni che ha permesso di arrivare fino allo stato S . Si noti che l'insieme degli stati attraversati per arrivare fino allo stato S è importante perché permette di capire se la ricerca sta per proseguire in uno stato già attraversato e, in questo caso, bloccarla in modo da garantire che la ricerca termini sempre. I nodi vengono organizzati in un **albero di ricerca** che ha come radice il nodo corrispondente allo stato I , il quale ha un insieme vuoto di stati già attraversati e viene raggiunto con una sequenza di azioni vuota. I nodi figli di un nodo N sono ottenuti andando a compiere tutte le azioni possibili partendo dal nodo N . Fissato un percorso nell'albero di ricerca, l'insieme dei nodi costruiti e non ancora scartati viene detto *frangia* (o *fringe*). In sintesi, la ricerca nello spazio degli stati può essere riassunta come segue:

1. La frangia viene inizializzata con lo stato iniziale I , un insieme di stati attraversati vuoto e la sequenza di azioni vuota;
2. Se la frangia è vuota, allora è stato provato che non esiste una soluzione al problema di pianificazione;
3. Viceversa, viene prelevato il nodo H alla testa della frangia e viene indicato con S il corrispondente stato, con L l'insieme degli stati attraversati e con P la corrispondente sequenza di azioni;
4. Se S è uno stato di goal, allora viene ritornato il piano ottenuto dalla sequenza di azioni P ;
5. Viceversa, lo stato S viene *espanso* generando l'insieme degli stati raggiungibili da S mediante l'applicazione di tutte le azioni disponibili;
6. Per ognuno degli stati S' raggiungibili da S mediante l'applicazione di un'azione a , se $S' \in L$ allora lo stato viene scartato;
7. Viceversa, viene costruito un nodo relativo ad S' usando come insieme degli stati attraversati $L \cup S$ ed estendendo la sequenza di azioni P con l'azione a , quindi il nuovo nodo viene aggiunto alla frangia; e
8. Si ritorna al punto 2.

Il metodo così sommariamente descritto ha almeno un punto in cui può essere raffinato. In particolare, non viene deciso in quale posizione della frangia verrà aggiunto un nuovo nodo ottenuto dall'espansione di un altro nodo.

Se ogni nuovo nodo viene:

- Aggiunto in fondo alla frangia, allora l'algoritmo di ricerca nello spazio degli stati viene detto **in ampiezza** (**Breadth-First Search** o **BFS**).
- Aggiunto in testa alla frangia, allora l'algoritmo di ricerca nello spazio degli stati viene detto **in profondità** (**Depth-First Search** o **DFS**).
- Aggiunto in modo che gli stati più vicini agli stati di goal siano più vicini alla testa della frangia, allora l'algoritmo di ricerca nello spazio degli stati viene detto **informato**, per distinguerlo dai precedenti due algoritmi che vengono detti **non informati**.

Si noti che tutti i metodi elencati hanno complessità computazionale temporale asintotica di caso pessimo esponenziale di ordine $O(b^d)$, dove $b = |A|$ e d è il numero di azioni necessarie per raggiungere, nel caso pessimo, uno stato di goal. Però, gli algoritmi hanno caratteristiche diverse da almeno altri due punti di vista:

1. BFS ha complessità computazionale spaziale asintotica di caso pessimo di ordine $O(b^d)$ e garantisce che, per problemi risolubili, venga ottenuta una soluzione composta dal numero minimo di azioni; ma
2. DFS ha complessità computazionale spaziale asintotica di caso pessimo di ordine $O(d)$ pur non garantendo, per problemi risolubili, che venga ottenuta una soluzione composta dal numero minimo di azioni.

Per ottenere un buon compromesso tra la BFS e la DFS, di solito si introduce la **ricerca ad approfondimenti successivi (della ricerca in profondità) (Iterative Deepening Depth-First Search, ID-DFS)** che lavora nel seguente modo:

1. Viene inizializzato il *parametro di profondità massima* N al valore uno;
2. Viene applicata la DFS fino alla costruzione di un albero di ricerca con non più di N livelli;
3. Se viene trovata una soluzione, allora viene immediatamente ritornata, senza necessariamente completare l'albero di ricerca; ma
4. Se non viene trovata una soluzione, allora viene incrementato N di uno e si ritorna al punto 2.

La ID-DFS ha la complessità computazionale spaziale asintotica di caso pessimo della DFS, ma garantisce che, per problemi risolubili, venga trovata una soluzione composta dal numero minimo di azioni, come accade per la BFS. Si noti che, a causa degli incrementi di N , la DFS viene applicata ad alberi che condividono la parte più vicina alla radice. Questo prevede che vengano costruiti più volte i nodi dei primi livelli degli alberi, senza però peggiorare la complessità computazionale temporale asintotica. Infatti, il numero di nodi di un albero finito è sempre di ordine $O(b^d)$, dove b è il numero massimo di figli di ogni nodo e d è la profondità dell'albero. Quindi, l'ordine del numero dei nodi di un albero di profondità finita dipende unicamente dal numero dei nodi nella frangia dell'albero.

La ricerca informata normalmente richiede che:

1. Il costo sia *definito positivo* e *additivo*; e
2. Venga definita una *funzione di costo* $c = f(N)$ per ogni nodo N che quantifichi il costo minimo necessario per partire dalla radice, passare per N e arrivare ad uno stato di goal.

In queste ipotesi, dato un nodo N ottenuto durante la costruzione dell'albero di ricerca, vale:

$$f(N) = h(N) + g(N) \quad (54)$$

dove $g(N)$ è il costo necessario per raggiungere il nodo N partendo da I e h è il costo necessario per raggiungere il più vicino stato di goal. Però, spesso, la funzione h non è disponibile e quindi si ricorre alla funzione $h^*(N)$ tale che:

$$0 \leq h^*(N) \leq h(N) \quad (55)$$

per ogni nodo N . La funzione h^* viene detta **funzione euristica** e se i nodi vengono inseriti nella frangia in senso crescente di $f^*(N) = h^*(N) + g(N)$, allora l'algoritmo di ricerca viene chiamato A^* . Si può dimostrare che A^* è ottimo nel senso che richiede il numero minimo di espansioni di nodi per ottenere una soluzione a costo minimo, se esiste.

Infine, si noti che la ricerca nello spazio degli stati che è stata descritta viene detta **in avanti** (o **forward chaining**) perché parte dallo stato iniziale fino a raggiungere uno stato di goal. Viceversa, è possibile modificare la ricerca in modo che parta dagli stati di goal e prosegua verso lo stato iniziale facendo, quindi, una ricerca **all'indietro** (o **backward chaining**), sostanzialmente, invertendo il ruolo di precondizioni e postcondizioni.

Appunti del Corso di Intelligenza Artificiale

Problemi di Soddisfacimento di Vincoli

Prof. Federico Bergenti

21 aprile 2024

1 Problemi di Soddisfacimento di Vincoli

Un problema di soddisfacimento di vincoli (Constraint Satisfaction Problem, CSP) è una tripla $\langle V, D, C \rangle$ in cui

- $V \neq \emptyset$ è un insieme non vuoto e finito di simboli detti variabili con $n = |V|$;
- $D \neq \emptyset$ è un insieme non vuoto di insiemi detti **dominî** delle variabili con $|D| \leq n$; e
- C è un insieme finito di **vincoli**.

Detta $dom : V \rightarrow D$ una funzione suriettiva totale che associa un dominio ad ogni variabile, si può parlare, per ogni variabile $x \in V$, del suo dominio $dom(x)$. In più, si suppone sempre che esista un ordinamento totale delle variabili e che questo ordinamento venga sempre usato in senso crescente quando vengono elencate le variabili. Normalmente, l'ordinamento utilizzato per le variabili è lasciato implicito ed evidente dal contesto. Utilizzando l'ordinamento delle variabili è possibile definire il dominio di un CSP \mathcal{P} come

$$dom(\mathcal{P}) = \prod_{x \in V} dom(x) \quad (1)$$

↓
 dominio del problema.
 di Soddisfacimento dei vincoli

Prodotto cartesiano
 dei dominî delle variabili

dove V è l'insieme delle variabili del CSP e le variabili vengono elencate nella costruzione del prodotto cartesiano in senso crescente secondo l'ordinamento scelto.

Dato un CSP con variabili in V , ogni vincolo $c \in C$ è una coppia $\langle V_c, \Delta_c \rangle$ dove $V_c \subseteq V$ è un insieme di variabili del CSP e Δ_c è

$$\Delta_c \subseteq \prod_{x \in V_c} dom(x) \quad (2)$$

dove le variabili vengono elencate nella costruzione del prodotto cartesiano in senso crescente secondo l'ordinamento scelto e Δ_c viene detto insieme degli **assegnamenti (parziali) consistenti (o ammissibili)** del vincolo c . Se $|V_c| = 1$, allora c viene detto **unario**, se $|V_c| = 2$, allora c viene detto **binario**, mentre c viene detto **globale** in tutti gli altri casi. Infine, si noti che $vars(c) = V_c$ è un modo per fare riferimento alle variabili di un vincolo e

$$dom(c) = \prod_{x \in V_c} dom(x) \quad (3)$$

↗ prodotto cartesiano dei dominî
 delle variabili coinvolte nel
 vincolo

viene detto **dominio del vincolo c** se si assume che le variabili vengano elencate in senso crescente secondo l'ordinamento scelto.

→ insieme delle n-pie con un numero di elementi pari alla cardinalità di V_c , che identifica l'insieme degli aggiornamenti delle variabili che sono consistenti con il vincolo

Fissato un CSP \mathcal{P} , è sempre possibile estendere Δ_c ad un sottoinsieme del dominio del CSP $\Delta = \text{dom}(\mathcal{P})$ aggiungendo alle ennuple di Δ_c le componenti mancanti. Siccome se una variabile non è coinvolta in un vincolo i suoi valori sono tutti e soli quelli del proprio dominio, le componenti aggiunte alle ennuple di Δ_c potranno assumere un valore qualsiasi nei dominî delle rispettive variabili. Detta img una funzione suriettiva totale che associa ad ogni vincolo l'estensione a Δ dell'insieme degli assegnamenti (parziali) consistenti, si può parlare, per ogni vincolo $c \in C$, del suo insieme degli **assegnamenti totali consistenti (o ammissibili)** $\text{img}(c) \subseteq \Delta$.

Se tutti i dominî delle variabili di un vincolo sono finiti, allora è possibile descrivere il vincolo elencando tutti gli assegnamenti parziali consistenti in modo **estensionale** e il vincolo viene detto **tabellare**. La rappresentazione tabellare diventa impraticabile anche per dominî con pochi elementi e quindi spesso si preferisce la rappresentazione **intensionale**.

Esempio 1. Si consideri un CSP con tre variabili, x , y e z , tali che $\text{dom}(x) = \text{dom}(y) = \text{dom}(z) = [1..6]$. L'insieme delle variabili del CSP è $V = \{x, y, z\}$ e il dominio del CSP è $\Delta = [1..6]^3$ dove, come di consueto, le variabili vengono ordinate in senso alfabetico. Il CSP contiene un vincolo c definito dalla proprietà

$$x \text{ e } z \text{ devono essere diversi ed entrambi pari}$$

Quindi, $c = \langle V_c, \Delta_c \rangle$ dove $V_c = \{x, z\}$ e Δ_c può essere espresso in forma estensionale come segue

$$\Delta_c = \{(2, 4), (2, 6), (4, 2), (4, 6), (6, 2), (6, 4)\} \quad (4)$$

dove, come di consueto, è stato utilizzato l'ordinamento alfabetico per le variabili. Si noti che l'estensione di Δ_c ad un sottoinsieme di Δ può essere espressa in forma estensionale come segue

$$\text{img}(c) = \{(2, 1, 4), (2, 2, 4), \dots, (6, 6, 4)\} \quad (5)$$

dove ogni elemento di Δ_c è stato utilizzato per produrre 6 elementi di $\text{img}(c)$. L'insieme degli assegnamenti parziali consistenti di c può essere espresso in forma intensionale come segue

$$\Delta_c = \{(x, z) : x \in I \wedge z \in I \wedge x \neq z \wedge x \bmod 2 = 0 \wedge z \bmod 2 = 0\} \quad (6)$$

e quindi

$$\text{img}(c) = \{(x, y, z) : x \in I \wedge y \in I \wedge z \in I \wedge x \neq z \wedge x \bmod 2 = 0 \wedge z \bmod 2 = 0\} \quad (7)$$

La possibilità di estendere gli insiemi degli assegnamenti parziali consistenti ad insiemi di assegnamenti totali consistenti permette di definire in modo semplice cosa si intenda per soluzione di un CSP. Dato un CSP $\mathcal{P} = \langle V, D, C \rangle$, l'insieme delle **soluzioni** di \mathcal{P} è

$$\text{sol}(\mathcal{P}) = \bigcap_{c \in C} \text{img}(c) \quad \begin{array}{l} \text{intersezione di tutte} \\ \text{le immagini di} \\ \text{tutti i vincoli che stiamo} \\ \text{considerando} \end{array} \quad (8)$$

e il \mathcal{P} si dice **risolubile (o risolvibile)** se questo insieme non è vuoto. Dati due CSP \mathcal{P}_1 e \mathcal{P}_2 definiti sullo stesso insieme di variabili, si dice che \mathcal{P}_1 è **equivalente** a \mathcal{P}_2 se $\text{sol}(\mathcal{P}_1) = \text{sol}(\mathcal{P}_2)$. Si noti che questa definizione può essere estesa a problemi con lo stesso numero di variabili rinominando opportunamente le variabili dei problemi.

Dato un CSP è interessante cercare delle trasformazioni in grado di generare dei CSP ad esso equivalenti ma che siano più utili per la ricerca delle soluzioni. In generale, però,

sono interessanti anche le trasformazioni in grado di produrre dei CSP non equivalenti a quello di partenza ma che permettano, comunque, di studiare le proprietà di quest'ultimo.

Esempio 2. Si consideri un CSP \mathcal{P} con due variabili x e y entrambe con dominio $[-10..10]$. Il CSP ha un unico vincolo descritto dalla proprietà $y = x^2$. Si noti che il CSP è risolubile perché, ad esempio, $x = -2$ e $y = 4$ è una soluzione. Il CSP \mathcal{P}_1 che restringe il dominio di y a $[0..10]$ è equivalente a \mathcal{P} perché è facile vedere che entrambi ammettono lo stesso insieme di soluzioni. Viceversa, il CSP \mathcal{P}_2 che restringe anche il dominio di x all'insieme $[0..10]$ non è equivalente a \mathcal{P} perché, ad esempio, $x = -2$ e $y = 4$ è una soluzione di \mathcal{P} ma non è una soluzione di \mathcal{P}_2 . Però, il CSP \mathcal{P}_2 , che è ottenuto da \mathcal{P} con una trasformazione detta **di rottura della simmetria** (o **symmetry breaking**), è comunque interessante perché una volta risolto è possibile ottenere le soluzioni di \mathcal{P} . In più, \mathcal{P}_2 ha un numero inferiore di elementi nel dominio di x e quindi la ricerca esaustiva delle soluzioni è facilitata.

Dato un CSP, ogni trasformazione che produce un CSP ad esso equivalente ma con meno vincoli o meno valori nei dominî viene detta **propagazione dei vincoli**. Se la trasformazione utilizza un sottoinsieme dei vincoli, allora viene detta **propagazione locale (dei vincoli)**. Se la trasformazione utilizza un sottoinsieme dei vincoli per rimuovere alcuni valori dai dominî delle variabili coinvolte nei vincoli considerati, allora viene detta **filtro (dei valori)**.

Normalmente, le trasformazioni tra CSP equivalenti vengono utilizzate per ottenere nuovi CSP con proprietà interessanti. Ad esempio, una trasformazione che spesso viene utilizzata permette di ottenere un CSP nodo-consistente equivalente ad un CSP dato. Si noti che un CSP si dice **nodo-consistente** (o **node-consistent**) se per ogni vincolo unario c vale la seguente proprietà

$$\forall v \in \text{dom}(x), v \in \Delta_c \quad \text{se per ogni vincolo unario } c, \text{ vale la proprietà che qualsiasi sia il valore nel dominio della variabile } x \text{ coinvolta nel vincolo, il valore appartiene al dominio di } c \quad (9)$$

dove x è l'unica variabile del vincolo c e Δ_c è l'insieme degli assegnamenti parziali consistenti di c .

Un **algoritmo di nodo-consistenza** lavora su un CSP per produrre un CSP nodo-consistente ad esso equivalente eliminando dai dominî delle variabili tutti i valori che non rispettano la definizione precedente. In più, un algoritmo di nodo-consistenza rimuove anche tutti i vincoli unari dal CSP perché, una volta eliminati i valori inconsistenti dai dominî, i vincoli unari non sono più necessari.

In modo simile, un **algoritmo di arco-consistenza** lavora su un CSP per produrre un CSP arco-consistente ad esso equivalente eliminando dai dominî delle variabili tutti i valori che non rispettano la definizione di arco-consistenza. Un CSP si dice **arco-consistente** (o **arc-consistent**) se per ogni vincolo binario c valgono le seguenti proprietà

$$\begin{aligned} \forall v_x \in \text{dom}(x), \exists v_y \in \text{dom}(y) : (v_x, v_y) \in \Delta_c \\ \forall v_y \in \text{dom}(y), \exists v_x \in \text{dom}(x) : (v_x, v_y) \in \Delta_c \end{aligned}$$

qualsiasi sia v_x , elemento del dominio di x , esiste un v_y , elemento del dominio di y , tali che la coppia v_x e v_y è un elemento delle nuple di questo vincolo binario
per ogni v_y nel dominio di y , esiste un v_x appartenente al dominio di x , tali che la coppia v_x e v_y è un elemento delle nuple del vincolo binario

dove x e y sono le due variabili del vincolo c e Δ_c è l'insieme degli assegnamenti parziali consistenti di c .

Si noti che esistono vari algoritmi di arco-consistenza che assumono che i dominî delle variabili siano finiti. L'algoritmo normalmente più utilizzato è AC-3, che ha una complessità temporale asintotica di caso pessimo di classe $O(e k^3)$, dove e è il numero di vincoli binari e k è la cardinalità massima dei dominî delle variabili coinvolte nei vincoli binari. Si noti, comunque, che la complessità temporale asintotica di caso pessimo di un algoritmo

di arco-consistenza ottimo è di classe $O(e k^2)$, a cui si può arrivare con l'algoritmo AC-4 mediante un significativo incremento della memoria utilizzata rispetto ad AC-3.

In più, si noti che la proprietà di arco-consistenza può essere estesa a vincoli globali introducendo la cosiddetta **iperarco-consistenza**. Infine, si noti che la nodo-consistenza coinvolge una sola variabile mentre la arco-consistenza coinvolge due variabili. Quindi è ragionevole aspettarsi di poter definire la cosiddetta **consistenza di percorso** costruendo percorsi di più di due variabili collegate da vincoli binari.

PRIMA di fare
qua, è PASSATO Alle.
SLIDE, appunti presi
a pg. 8

2 Problemi con Vincoli Polinomiali

La consistenza di nodo e la consistenza di arco permettono di rimuovere valori dai dominî delle variabili e sono spesso efficaci, specialmente se i vincoli sono espressi in forma tabellare. Viceversa, se i vincoli sono espressi in forma intensionale, ad esempio perché i dominî delle variabili non sono finiti, vengono spesso utilizzate altre forme di consistenza. Un esempio interessante a questo riguardo è rappresentato dai vincoli espressi mediante equazioni, disequazioni e disuguaglianze tra polinomi a coefficienti di variabili reali.

Si consideri un CSP \mathcal{P} con $n \in \mathbb{N}_+$ variabili in $V = \{x_i\}_{i=1}^n$ alle quali sono associati i dominî $dom(x_i) = [x_i, \bar{x}_i]$ rappresentati da intervalli chiusi in \mathbb{R} . Il problema può contenere anche più variabili, ma quelle di V sono quelle interessanti per i vincoli espressi mediante polinomi. Un vincolo c sulle variabili $V_c \subseteq V$ di \mathcal{P} si dice **polinomiale** se il suo insieme degli assegnamenti parziali consistenti può essere espresso in modo intensionale come

$$\Delta_c = \{\mathbf{x} \in dom(c) : p(\mathbf{x}) \odot q(\mathbf{x})\}$$

insieme delle variabili x , tale per cui
 $p(x)$ con p polinomio è in una certa relazione
del tipo $<, \leq, =, \neq, \geq, >$ con un altro polinomio
 $q(x)$

dove $\odot \in \{<, \leq, =, \neq, \geq, >\}$, $p : \mathbb{R}^k \rightarrow \mathbb{R}$ e $q : \mathbb{R}^k \rightarrow \mathbb{R}$ sono due funzioni polinomiali e $k = |V_c|$.

Si noti subito che ogni vincolo polinomiale può essere ridotto ad una delle due seguenti forme

$$\{\mathbf{x} \in dom(c) : p(\mathbf{x}) \geq 0\} \quad \text{oppure} \quad \{\mathbf{x} \in dom(c) : p(\mathbf{x}) > 0\} \quad (13)$$

utilizzando semplici manipolazioni algebriche e sfruttando il fatto che, data una funzione polinomiale $p : \mathbb{R}^k \rightarrow \mathbb{R}$, vale, per ogni $\mathbf{x} \in \mathbb{R}^k$,

$$p(\mathbf{x}) = 0 \iff p(\mathbf{x}) \leq 0 \wedge p(\mathbf{x}) \geq 0 \quad (14)$$

$$p(\mathbf{x}) \neq 0 \iff p^2(\mathbf{x}) > 0 \quad (15)$$

Infine, si noti che è sempre possibile riscrivere i vincoli in modo che tutte le variabili abbiano dominî definiti come intervalli chiusi di \mathbb{R}_+ mediante opportune trasformazioni affini operate sulle variabili. Infatti, essendo tutti i dominî delle variabili degli intervalli chiusi, è sufficiente traslare tutti gli intervalli per garantire che le variabili abbiano dominî definiti come intervalli chiusi di \mathbb{R}_+ .

Definizione: Un CSP si dice **consistente sugli intervalli** (o **bounds consistent**) se per ogni vincolo polinomiale c vale che per ognuna delle sue variabili x con dominio $[x, \bar{x}]$ esiste almeno un assegnamento parziale consistente del vincolo che abbia il valore \underline{x} per x ed, eventualmente un altro, assegnamento parziale consistente che abbia il valore \bar{x} per x .

Quindi, indipendentemente dalla consistenza dei valori all'interno dell'intervalle $[x, \bar{x}]$, la bounds consistency considera solo la consistenza dei valori ai due estremi di ogni intervallo coinvolto nel vincolo. Si noti che normalmente si è interessati al più piccolo intervallo, nel senso del contenimento, che garantisca la bounds consistency di una variabile in uno o più vincoli del problema.

Un algoritmo di bounds consistency è un algoritmo che trasforma un CSP in un secondo CSP ad esso equivalente in cui gli intervalli che definiscono i domini delle variabili garantiscono la bounds consistency. Come già discusso, un algoritmo di bounds consistency può considerare solo variabili definite su \mathbb{R}_+ e vincoli in forma di disequazione, stretta o meno. In queste ipotesi, è possibile definire un opportuno algoritmo di bounds consistency sfruttando la seguente proposizione.

Proposizione 1. *Dato $n \in \mathbb{N}$, sia $p : \mathbb{R}^n \rightarrow \mathbb{R}$ una funzione polinomiale a coefficienti reali positivi, se $\underline{\mathbf{x}} \in \mathbb{R}_+^n$ e $\bar{\mathbf{x}} \in \mathbb{R}_+^n$, allora per ogni $\mathbf{v} \in [\underline{\mathbf{x}}, \bar{\mathbf{x}}]$ vale*

$$p(\underline{\mathbf{x}}) \leq p(\mathbf{v}) \leq p(\bar{\mathbf{x}}) \quad (16)$$

nell'ipotesi non restrittiva che per ogni $1 \leq i \leq n$ valga $\underline{x}_i \leq \bar{x}_i$.

La descrizione generale di un algoritmo di bounds consistency che sfrutti la proposizione precedente richiede l'introduzione della notazione dei **multi-indici**. Quindi, anziché descrivere un algoritmo di bounds consistency nella sua generalità, verrà descritto sommariamente un metodo che permette di ridurre i domini delle variabili sfruttando il fatto che queste siano definite su intervalli chiusi di \mathbb{R}_+ .

Si consideri vincolo descritto dalla proprietà $p(\mathbf{x}) \geq 0$ nel caso in cui p sia una funzione lineare a coefficienti reali di $n \in \mathbb{N}_+$ variabili

$$p(\mathbf{x}) = \sum_{i=1}^n a_i x_i + b \quad (17)$$

Spiegazione
Lezione 16, 1:25:00

con, per semplicità $b \geq 0$. La proprietà che definisce il vincolo può essere riscritta spostando a destra tutti i termini di p con coefficienti negativi e quindi

$$\sum_{a_i > 0} a_i x_i + b \geq \sum_{a_i < 0} -a_i x_i \quad (18)$$

x sempre definiti positivi, il segno quindi lo decide a.

Fissata una variabile di indice k , se $a_k > 0$ è possibile scrivere

$$\sum_{i \neq k, a_i > 0} a_i x_i + a_k x_k + b \geq \sum_{a_i < 0} -a_i x_i \quad (19)$$

Sfruttando la proposizione precedente è possibile ottenere

$$\sum_{i \neq k, a_i > 0} a_i x_i \leq \sum_{i \neq k, a_i > 0} a_i \bar{x}_i = u \quad \text{UPPER} \quad l = \sum_{a_i < 0} -a_i \underline{x}_i \leq \sum_{a_i < 0} -a_i x_i \quad \text{LOWER} \quad (20)$$

e quindi la proprietà che descrive il vincolo impone che

$$x_k \geq \frac{l - u - b}{a_k} \quad (21)$$

per ogni variabile x_k che viene moltiplicata in p per un coefficiente positivo. Si noti subito che è semplice rimuovere l'ipotesi $b \geq 0$ e che, ragionamenti simili a quelli appena visti, consentono di identificare una disequazione da utilizzare per le variabili che vengono moltiplicate per un coefficiente negativo in p . Entrambe queste disequazioni possono anche essere utilizzate per trattare vincoli descritti mediante disequazioni strette, previo cambiamento del simbolo di diseguaglianza. Si capisce quindi facilmente come la possibilità di

stimare i valori massimi e minimi di un polinomio di una variabile in un intervallo chiuso sia di fondamentale importanza per gli algoritmi di bounds consistency.

Tutte queste disequazioni possono essere utilizzate ripetutamente per restringere i dominî delle variabili del vincolo considerato fino al raggiungimento di almeno una delle seguenti condizioni:

1. L'applicazione delle disequazioni non genera nuovi restringimenti dei dominî e quindi è possibile cercare le soluzioni del CSP nei nuovi dominî; oppure
2. L'applicazione delle disequazioni ha svuotato almeno uno dei dominî e quindi è stato dimostrato che il CSP non ammette soluzioni.

I due esempi che seguono mostrano come applicare questo metodo nei due casi appena descritti. In particolare, il seguente esempio consente di restringere i dominî delle variabili.

Esempio 3. Si consideri un CSP con tre variabili x , y e z i cui dominî sono $\text{dom}(x) = \text{dom}(y) = \text{dom}(z) = [1, 10]$. Si consideri il vincolo definito dalla proprietà

$$2x + 3y - z \leq 1 \quad (22)$$

La proprietà che definisce il vincolo può essere riscritta come

$$z + 1 \geq 2x + 3y \quad \xrightarrow{\begin{array}{l} \text{valuto il caso "sotto"} \\ z+1 \geq 2 \cdot 1 + 3 \cdot 1 > 5 \end{array}} \quad (23)$$

e quindi

$$z + 1 \geq 2x + 3y \geq 5 \quad (24)$$

da cui si evince che $z \geq 4$. Ma, ragionando sulla variabile x si ottiene \rightarrow valutiamo quindi l'upper: $\bar{z} = 10$

$$\begin{aligned} 11 &\geq z + 1 \geq 2x + 3y \geq 2x + 3 \\ 11 &\geq 10 + 1 \end{aligned} \quad (25)$$

da cui si evince che $x \leq 4$. Infine, ragionando sulla variabile y si ottiene

$$11 \geq z + 1 \geq 2x + 3y \geq 2 + 3y \quad (26)$$

da cui si evince che $y \leq 3$. Si noti che questi ragionamenti possono essere iterati finché nessun dominio viene più modificato oppure finché almeno uno dei dominî non diventa l'insieme vuoto. Nel primo caso si è ottenuto un CSP bounds consistent equivalente al CSP iniziale, mentre nel secondo caso è stato dimostrato che il CSP iniziale non ammette soluzioni. Provando ad iterare i ragionamenti fatti si nota subito che i dominî non vengono ulteriormente ridotti e quindi è stato ottenuto un nuovo CSP, equivalente a quello iniziale ma bounds consistent, con tre variabili x , y e z aventi dominî

$$\text{dom}(x) = [1..4] \quad \text{dom}(y) = [1..3] \quad \text{dom}(z) = [4..10] \quad (27)$$

e un solo vincolo che richiede che valga la proprietà $2x + 3y - z \leq 1$. Si noti che il vincolo non può essere rimosso perché la proprietà di bounds consistency non fornisce informazioni sulla consistenza dei valori interni agli intervalli considerati. La bounds consistency studia unicamente gli estremi dei dominî delle variabili.

Il seguente esempio mostra come sia possibile stabilire che il CSP considerato non ammette soluzioni. Infatti, l'applicazione delle disequazioni studiate in precedenza consente di ridurre all'insieme vuoto il dominio di almeno una variabile, terminando il processo di riduzione e stabilendo che il CSP non ammette soluzioni.

✓ **Esempio 4.** Si consideri un CSP con tre variabili x , y e z i cui dominî sono $dom(x) = dom(y) = dom(z) = [1, 10]$. Si consideri il vincolo definito dalla proprietà

$$12x + 8y - z \leq 1 \quad (28)$$

La proprietà che definisce il vincolo può essere riscritta come

$$z + 1 \geq 12x + 8y \geq 20 \quad (29)$$

e quindi

$$z + 1 \geq 12x + 8y \geq 20 \quad (30)$$

da cui si evince che il vincolo richiede che $z \geq 19$. Però, $z \leq 10$ per ipotesi e quindi è stato facilmente dimostrato che il CSP non ammette soluzioni.

Si noti che il metodo per la riduzione dei dominî delle variabili descritto sommariamente nel caso di funzioni lineari può essere esteso facilmente al caso di funzioni polinomiali non lineari ricordando che, nelle ipotesi considerate, se una variabile x ha per dominio $dom(x) = [\underline{x}, \bar{x}] \subseteq \mathbb{R}_+$, allora se $m \in \mathbb{N}_+$

$$\underline{x}^{m-1} x \leq x^m \leq \bar{x}^{m-1} \bar{x} \quad (31)$$

e quindi i termini non lineari del tipo x^m possono essere ridotti a termini lineari. In modo simile, se una seconda variabile y ha dominio $dom(y) = [\underline{y}, \bar{y}] \subseteq \mathbb{R}_+$, allora

$$\underline{x}\underline{y} \leq xy \leq \bar{x}\bar{y} \quad \text{e} \quad \underline{y}\underline{x} \leq yx \leq \bar{y}\bar{x} \quad (32)$$

e quindi i termini non lineari del tipo xy possono essere ridotti a termini lineari.

I vincoli polinomiali vengono spesso utilizzati restringendo i dominî delle variabili a intervalli in \mathbb{Z}_+ e richiedendo che i coefficienti delle funzioni polinomi siano in \mathbb{Z} . In questo caso si parla di **vincoli polinomiali a dominî finiti** ed è possibile sfruttare le particolarità di questo tipo di vincoli per definire degli algoritmi di propagazione dei vincoli efficaci ed efficienti. In particolare, il fatto che le variabili abbiano dominî in \mathbb{Z}_+ e il fatto che i coefficienti delle funzioni polinomiali siano in \mathbb{Z} permette di ridurre lo studio di questo tipo di vincoli a sole disequazioni non strette perché vale

$$p(\mathbf{x}) > 0 \iff p(\mathbf{x}) - 1 \geq 0 \quad (33)$$

per una qualsiasi funzione polinomiale a coefficienti interi e variabili intere positive. In più, si noti che, siccome le variabili sono ristrette ad assumere solo valori interi, in questo caso viene adottata la notazione degli intervalli interi e si scrive $[x.. \bar{x}]$, con $x \in \mathbb{Z}$, $\bar{x} \in \mathbb{Z}$ e $x \leq \bar{x}$, per indicare l'intervallo contenente tutti gli interi maggiori o uguali a \underline{x} e minori o uguali a \bar{x} . Naturalmente, $[x.. \bar{x}] = \emptyset$ se $x > \bar{x}$.

LEZIONE 15: SLIDE DA 50 A 58

Tutte le volte che si vuole cercare di risolvere un CSP, la cosa più semplice che si possa fare è utilizzare una tecnica che prende il nome di Generate & test, che è la tecnica più generale ma più sfortunata, ma più tornare utile quando si vuole confrontare il risultato di una tecnica raffinata, nell'ipotesi che si possano enumerare i valori nei domini delle variabili in modo efficace. Quindi partiamo dall'ipotesi che il dominio delle variabili siano abbastanza piccoli (come cardinalità) da poter enumerare tutti i valori possibili in memoria, per ogni variabile noi avremo tutti i valori possibili, scritti da qualche parte.

1) affronteremo sempre CSP con variabili con domini finiti, che però nella pratica sono quelli più utilizzati;

2) utilizziamo domini non troppo grandi, pratica abbastanza comune anche nel quotidiano;

Ipotizzando i punti 1 e 2 possiamo ipotizzare di applicare il G&T, che comunque richiederà troppo tempo ma ci può aiutare a capire dove andremo a finire.

(Vedi codice slide 51)

G&T è una possibilità, non è detto che sia quella ottima, non è detto che non si possa fare meglio: nonostante sia una tecnica grezza, nel caso pessimo di complessità è ottima.

Standard Backtracking è una tecnica che modificando effettivamente pochissimo codice, si hanno guadagni di performance enormi.

L'idea dello SBT è non fare le verifiche dei vincoli solo quando ho un assegnamento completo, tutte le variabili sono state assegnate ad un valore, ma anche quando ho gli assegnamenti parziali (almeno una variabile è stata assegnata) a questo punto posso verificare su quella variabile tutti i vincoli unari, che coinvolgono quella variabile, se sono tutti rispettati vado avanti, assegno la seconda, verifico i vincoli unari sulla singoli e i vincoli binari sulla coppia, assegno una terza, verifico i vincoli unari, binari e globali; continuo così fino a che qualcosa fallisce (e quindi torno indietro, faccio backtracking) oppure arrivo in fondo.

L'unica verifica che faccio è che la verifica del soddisfacimento dei vincoli, lo metto dentro all'assegnamento: assegno la variabile al valore, se i vincoli non sono violati, vado avanti e quindi index ++; altrimenti butto l'assegnamento e riassegno e riprovo.

Quando index vale n, allora ho fatto un assegnamento completo.

Come complessità temporale sono uguali (G&T), come complessità di calcolo sembra pure peggio.

LEZIONE 16: SLIDE DA 59 A 77

Fare nodo consistenza togliere i valori da i domini che non rispettano i valori unari, una volta tolta questi valori, i vincoli unari non servono più e quindi vanno tolti; fare nodo consistenza può essere visto come far è un filtro iniziale.

Attenzione che fare arco consistenza, toglie altri valori e che quindi nuovi vincoli unari si debbano ritogliere.

Una versione più leggera dell'arco consistenza è il forward checking: stessa idea dell'arco consistenza, prendo dei vincoli binari e verifico che ogni valore abbia un supporto, ma contrariamente a quello che fa l'arco consistenza, che quando si toglie un valore perché non ha un supporto, viene riapplicata a tutte le variabili; il forward checking si limita a dire: guardo avanti, se c'è qualcosa da togliere, lo tolgo ma una volta fatto ciò, non riparto a cercare di togliere altre cose. Questo fa sì che dipenda molto il risultato dall'ordine in cui si sceglie come fare forward checking, volendo si mette un'euristica che sceglie il nome delle variabili ma in realtà quello che si fa di solito è fare forward checking per fare una cosa veloce poco impattante dal punto di vista computazionale che toglie qualche valore.